

ising-MC-efficient

October 29, 2026

1 2D Ising model Monte Carlo

MC code to simulate Ising model in 2D

$$H = -J \sum_{\langle ij \rangle} s_i s_j - h \sum_i s_i$$

This model, for $J > 0$ has a Curie temperature of

$$T_c = \frac{2J}{\ln(\sqrt{2} + 1)} \approx 2.269J$$

Now coded with more efficient lookup tables for Arrhenius factors.

```
[1]: import numpy as np
rng = np.random.default_rng()
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [6, 6]
from numba import jit
from tqdm.auto import trange

[2]: # simple ferromagnetic coupling
param = {'J': 1.0, 'h': 0.0}

[3]: # set up cell and PBC
Nx, Ny = 256, 256
N = Nx*Ny

d, z = 2, 4 # dimensionality and coordination number
site = np.zeros((N, d), dtype=int)
neigh = np.zeros((N, z), dtype=int)

n = 0
for i in range(Nx):
    for j in range(Ny):
        site[n] = i, j
        n += 1

for n in range(N):
    i, j = site[n]
```

```

neigh[n, 0] = ((i+1)%Nx)*Ny + j
neigh[n, 1] = ((i-1)%Nx)*Ny + j
neigh[n, 2] = i*Ny + (j+1)%Ny
neigh[n, 3] = i*Ny + (j-1)%Ny

```

```

[4]: def arrhenius(kT, J=param['J'], h=param['h'], z=z):
    """Returns a matrix  $arr[si.sj, si] = \exp(-(2J*(si.sj) + 2h*si)/kT)$  that is
    the Arrhenius factor where
     $si.sj = \sum_j si*sj$  and  $si$ . Since we flip  $si \rightarrow -si$ , this is the Arrhenius
    factor for the change in
    energy with such a flip."""
    arr = np.zeros((2*z+1, 3))
    for sisj in range(-z, z+1):
        for si in (-1,1):
            arr[sisj, si] = np.exp(-2*(J*sij + h*si)/kT)
    return arr

```

```

[5]: # neighbor list check
if (False):
    for n in range(N):
        print(f'{site[n]}:')
        for m in neigh[n]:
            print(f' {site[m]}')

```

```

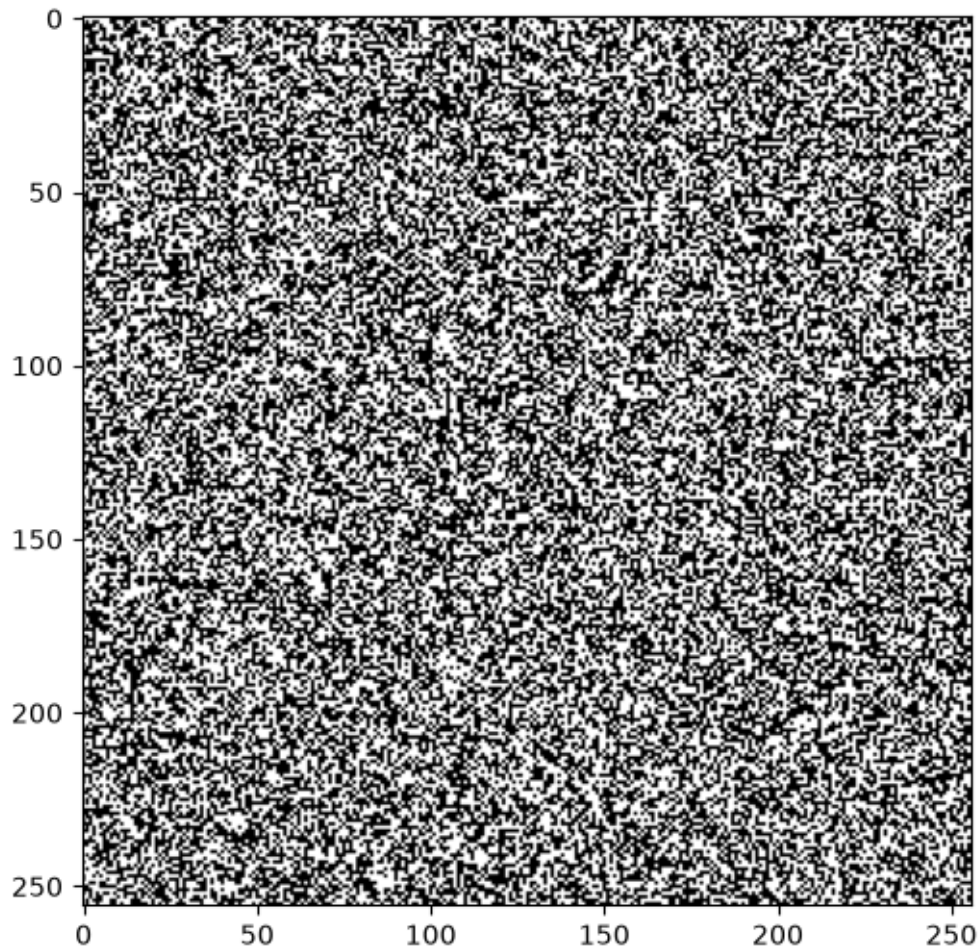
[6]: # initialize our spins on a square lattice
spin = rng.choice((-1, 1), size=N)

```

```

[7]: # visualize
plt.imshow(spin.reshape(Nx, Ny), cmap="grey");

```



```
[8]: @jit(nopython=True)
def ene(spin, neigh, J=param['J'], h=param['h']):
    """Computes the energy for a given configuration `spin` using `neigh` as a
    ↪neighbor
    list, with `J` and `h` as our parameters"""
    N = neigh.shape[0]
    z = neigh.shape[1]
    E_J = 0
    for n in range(N):
        neigh_list = neigh[n]
        s = spin[n]
        for i in range(z):
            if s==spin[neigh_list[i]]:
                E_J += 1
            else:
                E_J += -1
    return -0.5*J*E_J - h*sum(spin)
```

```
[9]: ene(spin, neigh)
```

```
[9]: 324.0
```

```
[10]: @jit(nopython=True)
# def MC(spin, kT, urand, neigh, J=param['J'], h=param['h']):
def MC(spin, kT, urand, neigh, arr):
    """Run MC for Ising spins where we attempt to flip each spin in order.
    Returns acceptance ratio only, and overwrites spin in place."""
    N = neigh.shape[0]
    z = neigh.shape[1]

    accept = 0
    for i in range(len(urand)):
        u = urand[i]
        n = i%N # spin to attempt to flip
        s = spin[n]
        sisj = 0
        for j in range(z):
            m = neigh[n, j]
            if s == spin[m]:
                sisj += 1
            else:
                sisj += -1
        if u < arr[sisj, s]:
            spin[n] = -s
            accept += 1
    return accept/len(urand)
```

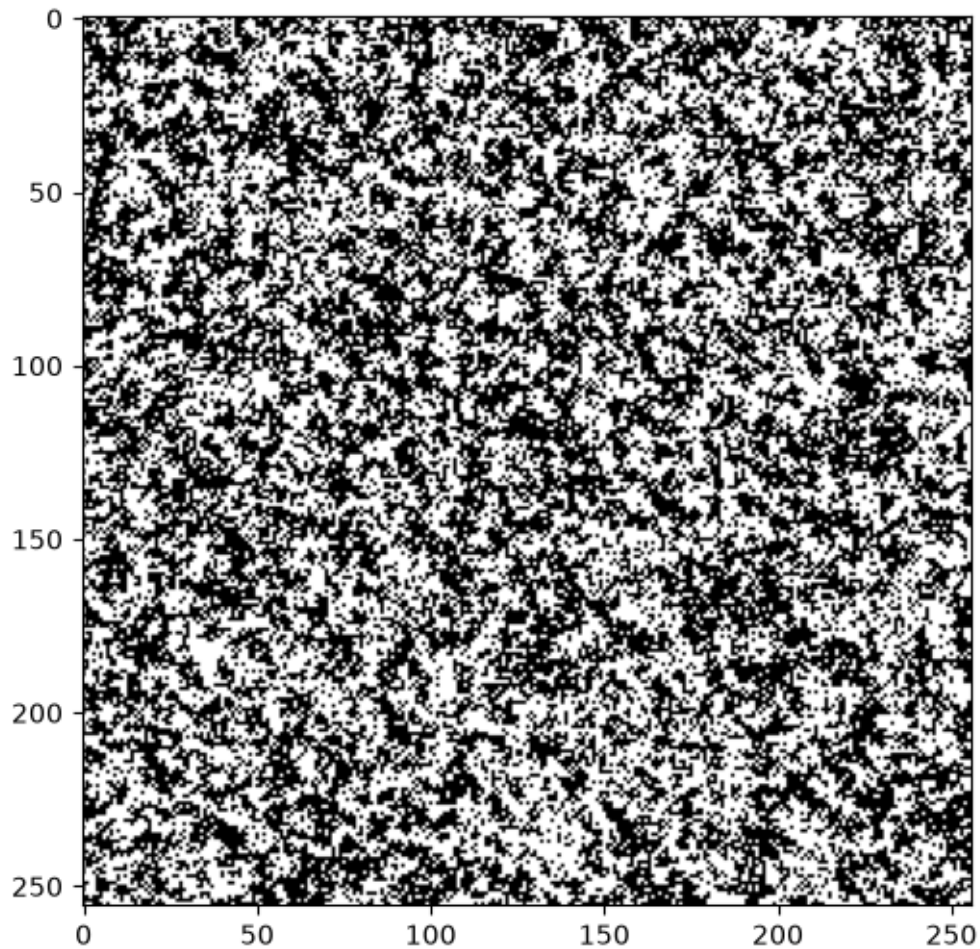
```
[11]: N_MC = N

kT = 2.5
arr = arrhenius(kT)
E = ene(spin, neigh)
urand = rng.uniform(size=(N_MC))
accept = MC(spin, kT, urand, neigh, arr)
```

```
[12]: accept, np.average(spin)
```

```
[12]: (0.5977020263671875, np.float64(0.002685546875))
```

```
[13]: # visualize
plt.imshow(spin.reshape(Nx, Ny), cmap="grey");
```



```
[14]: ene(spin, neigh)
```

```
[14]: -32412.0
```

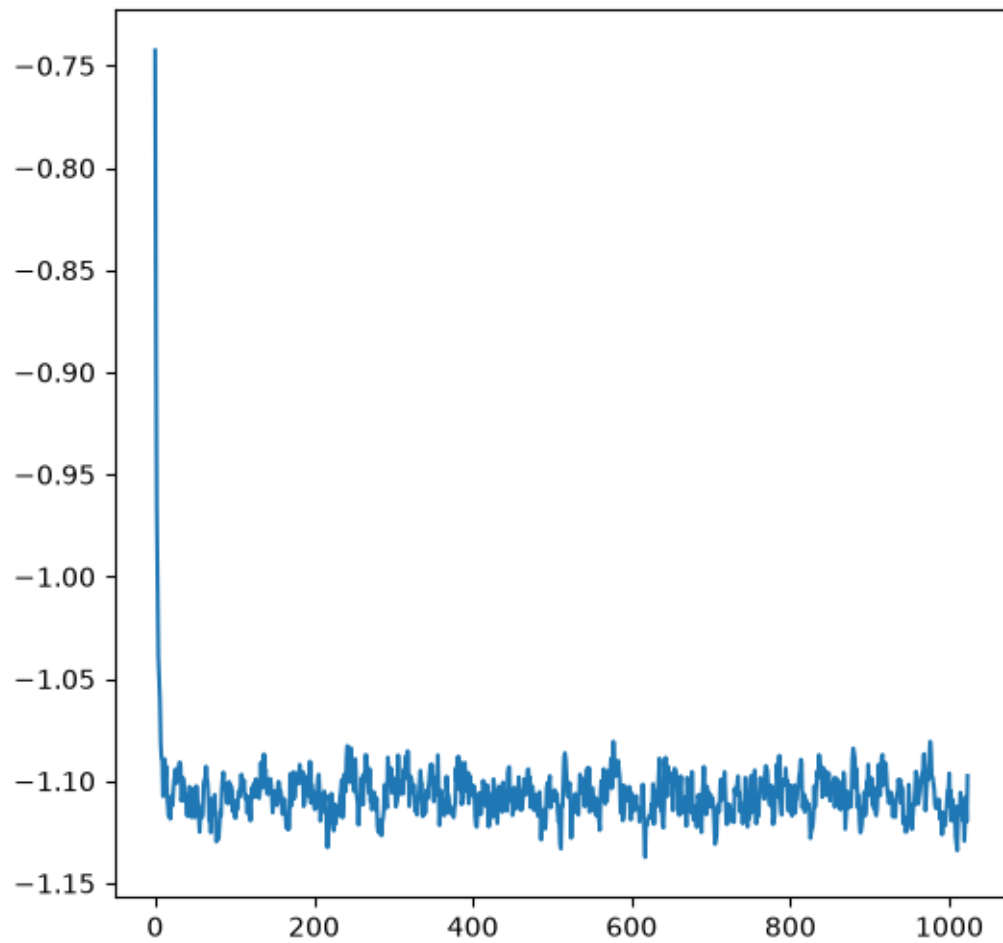
```
[15]: N_sweeps = 1024
      E_list, accept_list, m_list = [], [], []
      for _ in trange(N_sweeps):
          accept = MC(spin, kT, rng.uniform(size=N), neigh, arr)
          E_list.append(ene(spin, neigh)/N)
          accept_list.append(accept)
          m_list.append(np.average(spin))
      E_list = np.array(E_list)
      accept_list = np.array(accept_list)
      m_list = np.array(m_list)
```

```
0%|          | 0/1024 [00:00<?, ?it/s]
```

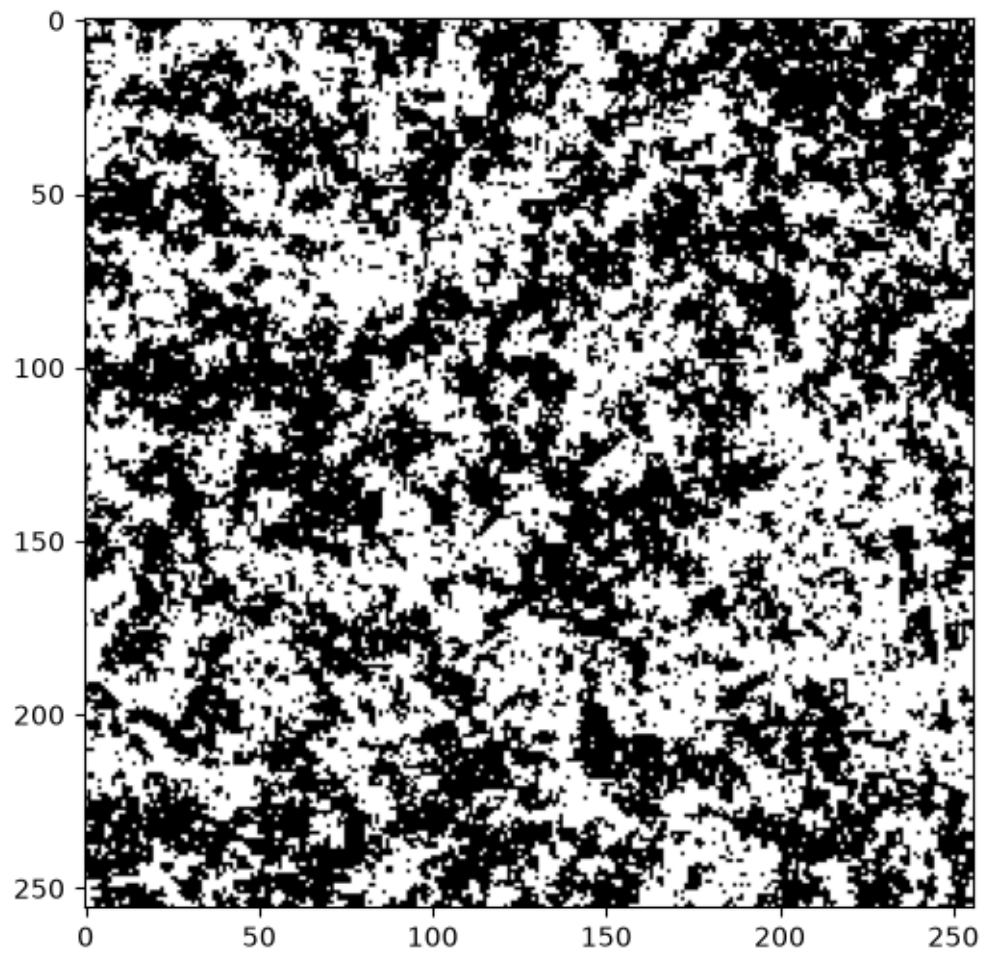
```
[16]: np.average(accept_list)
```

```
[16]: np.float64(0.32323339581489563)
```

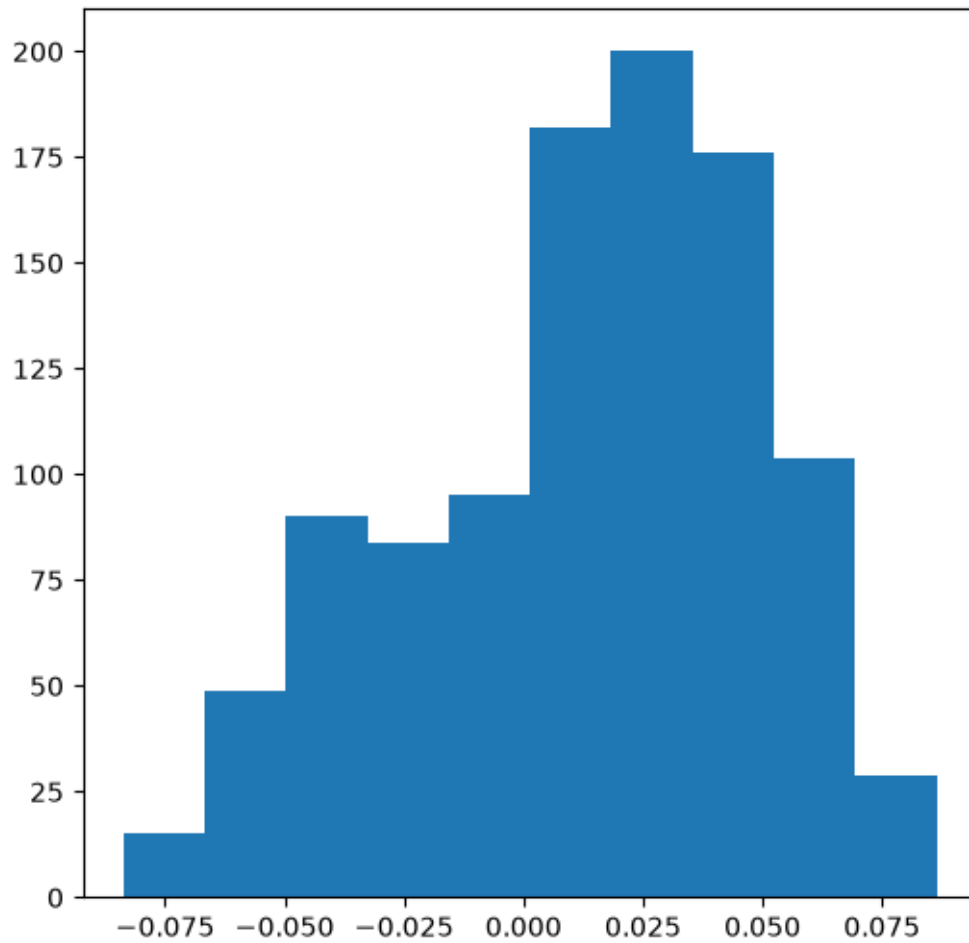
```
[17]: plt.plot(E_list);
```



```
[18]: plt.imshow(spin.reshape(Nx, Ny), cmap='grey');
```



```
[19]: plt.hist(m_list);
```



```
[20]: kT = 2.0
      arr = arrhenius(kT)

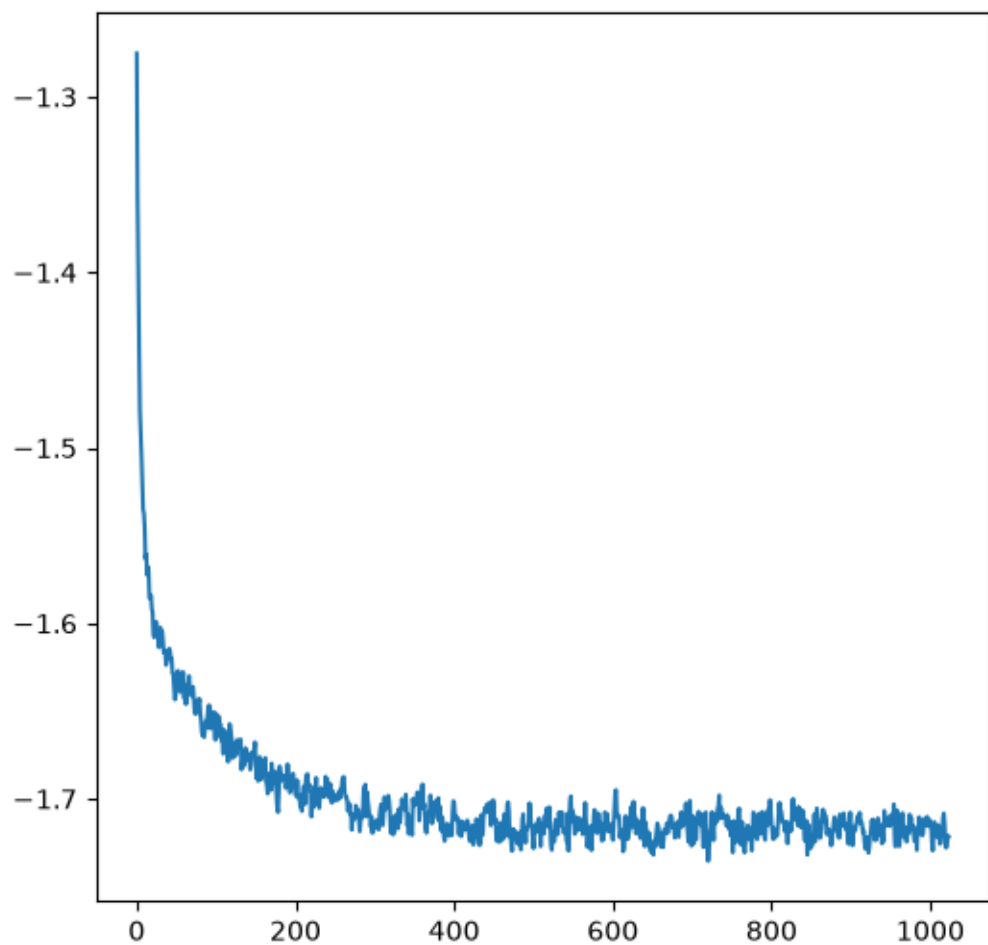
      N_sweeps = 1024
      E_list, accept_list, m_list = [], [], []
      for _ in trange(N_sweeps):
          accept = MC(spin, kT, rng.uniform(size=N), neigh, arr)
          E_list.append(ene(spin, neigh)/N)
          accept_list.append(accept)
          m_list.append(np.average(spin))
      E_list = np.array(E_list)
      accept_list = np.array(accept_list)
      m_list = np.array(m_list)

      0%|          | 0/1024 [00:00<?, ?it/s]
```

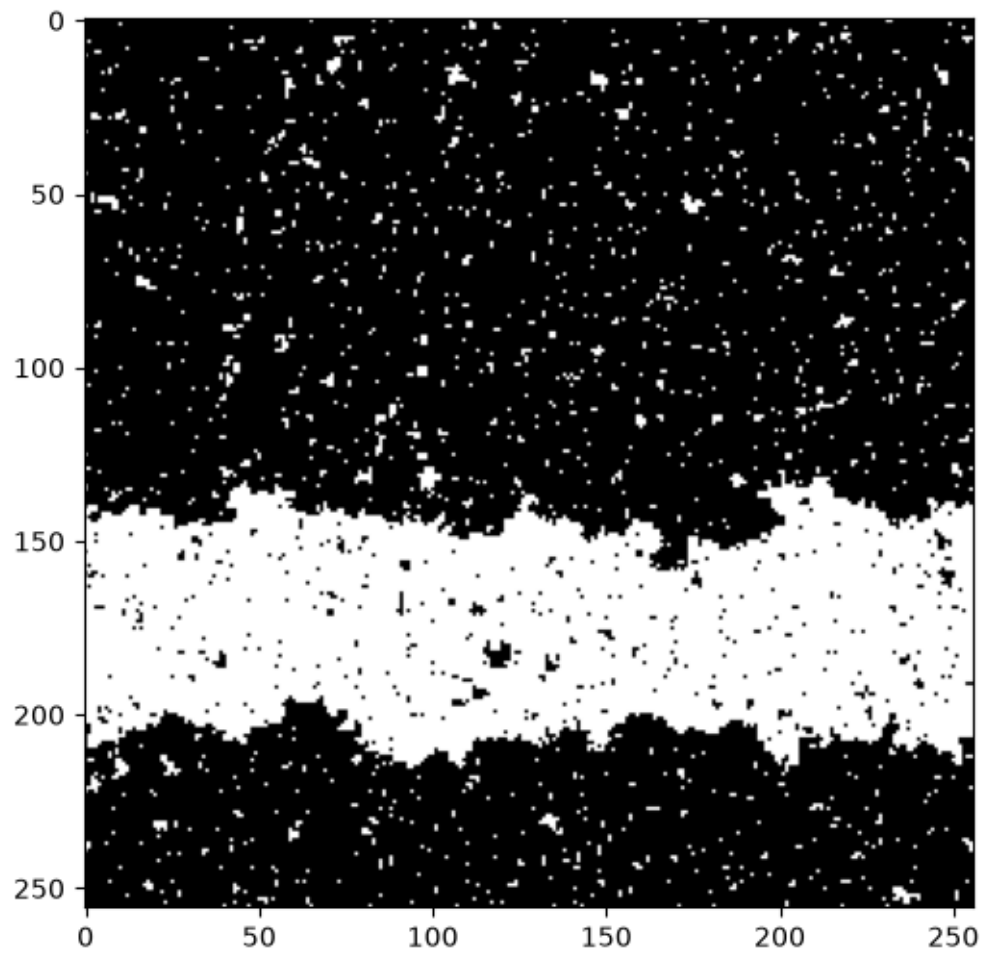
```
[21]: np.average(accept_list)
```

```
[21]: np.float64(0.09455360472202301)
```

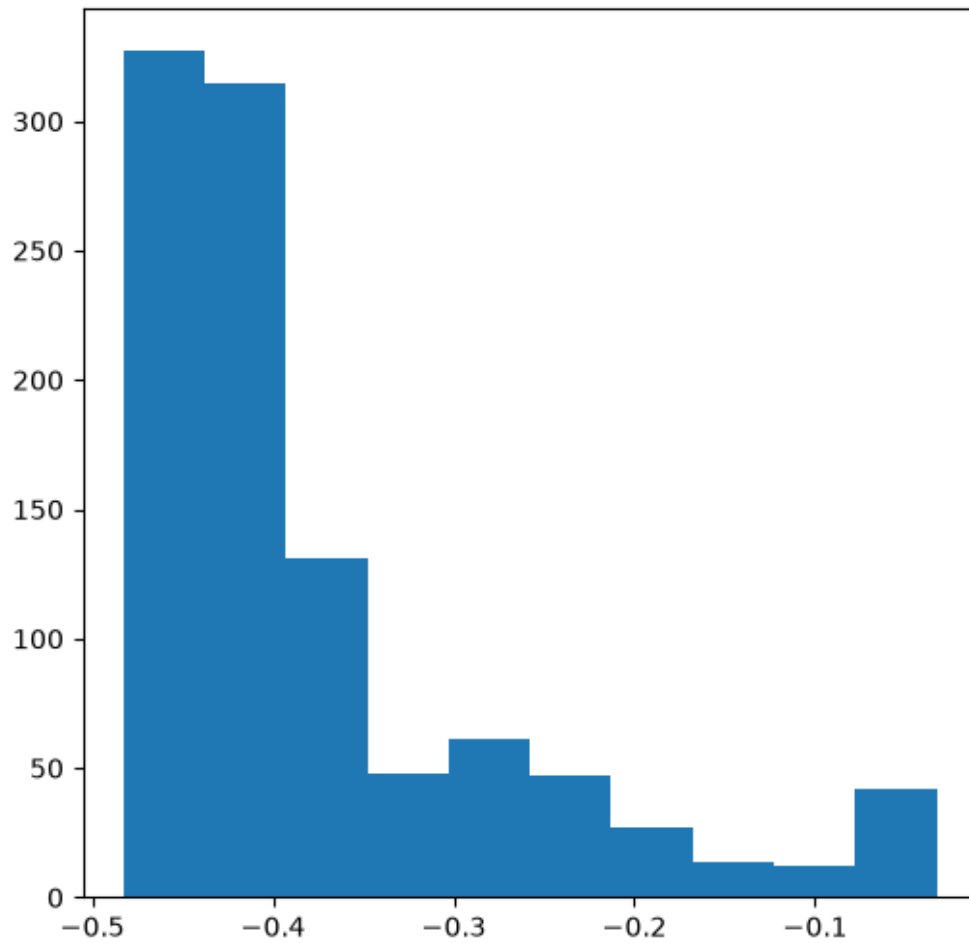
```
[22]: plt.plot(E_list);
```



```
[23]: plt.imshow(spin.reshape(Nx, Ny), cmap='grey');
```



```
[24]: plt.hist(m_list);
```



```
[25]: kT = 4.0
arr = arrhenius(kT)

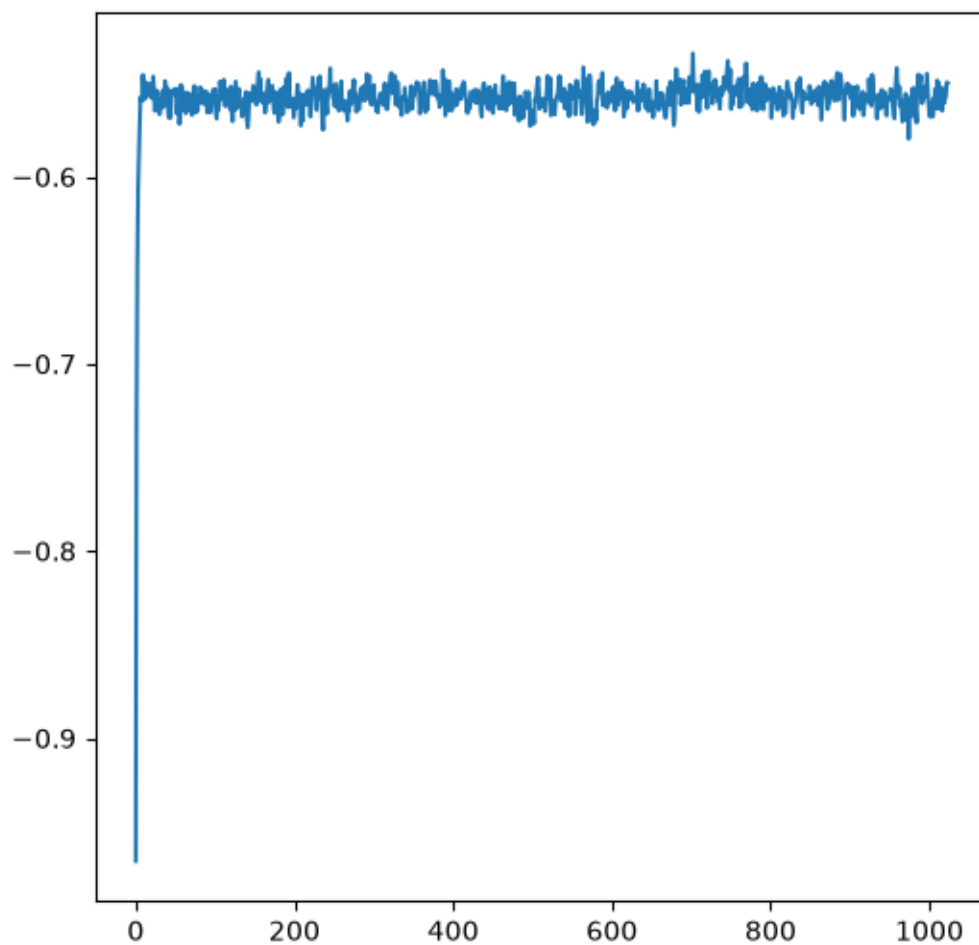
N_sweeps = 1024
E_list, accept_list, m_list = [], [], []
for _ in trange(N_sweeps):
    accept = MC(spin, kT, rng.uniform(size=N), neigh, arr)
    E_list.append(ene(spin, neigh)/N)
    accept_list.append(accept)
    m_list.append(np.average(spin))
E_list = np.array(E_list)
accept_list = np.array(accept_list)
m_list = np.array(m_list)

0%|          | 0/1024 [00:00<?, ?it/s]
```

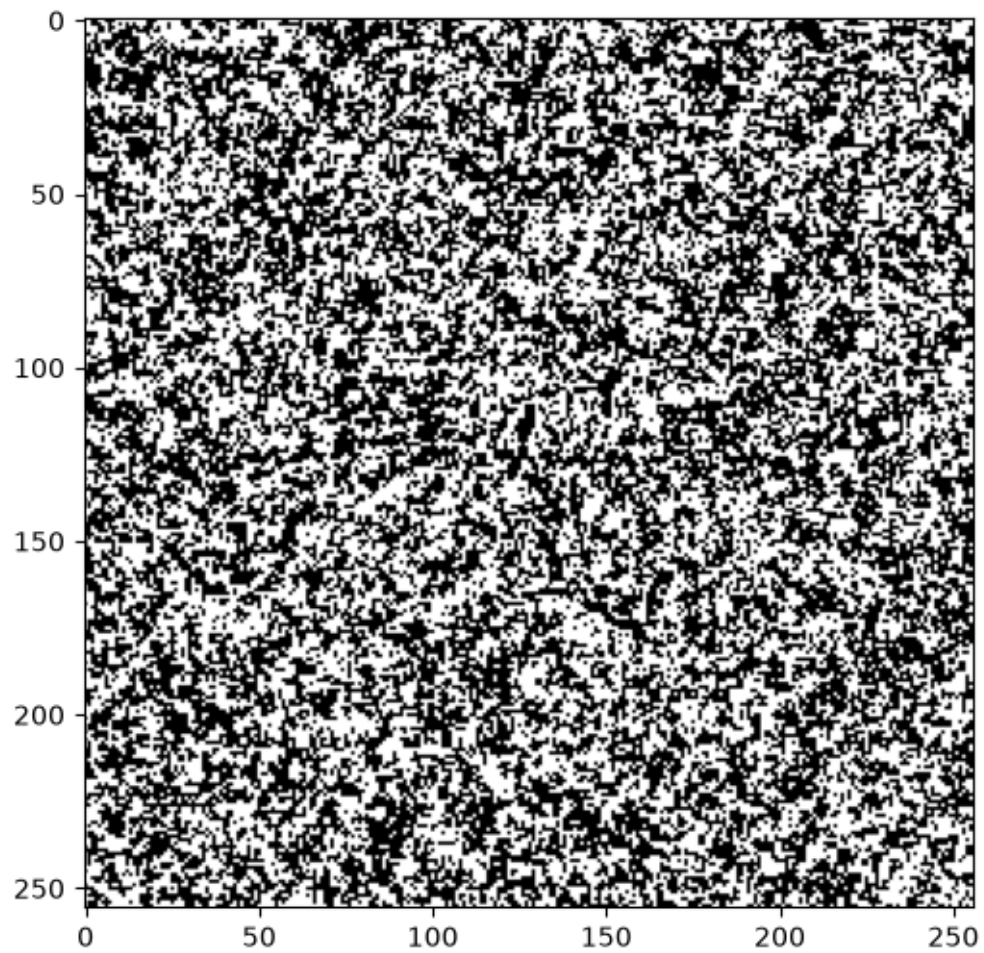
```
[26]: np.average(accept_list)
```

```
[26]: np.float64(0.6090470105409622)
```

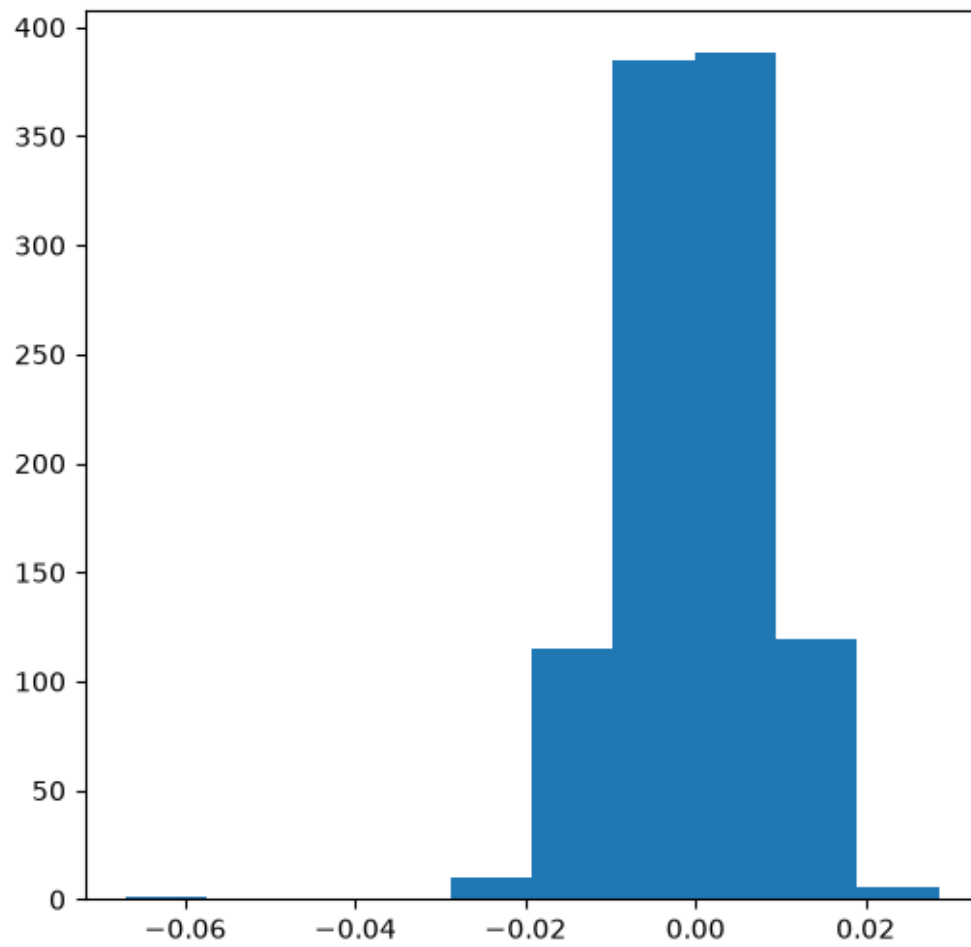
```
[27]: plt.plot(E_list);
```



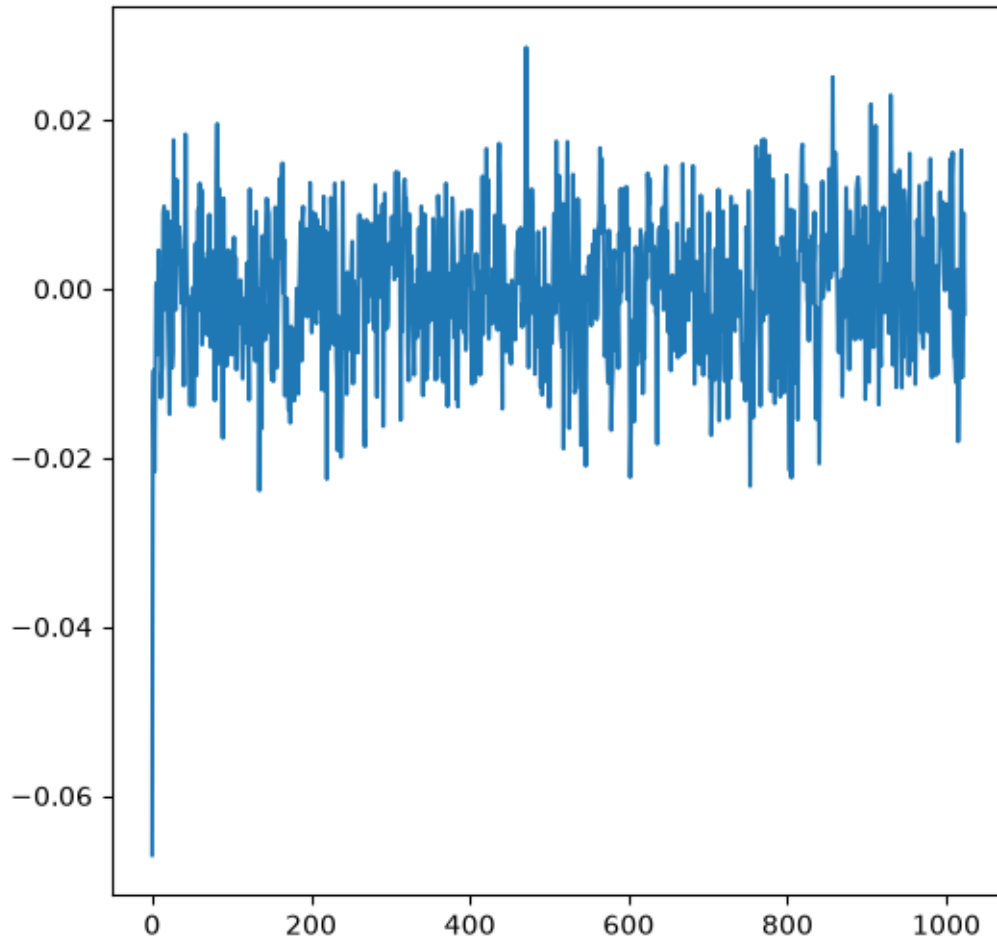
```
[28]: plt.imshow(spin.reshape(Nx, Ny), cmap='grey');
```



```
[29]: plt.hist(m_list);
```



```
[30]: plt.plot(m_list);
```



1.0.1 Magnetization with temperature

```
[31]: kT_list = np.linspace(3.0, 2.0, num=11)
N_sweeps = 2**14
m_T = []
for kT in kT_list:
    arr = arrhenius(kT)
    m_list = []
    for _ in trange(N_sweeps):
        # just capture magnetization
        MC(spin, kT, rng.uniform(size=N), neigh, arr)
        m_list.append(np.average(spin))
    m_T.append(np.array(m_list))
```

```
0%|          | 0/16384 [00:00<?, ?it/s]
```

```
0%|          | 0/16384 [00:00<?, ?it/s]
```

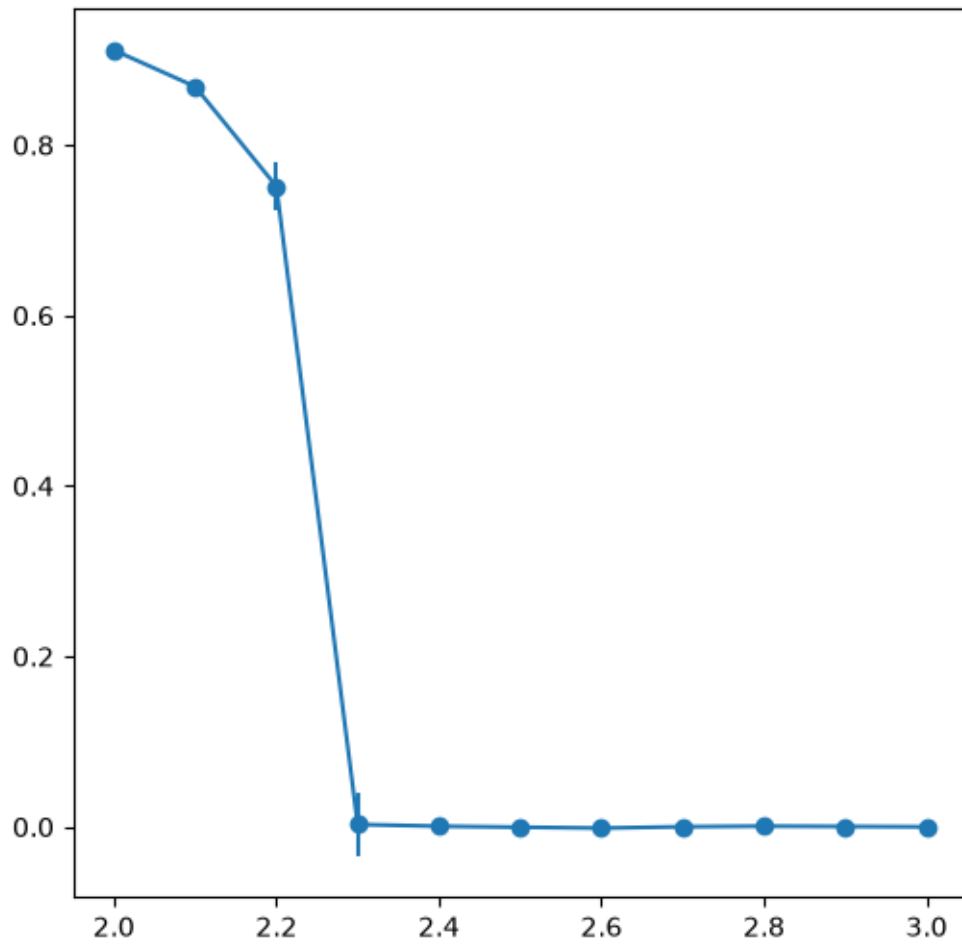
```
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
0%|          | 0/16384 [00:00<?, ?it/s]
```

```
[32]: ave_m_T = np.array([np.average(m) for m in m_T])
      err_m_T = np.array([np.std(m)/np.sqrt(N_sweeps) for m in m_T])
```

```
[33]: err_m_T
```

```
[33]: array([1.05363167e-04, 1.15340028e-04, 1.31240732e-04, 1.50952307e-04,
          1.84614594e-04, 2.39808414e-04, 3.66378048e-04, 1.27246047e-03,
          9.08543611e-04, 4.52012291e-05, 2.65709382e-05])
```

```
[34]: plt.errorbar(kT_list, ave_m_T, 30*err_m_T, fmt='o-');
```

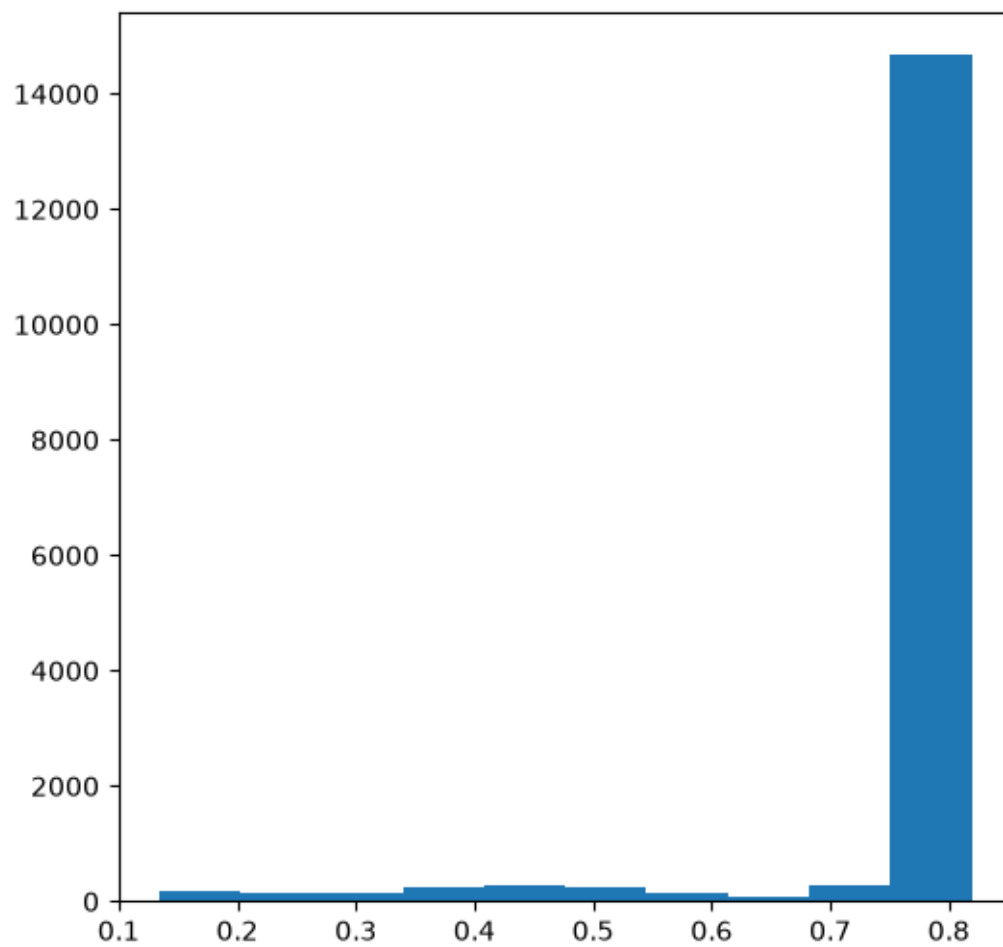


```
[35]: np.argwhere(kT_list==2.2)
```

```
[35]: array([[8]])
```

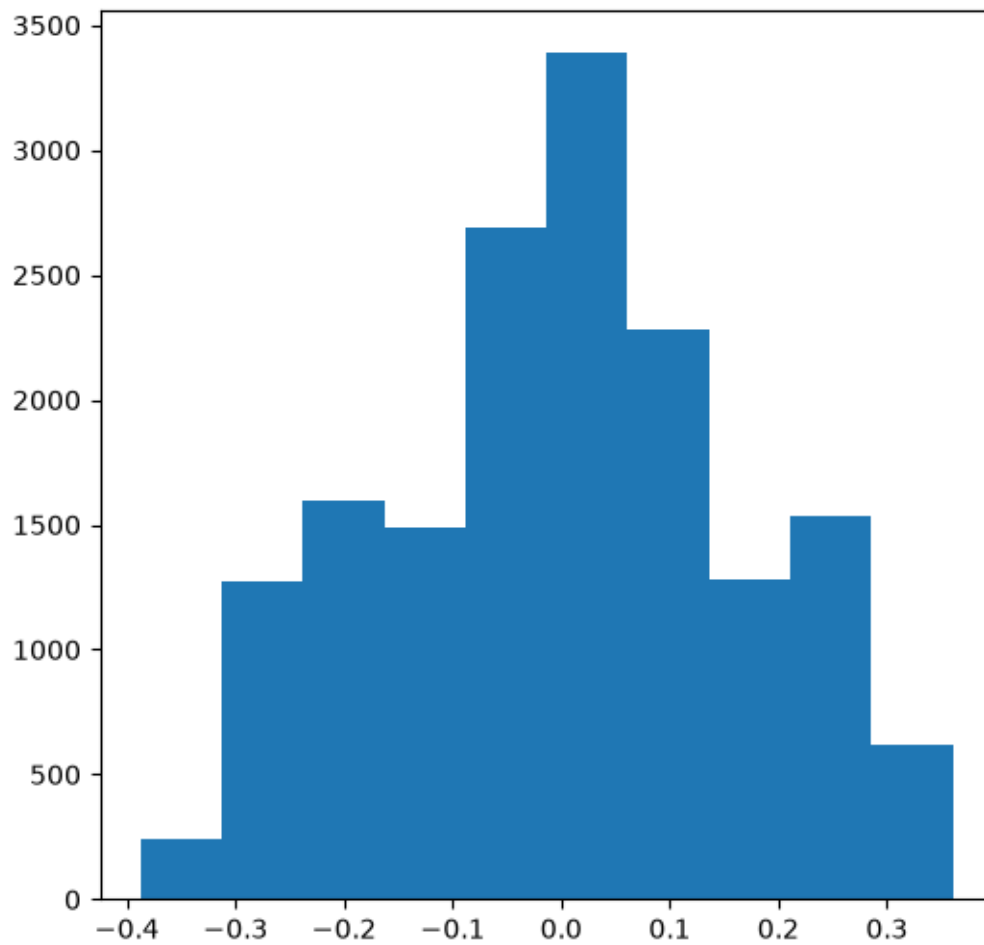
```
[36]: plt.hist(m_T[8]);  
      kT_list[8]
```

```
[36]: np.float64(2.2)
```



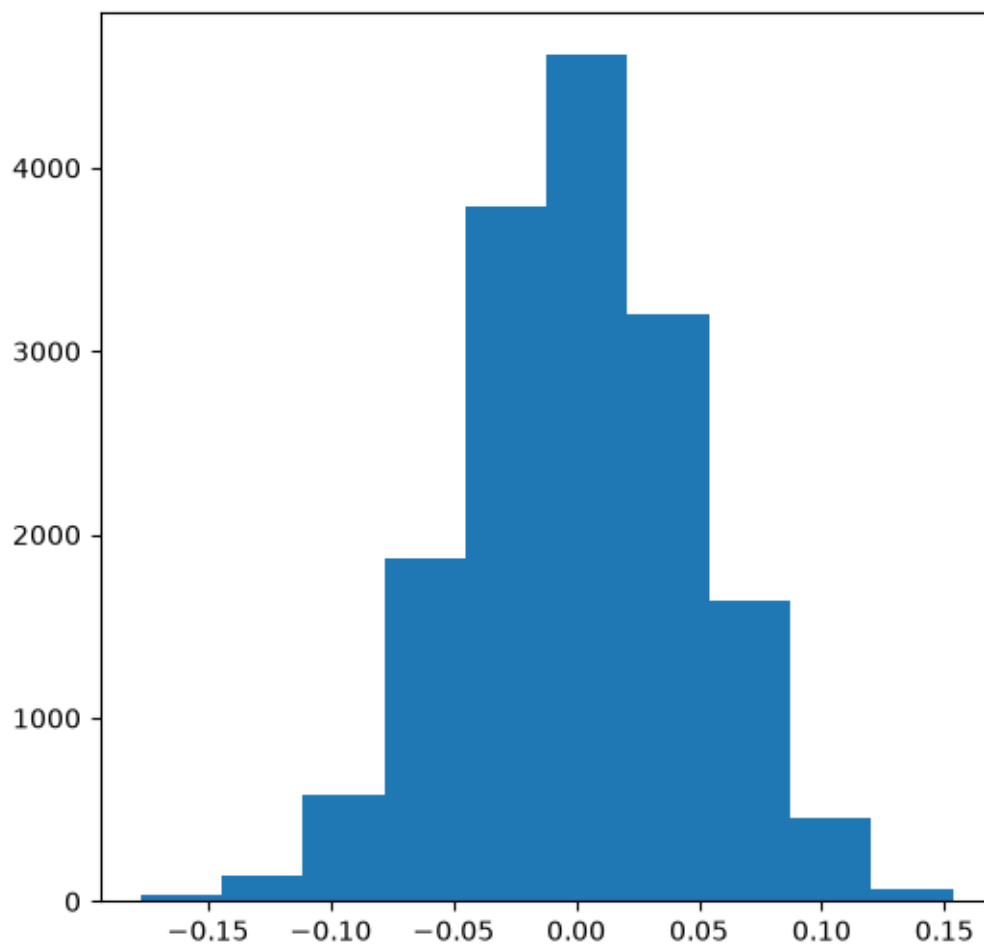
```
[37]: plt.hist(m_T[7]);  
      kT_list[7]
```

```
[37]: np.float64(2.3)
```



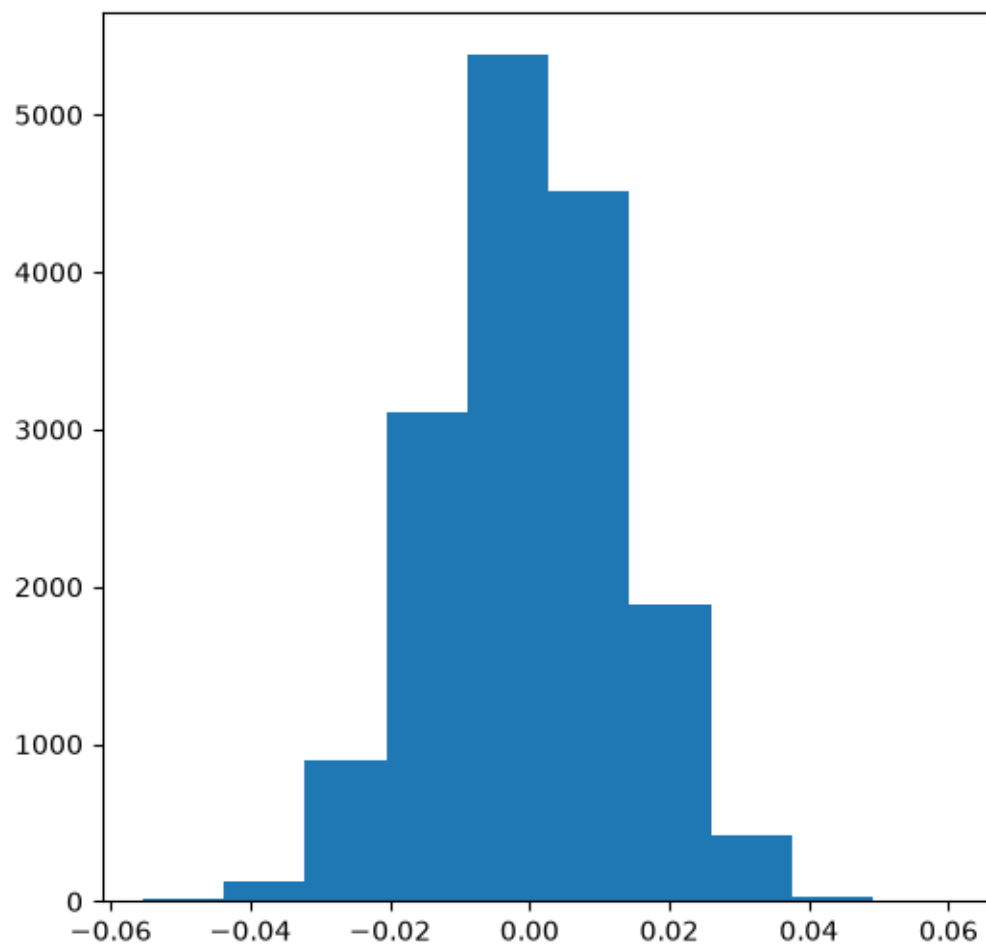
```
[38]: plt.hist(m_T[6]);  
      kT_list[6]
```

```
[38]: np.float64(2.4)
```



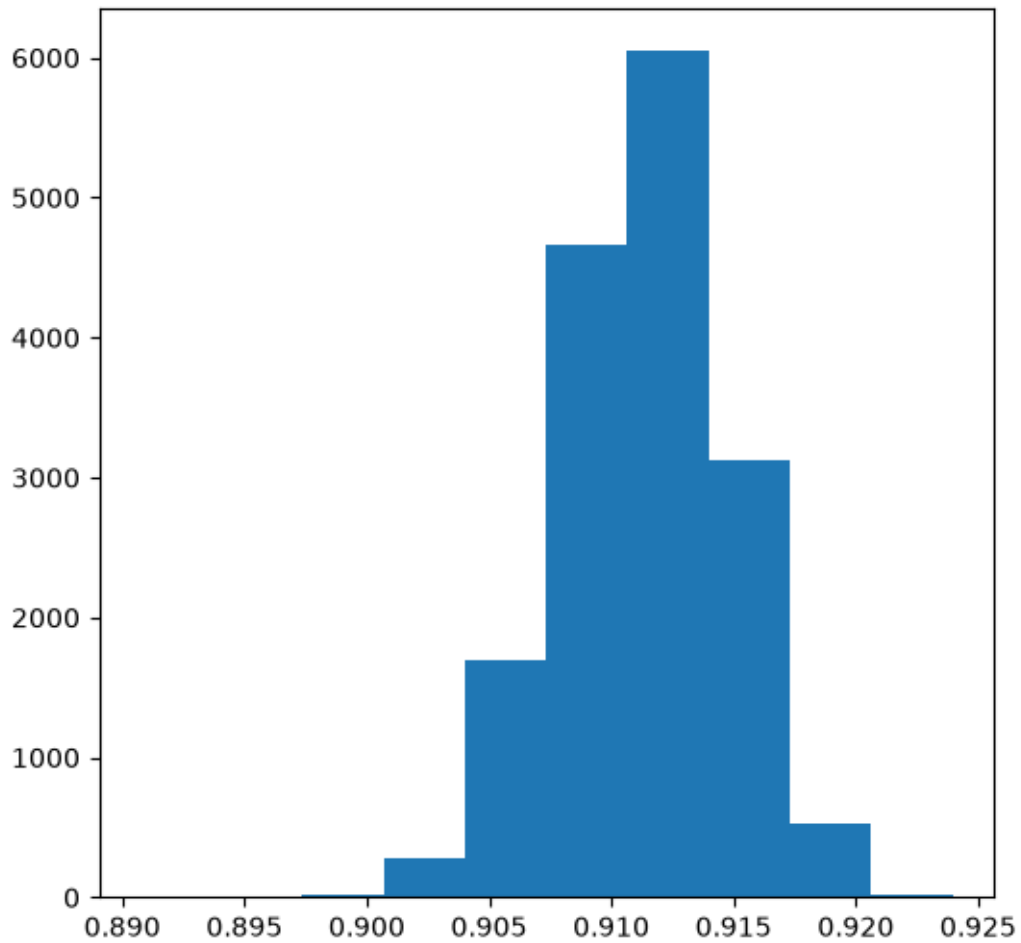
```
[39]: plt.hist(m_T[0]);  
      kT_list[0]
```

```
[39]: np.float64(3.0)
```



```
[40]: plt.hist(m_T[-1]);  
      kT_list[-1]
```

```
[40]: np.float64(2.0)
```



1.0.2 Long run near T_c

```
[41]: # kT = 2.0/(np.log(np.sqrt(2)+1))*param['J']
      kT = 2.3
      arr = arrhenius(kT)

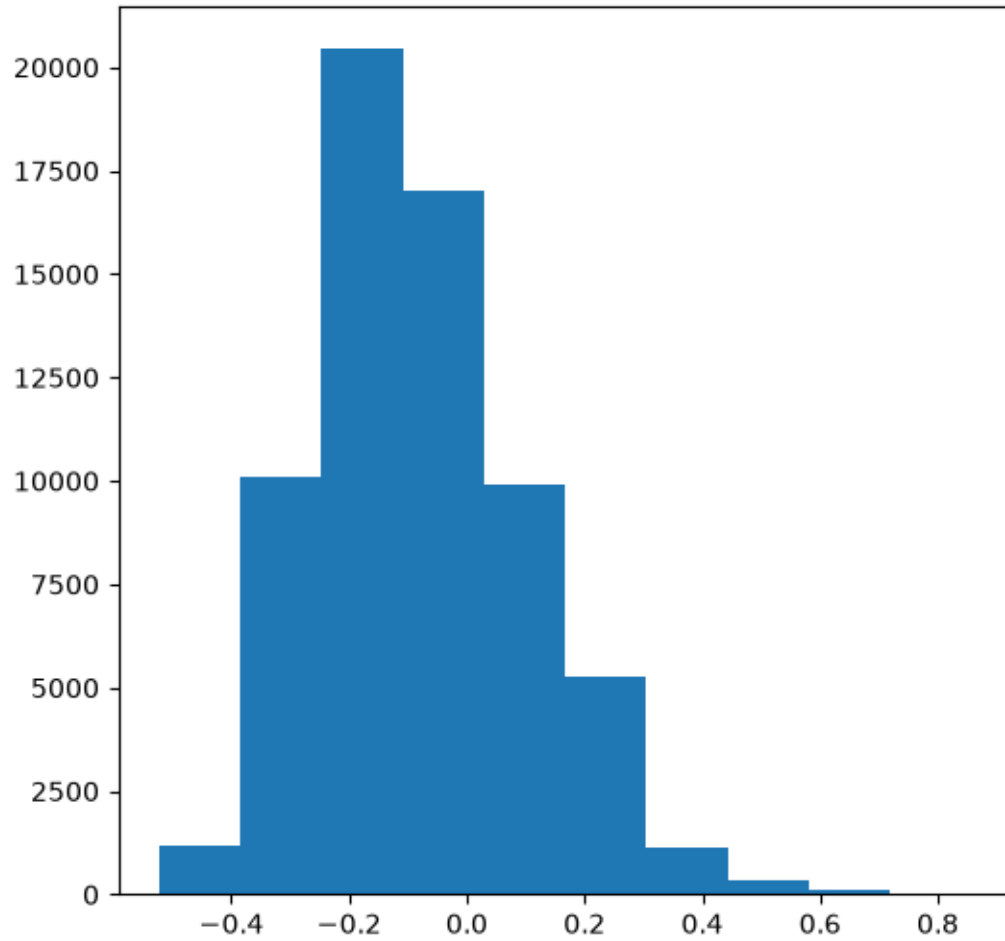
      N_sweeps = 2**16
      E_list, accept_list, m_list = [], [], []
      for _ in trange(N_sweeps):
          accept = MC(spin, kT, rng.uniform(size=N), neigh, arr)
          E_list.append(ene(spin, neigh)/N)
          accept_list.append(accept)
          m_list.append(np.average(spin))
      E_list = np.array(E_list)
      accept_list = np.array(accept_list)
      m_list = np.array(m_list)
```

0% | 0/65536 [00:00<?, ?it/s]

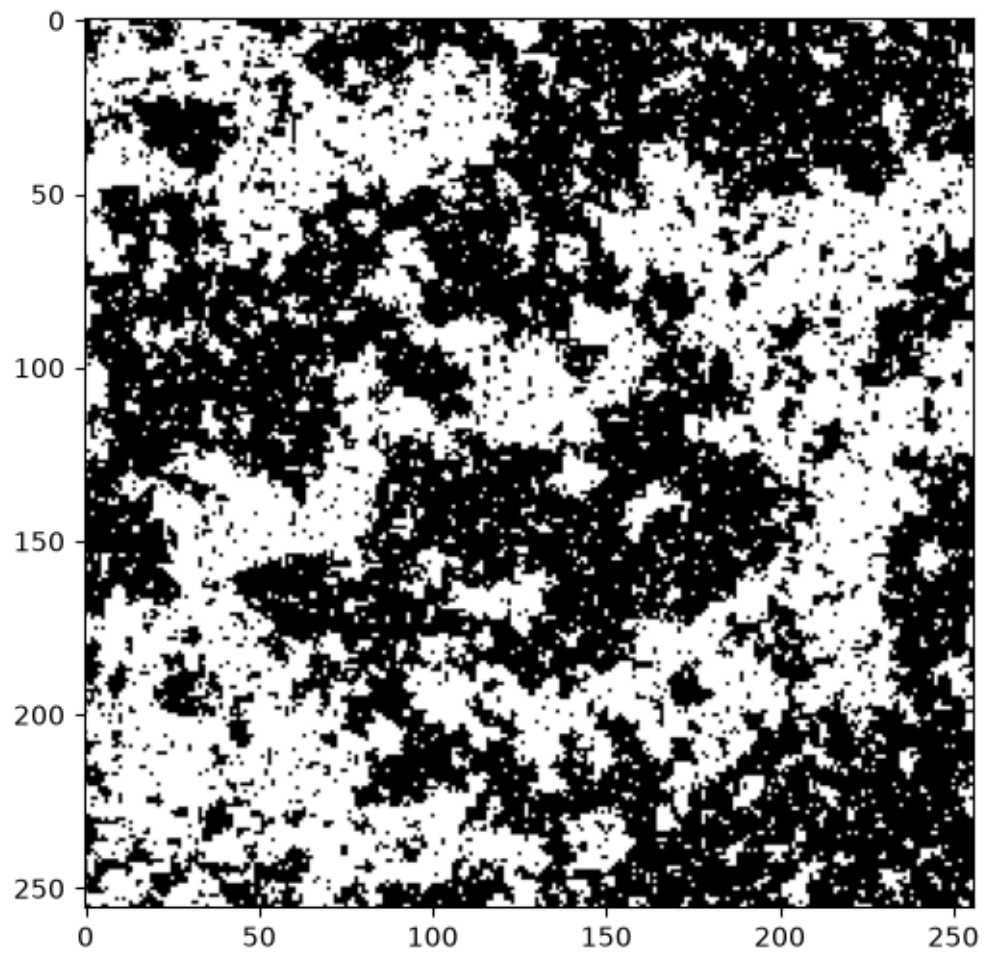
```
[42]: np.average(accept_list)
```

```
[42]: np.float64(0.22617468936368823)
```

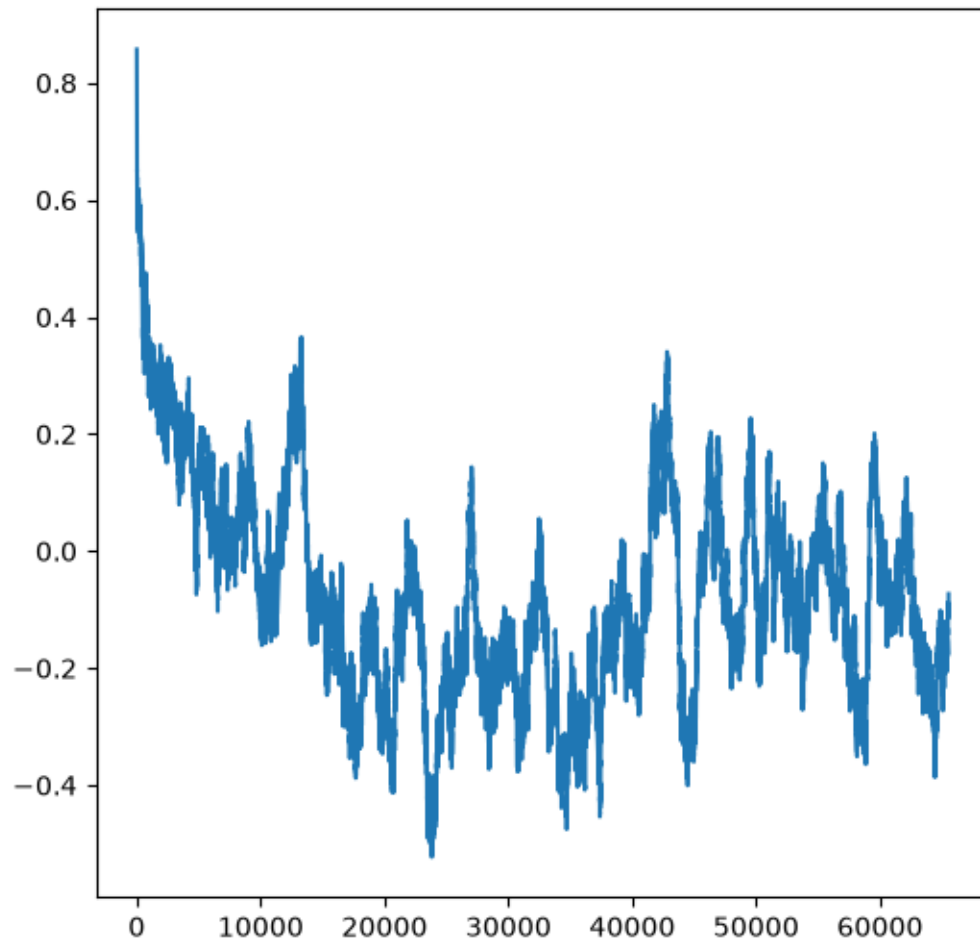
```
[43]: plt.hist(m_list);
```



```
[44]: plt.imshow(spin.reshape(Nx, Ny), cmap='grey');
```

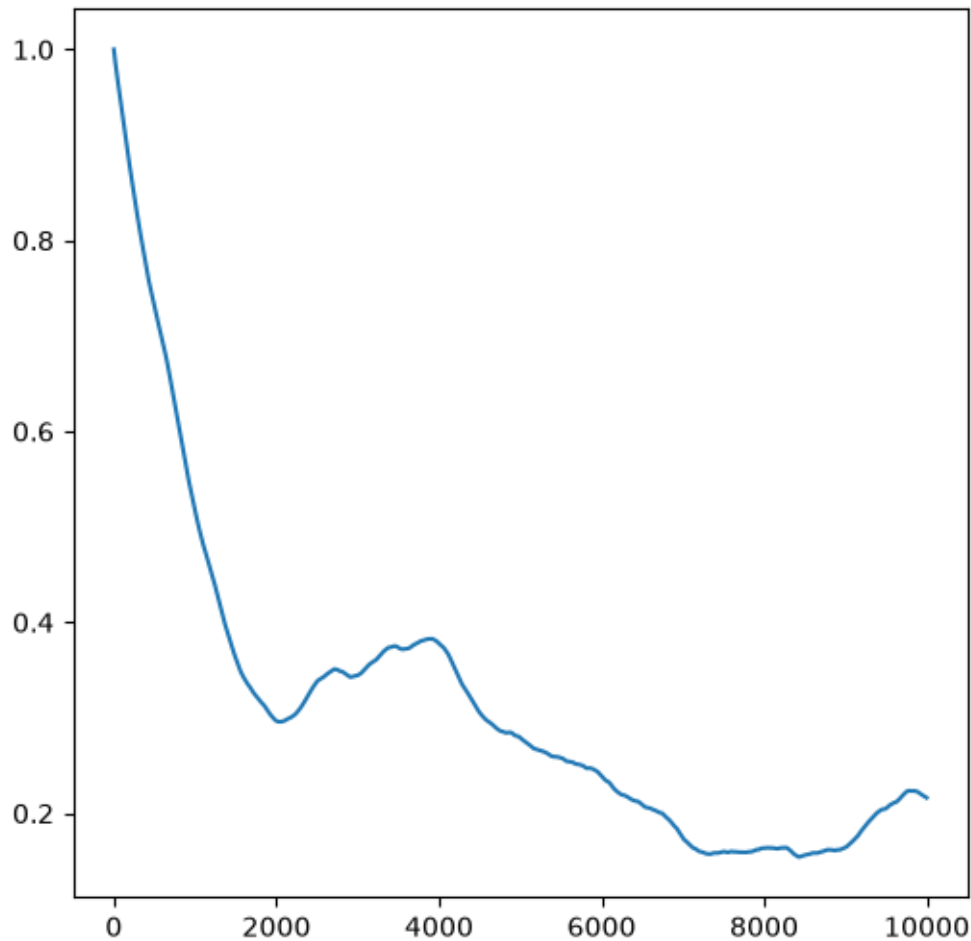


```
[45]: plt.plot(m_list);
```



```
[46]: def acf_calc(v):
        """Returns the auto-correlation function for the sequence v"""
        v_ave = np.mean(v)
        v_len = len(v)
        return np.correlate(v-v_ave, v-v_ave, "full")[(v_len-1):] / \
            (np.var(v)*np.array([v_len - x for x in range(v_len)]))
```

```
[47]: plt.plot(acf_calc(m_list)[:10000]);
```



1.0.3 Cluster algorithm

```
[48]: from scipy import sparse # graph analysis for clusters
```

```
[49]: @jit(nopython=True)
def spin_make_graph(row, col, spin, neigh_sorted, connect):
    """Makes a sparse matrix of connected bonds. `spin` is the state,
    ↳ `neigh_sorted` is our
    ↳ *sorted* neighbor list, `row` and `col` are the outputs, and `connect` is a
    ↳ random matrix
    ↳ of booleans deciding if we connect a given bond. Returns the number of
    ↳ connections."""
    Nconnect = 0
    counter = 0
    for i, s, nn in zip(range(len(spin)), spin, neigh_sorted):
        for j in nn:
            if i < j and s == spin[j] and connect[counter]:
```

```

        row[Nconnect] = i
        col[Nconnect] = j
        Nconnect += 1
        counter += 1
    return Nconnect

neigh_sorted = neigh.copy()
for i in range(N):
    neigh_sorted[i] = sorted(neigh[i])

row = np.zeros(N*z, dtype=int)
col = np.zeros(N*z, dtype=int)
data = np.ones(N*z, dtype=int)

```

```

[50]: # cluster algorithm...
NclusterMC = 1024
E_list, m_list = [], []

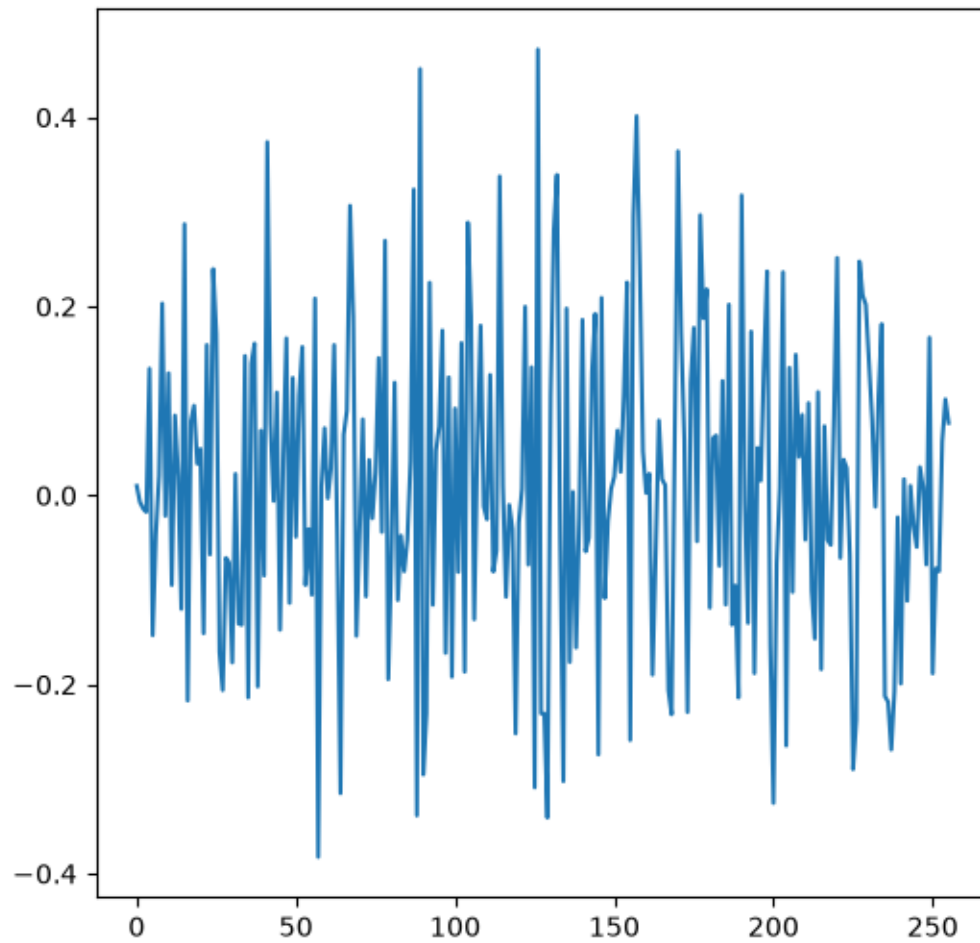
for _ in range(NclusterMC):
    # make our connections:
    Nconnect = spin__make_graph(row, col, spin, neigh_sorted, rng.
    ↪uniform(size=(N*z)) < (1-np.exp(-2*param['J']/kT)))
    # find the clusters:
    connectivity = sparse.csr_matrix((data[:Nconnect], (row[:Nconnect], col[:
    ↪Nconnect])), shape=(N,N))
    nsub, labels = sparse.csgraph.connected_components(connectivity,
    ↪directed=False)
    # label[i] = which cluster site i belongs to; we need to find the clusters
    ↪of size > 1 to flip
    label_count = np.zeros(nsub, dtype=int)
    for i in labels:
        label_count[i] += 1
    # now we make a list of which sites we're going to flip; we can either
    ↪*always* flip
    # or *randomly* choose to flip
    # label_flip = set(label for label, count in enumerate(label_count) if
    ↪count > 1) # always flip
    label_flip = set(label for label, count, flip in
                      zip(range(len(label_count)), label_count, rng.
    ↪choice((True, False), size=nsub))
                      if count > 1 and flip)
    for i, label in enumerate(labels):
        if label in label_flip:
            spin[i] = -spin[i]
    m_list.append(np.average(spin))
    E_list.append(ene(spin, neigh_sorted))

```

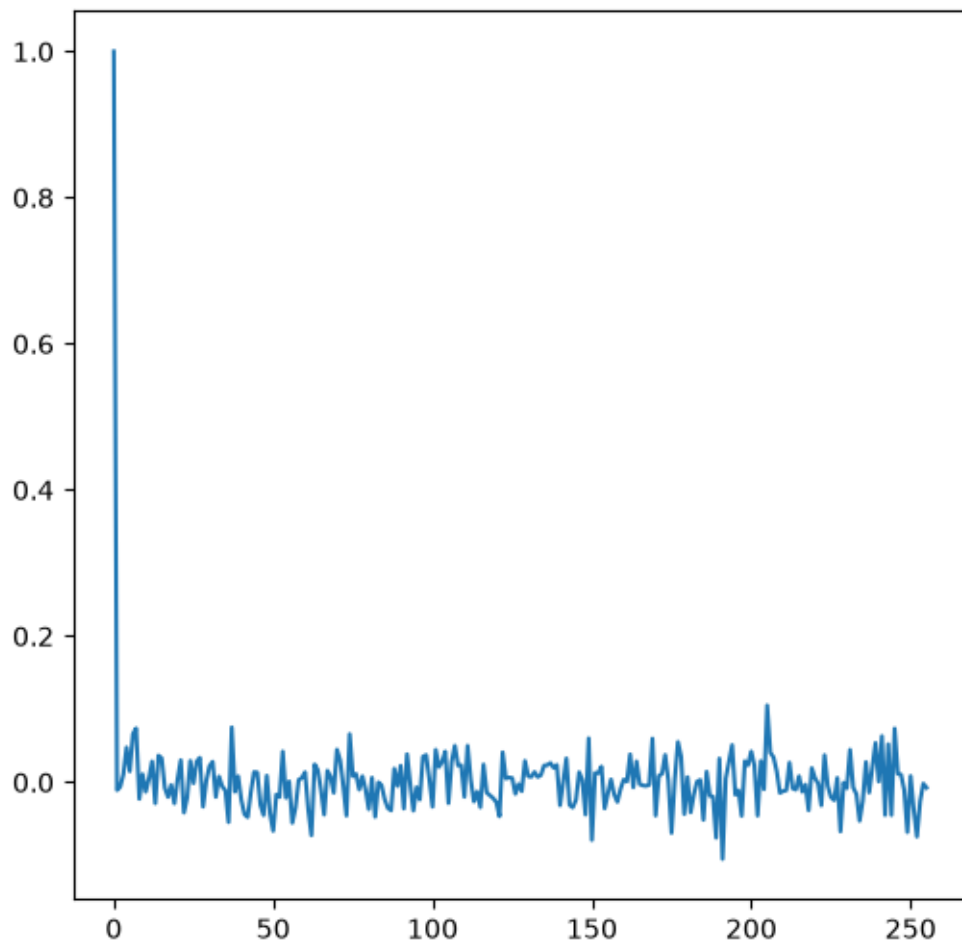
```
m_list = np.array(m_list)
E_list = np.array(E_list)
```

```
0%|          | 0/1024 [00:00<?, ?it/s]
```

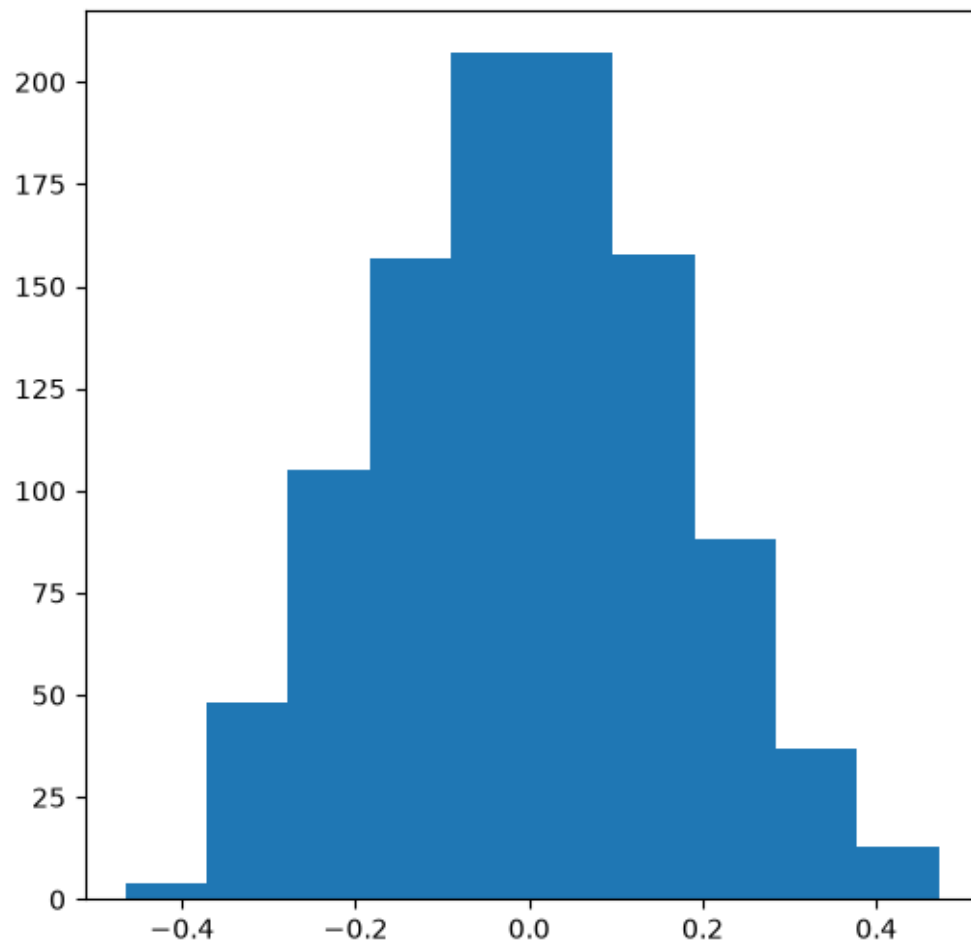
```
[51]: plt.plot(m_list[:256]);
```



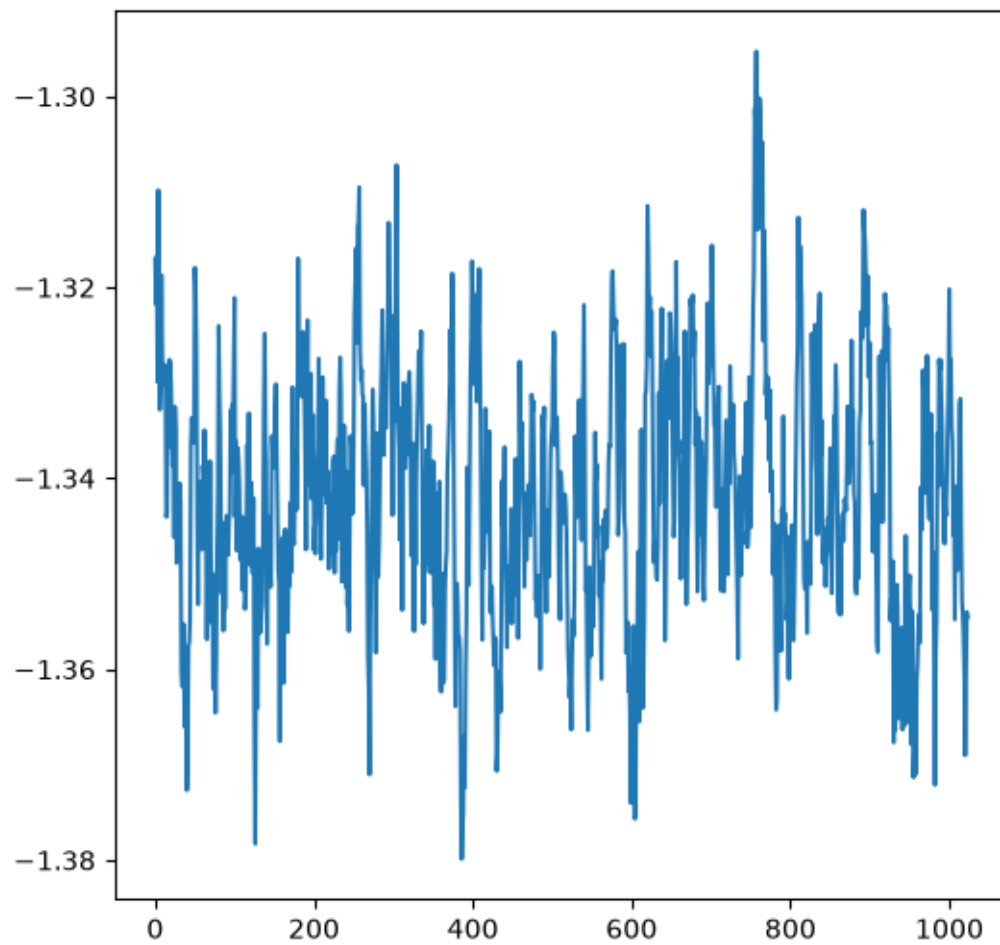
```
[52]: plt.plot(acf_calc(m_list)[:256]);
```



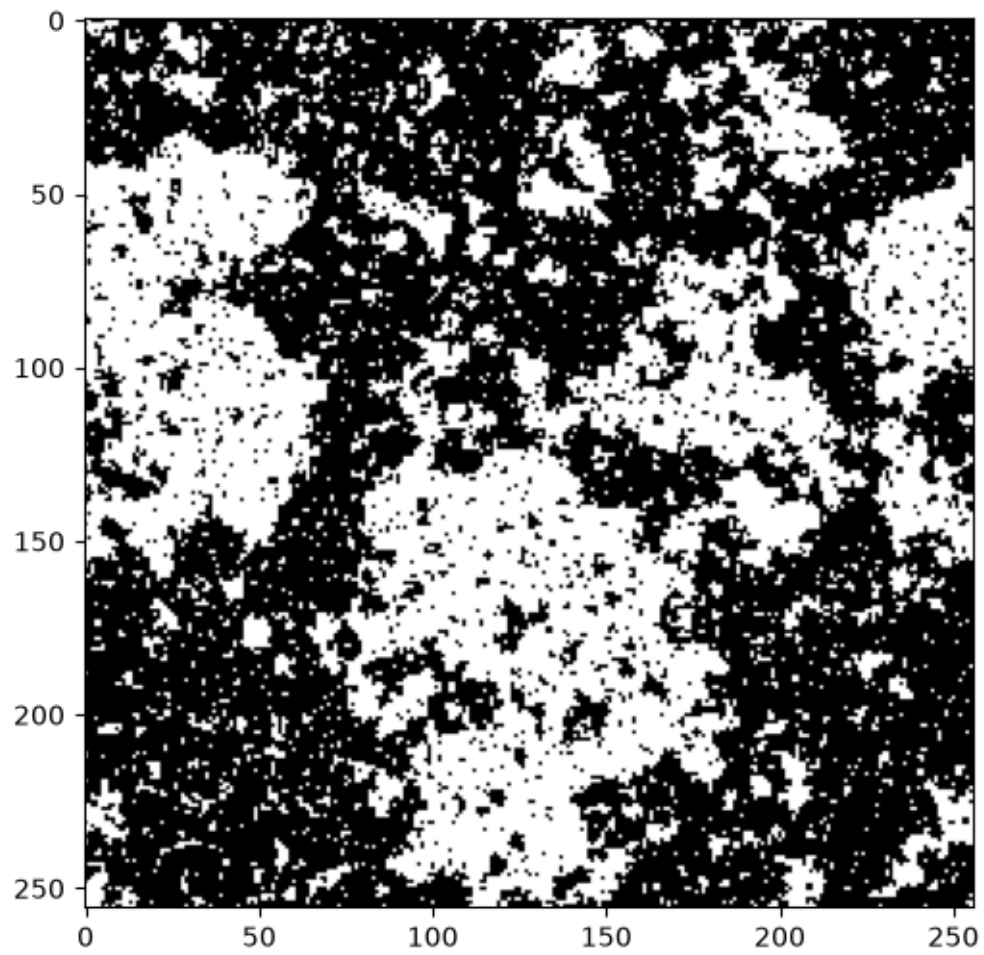
```
[53]: plt.hist(m_list);
```



```
[54]: plt.plot(E_list/N);
```



```
[55]: plt.imshow(spin.reshape(Nx, Ny), cmap='grey');
```



[]: