```python
from pattern_utilities import generate_n_random_patterns
from hopfield import StochasticHopfieldNetwork
import numpy as np


def main():
    n_bits = 200
    n_patterns = 7
    noise_parameter = 2
    T = int(2e5)
    n_iterations = 100

    patterns = generate_n_random_patterns(n_patterns=n_patterns, n_bits=n_bits)

    network = StochasticHopfieldNetwork()
    network.set_patterns(patterns)
    network.set_diagonal_weights_rule("zero")
    network.set_noise_parameter(noise_parameter)
    network.generate_weights()

    pattern1 = patterns[0, :]
    updated_pattern = pattern1.copy()

    m = np.zeros(n_iterations)
    for i in range(n_iterations):
        print(i)
        for t in range(T):
            updated_pattern = network.update_random_neuron(updated_pattern)
            m[i] += np.inner(updated_pattern, pattern1)
        m[i] /= n_bits
        m[i] /= T

    m_estimate = sum(m) / n_iterations
    print("{:.3f}".format(m_estimate))


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

Listing 1: Main method for computing the order parameter.

```python
import numpy as np
from scipy import stats
from abc import ABC, abstractmethod


class HopfieldNetwork(ABC):
```

```
    def set_diagonal_weights_rule(self, diagonal_weights_rule):
        if diagonal_weights_rule == "zero":
            self.diagonal_weights_equal_zero = True
        elif diagonal_weights_rule == "non-zero":
            self.diagonal_weights_equal_zero = False
        else:
            raise KeyError(
                    diagonal_weights_rule
                    + " not a valid diagonal_weights_rule"
                    )

    def set_patterns(self, patterns):
        self.patterns = patterns

    def generate_weights(self):
        _, n_bits = self.patterns.shape
        self.weights = np.zeros((n_bits, n_bits))
        for pattern in self.patterns:
            self.weights += np.outer(pattern, pattern)
        self.weights /= n_bits

        if self.diagonal_weights_equal_zero:
            np.fill_diagonal(self.weights, 0)

    def asynchronous_update(self, pattern, n_updates):
        updated_pattern = pattern.copy()
        for i in range(n_updates):
            updated_pattern = self.update_random_neuron(updated_pattern)
        return updated_pattern

    def update_random_neuron(self, pattern):
        n_bits = pattern.shape
        neuron_index = np.random.randint(n_bits)
        return self.update_neuron(pattern, neuron_index)

    def update_neuron(self, pattern, neuron_index):
        weights_i = self.weights[neuron_index, :]
        local_field = np.inner(weights_i, pattern)
        updated_bit = self.get_state_of_local_field(local_field)
        updated_pattern = pattern.copy()
        updated_pattern[neuron_index] = updated_bit
        return updated_pattern

    @abstractmethod
    def get_state_of_local_field(self, local_field):
        pass


class DeterministicHopfieldNetwork(HopfieldNetwork):
```

```
    def get_state_of_local_field(self, local_field):
        return sign_zero_returns_one(local_field)


class StochasticHopfieldNetwork(HopfieldNetwork):
    def get_state_of_local_field(self, local_field):
        p = 1/(1 + np.exp(-2*self.noise_parameter*local_field))
        rand = stats.bernoulli.rvs(p)
        return 1 if rand else -1

    def set_noise_parameter(self, noise_parameter):
        self.noise_parameter = noise_parameter


def sign_zero_returns_one(value):
    return 1 if value >= 0 else -1
```

Listing 2: Classes for creating Hopfield Networks.

```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt


def generate_n_random_patterns(n_patterns, n_bits):
    random_0s_and_1s = stats.bernoulli.rvs(0.5, size=(n_patterns, n_bits))
    random_minus_1s_and_1s = 2*random_0s_and_1s - 1
    return random_minus_1s_and_1s


def get_index_of_equal_pattern(pattern_to_match, patterns):
    for index, pattern in enumerate(patterns):
        n_different_bits = get_n_different_bits(pattern_to_match, pattern)
        if n_different_bits == 0:
            return index
    return -1


def get_n_different_bits(pattern1, pattern2):
    return sum(pattern1 != pattern2)


def vector_to_typewriter(vector, n_columns):
    return np.reshape(vector, (-1, n_columns))


def print_typewriter_pattern(pattern, n_columns):
    print_pattern(vector_to_typewriter(pattern, n_columns))

```

```
31
32 def print_pattern(pattern):
33     np.set_printoptions(formatter={"float_kind": lambda x: "%.4f" % x})
34     print(repr(pattern), sep=", ")
35
36
37 def plot_pattern(pattern):
38     plt.imshow(pattern, cmap="Greys")
39     plt.tick_params(
40             axis="both",
41             which="both",
42             bottom=False,
43             top=False,
44             left=False,
45             labelbottom=False,
46             labelleft=False)
```

Listing 3: Help module for handling patterns.