AM783 | Applied Markov Processes | CW2 solutions to numerical questions

Hugo Touchette

Last updated: 26 August 2022

Python 3

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
```

In [2]:
```
# Magic command for vectorised figures
%config InlineBackend.figure_format = 'svg'
```

Q6

For $P(0 \to 1) = a$ and $P(1 \to 0) = b$, the transition matrix is

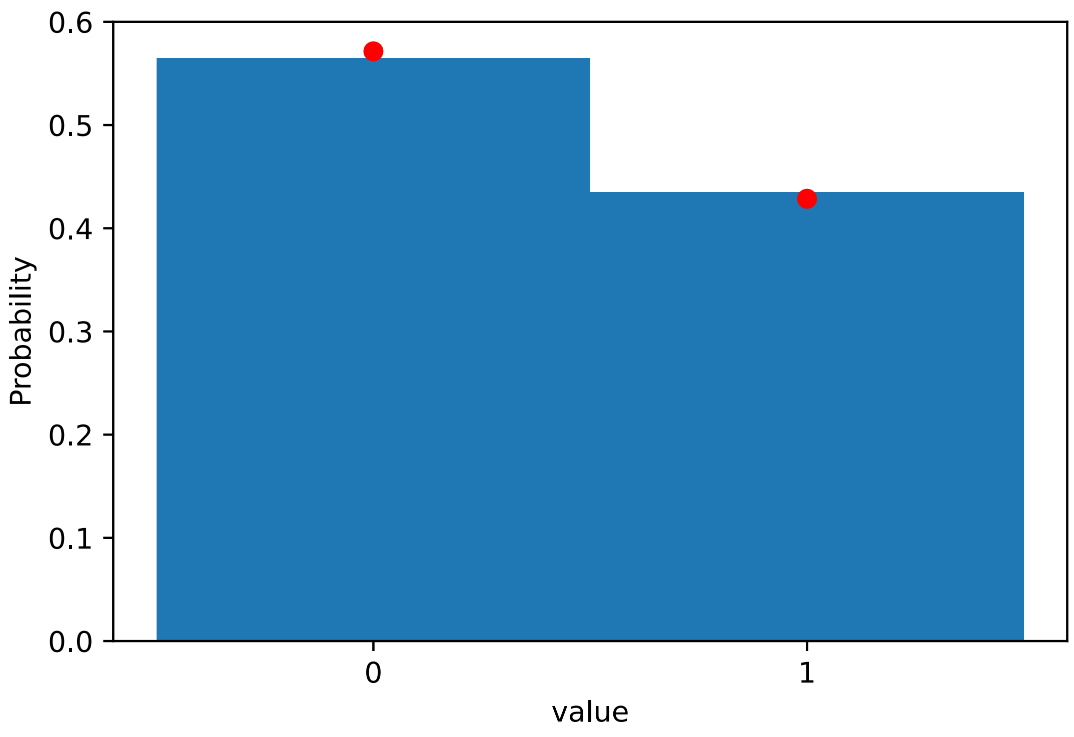
$$\Pi = \begin{pmatrix} 1-a & a \\ b & 1-b \end{pmatrix},$$

which yields the following stationary distribution:

$$p^* = \left(\frac{b}{a+b}, \frac{a}{a+b}\right).$$

The following code checks this result by simulating independent Markov chains (in series, i.e., one after the other) and by keeping the final state to form a sample. Note that we don't need to store the whole trajectory in time; only the current state must be kept to generate the next state.

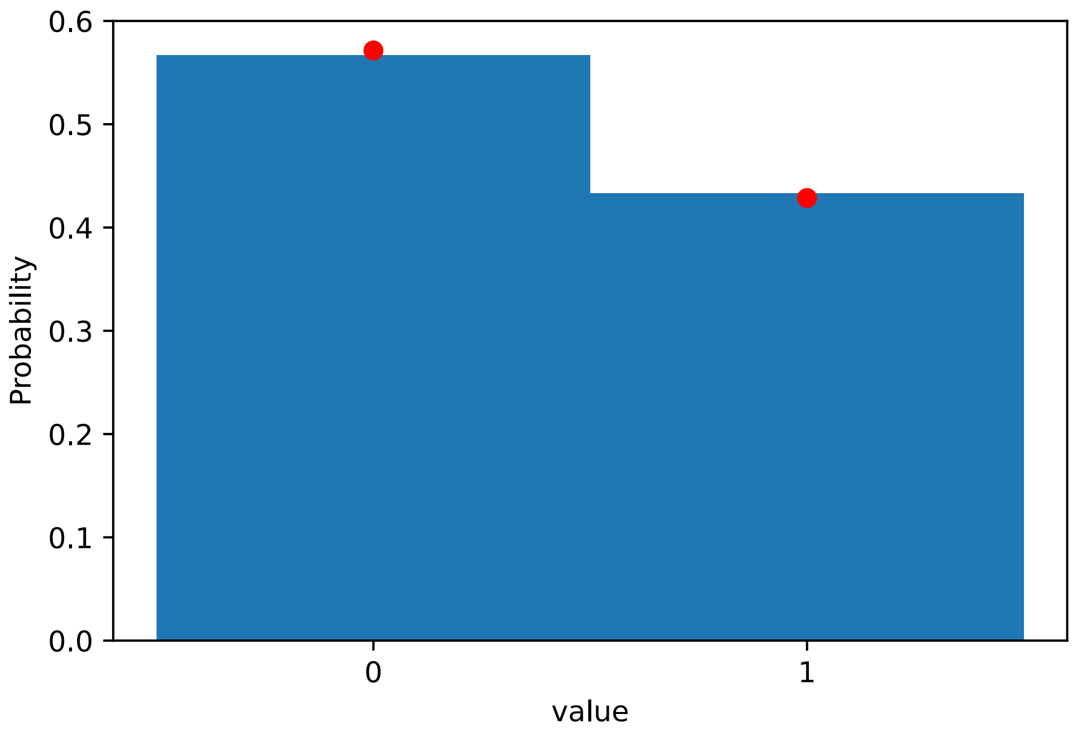
In [21]:
```python
nb_steps = 10**3
final_time = 100
a = 0.3
b = 0.4
last_state_sample = np.zeros(nb_steps, int)

# Simulation
for j in range(nb_steps):
    MC_state = 0
    for i in range(final_time-1):
        if MC_state == 0:
            r = np.random.random()
            if r < a:
                MC_state = 1  # Flip state 0 -> 1 with probability a
        else:
            r = np.random.random()
            if r < b:
                MC_state = 0  # Flip state 1 -> 0 with probability b

    # Put last state in sample
    last_state_sample[j] = MC_state

# Histogram
hist, bin_spec = np.histogram(last_state_sample, bins=[0, 1, 2], density=
plt.bar(bin_spec[:-1], hist, width = 1)
plt.plot([0, 1], [b/(a+b), a/(a+b)], 'ro')
plt.xticks([0, 1])
plt.xlabel('value')
plt.ylabel('Probability');
```



The blue bars are the simulation results; the red dots the theoretical results. The agreement is good.

Next, we accumulate the sample in time rather than over repeated simulations of the Markov chain, so we have one loop intead of two.

In [4]:
```python
final_time = 10**3
a = 0.3
b = 0.4
state_sample = np.zeros(final_time, int)

# Simulation
MC_state = 0
for i in range(final_time):
    if MC_state == 0:
        r = np.random.random()
        if r < a:
            MC_state = 1   # Flip state 0 -> 1 with probability a
    else:
        r = np.random.random()
        if r < b:
            MC_state = 0   # Flip state 1 -> 0 with probability b

    # Put current state in sample
    state_sample[i] = MC_state

# Histogram
hist, bin_spec = np.histogram(state_sample, bins=[0, 1, 2], density=True)
plt.bar(bin_spec[:-1], hist, width = 1)
plt.plot([0, 1], [b/(a+b), a/(a+b)], 'ro')
plt.xticks([0, 1])
plt.xlabel('value')
plt.ylabel('Probability');
```



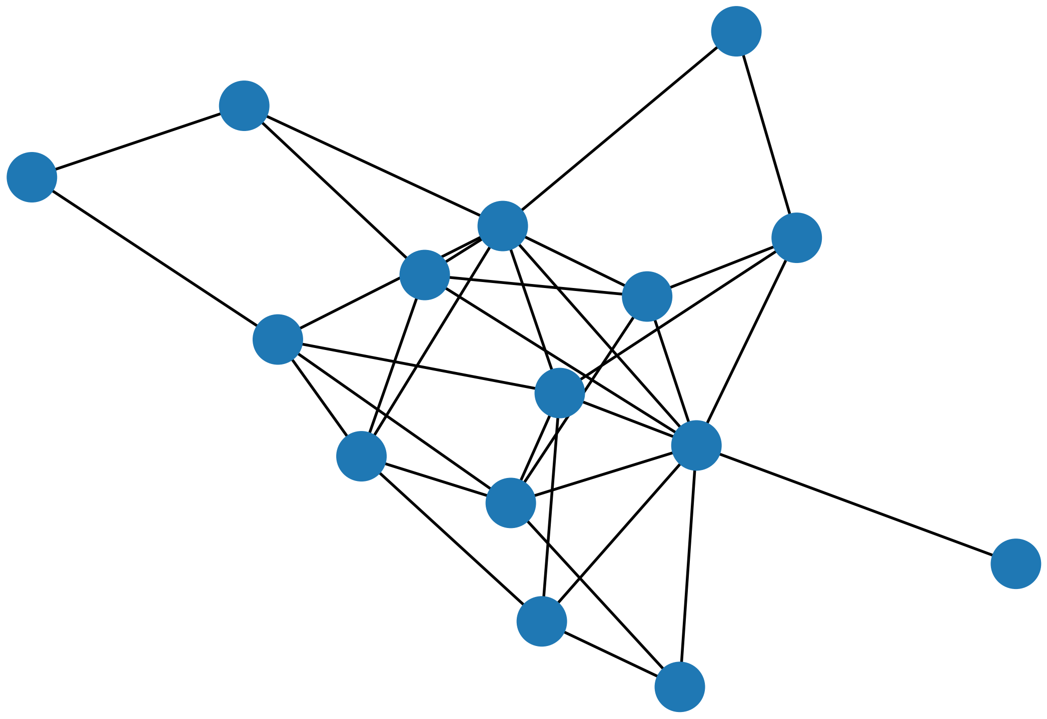
The results are also good.

It should be clear that the latter method, based on the ergodic theorem, is more efficient, as we're not wasting any states (iterates) of the Markov chain.

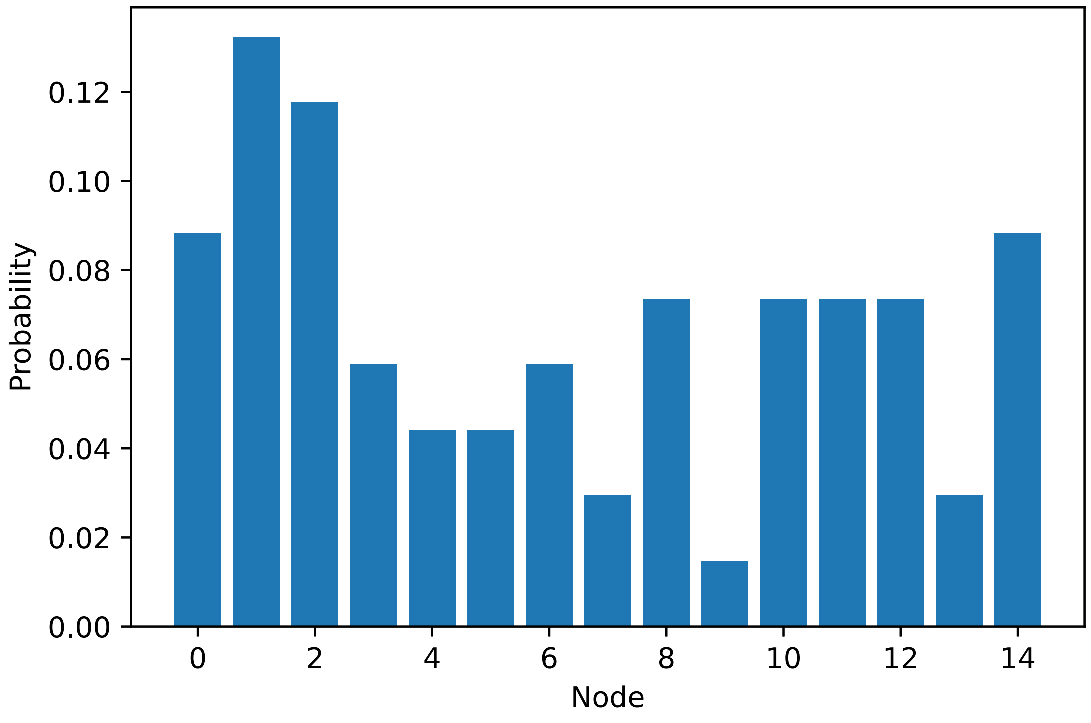
Q7

Python has a nice graph library, called NetworkX (imported above), that we can use to generate and manipulate graphs. The following solution uses that library, but all steps could be also done easily with loops.

In [6]:
```
# Generate a random graph (here a binomial or Erdos-Renyi graph) and show
nb_nodes = 15
G = nx.binomial_graph(nb_nodes, np.random.random())
nx.draw(G)
```



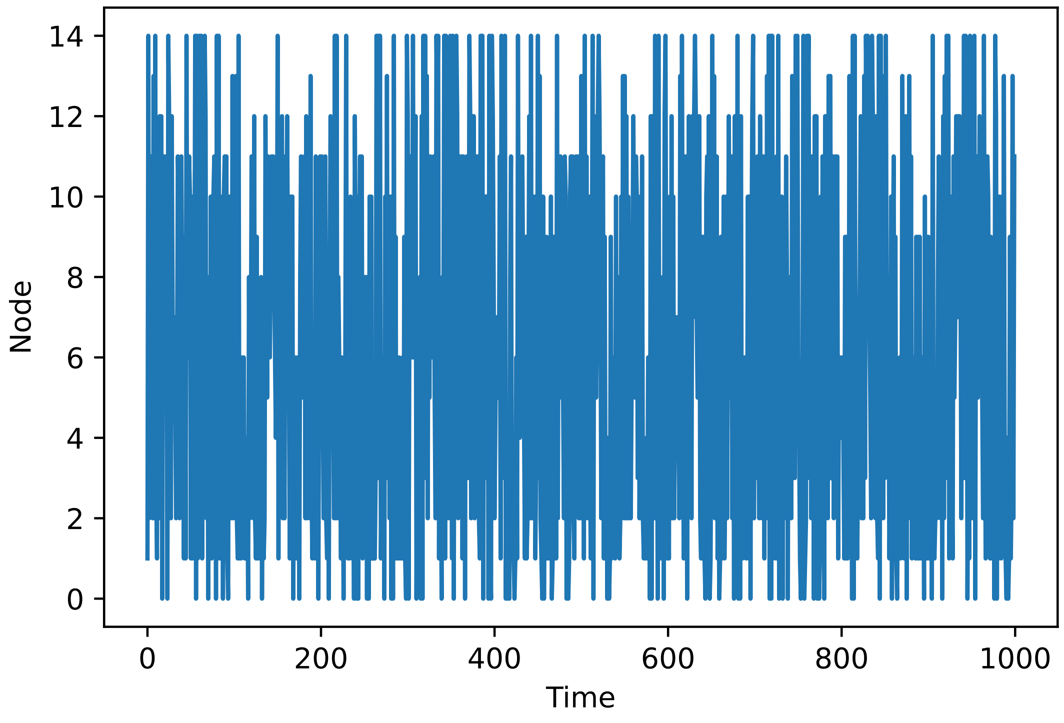
In [7]:
```
# List of degrees to compute stationary distribution
degree_list = np.array([val for (node, val) in G.degree()])
p_star = degree_list/sum(degree_list)
plt.bar(range(nb_nodes), p_star)
plt.xlabel('Node')
plt.ylabel('Probability');
```



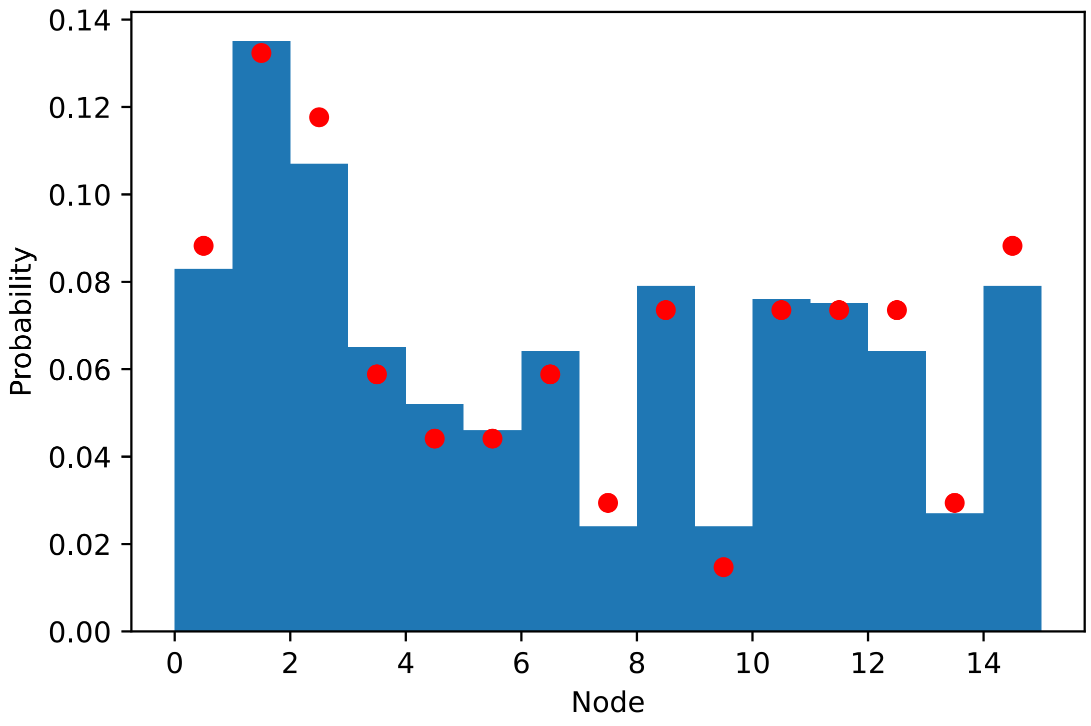
This bar chart shows the stationary distribution each of the 15 nodes, numbered (in Python) from 0 to 14.

Next is the simulation of the random walk.

In [9]:
```
# Get adjacency matrix (sparse matrix object to be transformed to normal
A_sparse = nx.adjacency_matrix(G)
A = A_sparse.toarray()

# Simulation of the random walk on the graph
nb_steps = 10**3
state_list = np.zeros(nb_steps, int)

node = 0
for i in range(nb_steps):
    connected_nodes = np.nonzero(A[node, :])[0]  # Find nodes connected t
    node = np.random.choice(connected_nodes)    # Choose one of these node
    state_list[i] = node                        # Save visited node

# Trajectory
plt.plot(range(nb_steps), state_list)
plt.xlabel('Time')
plt.ylabel('Node');
```



In [10]:
```
# Empirical distribution from visited states
plt.hist(state_list, bins=range(nb_nodes+1), density=True)
plt.plot(np.arange(nb_nodes)+0.5, p_star, 'ro')
plt.xlabel('Node')
plt.ylabel('Probability');
```



The bars show the simulation results, compared with the theoretical distribution in red. The results are qualitatively good; better agreement is obtained for longer simulation times, although the convergence is slow. This is known mathematically: the ergodic theorem is slow to converge on random Erdos-Renyi graphs.

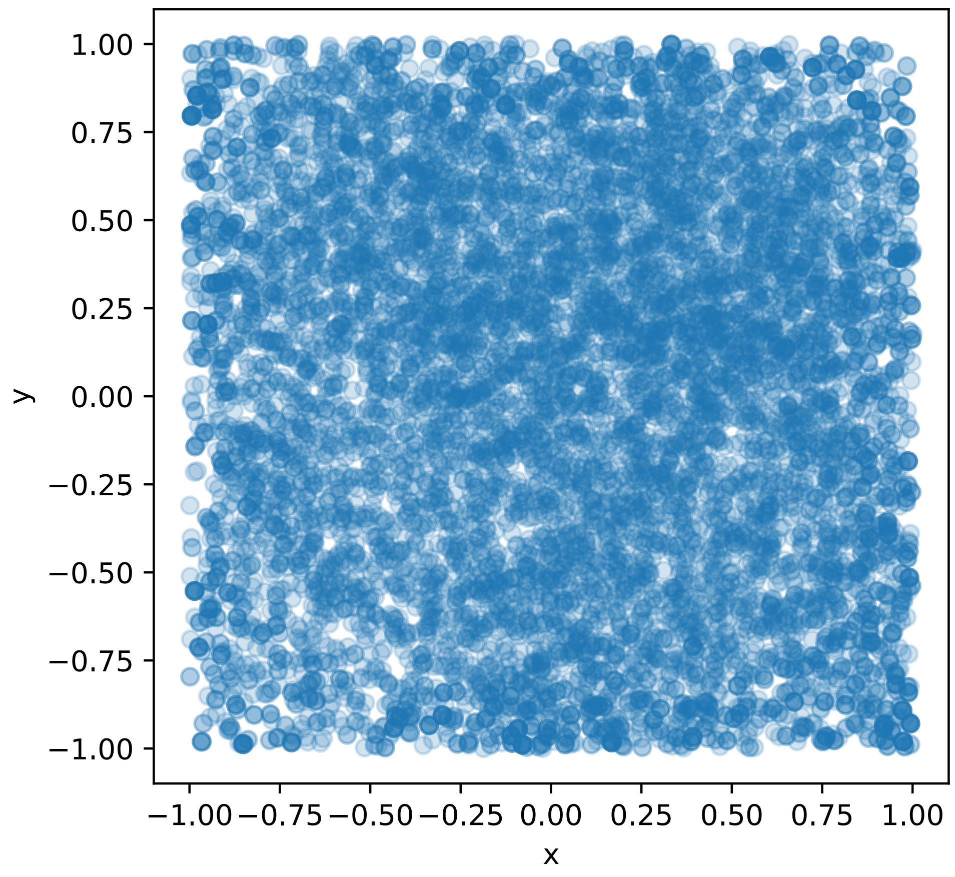
Q8

We'll use a small Gaussian displacement with zero mean throughout, meaning $\delta P_x \sim \delta P_y \sim \mathcal{N}(0, \sigma^2)$, with small σ.

a)

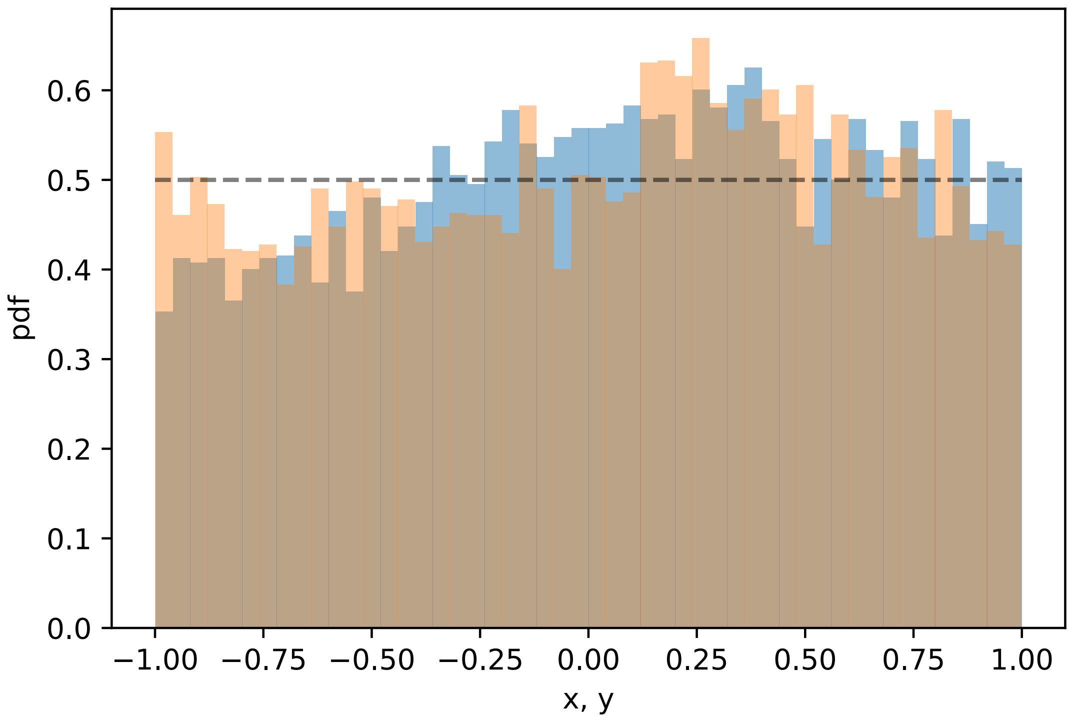
In [11]:
```python
# Parameters
nb_steps = 10**4 # Number of steps
sigma = 0.2      # Variance for displacements
plist = []       # List of points to keep

# Initial point
p = np.array([0., 0.])

# Counter for acceptance rate
cnt = 0

# Simulation
for i in range(nb_steps):
    dp = np.random.normal(0, sigma, 2)  # Displacement
    p_try = p+dp                        # Possible change
    if p_try[0] <= 1.0 and p_try[0] >= -1.0 and p_try[1] <= 1.0 and p_try
        p = p_try                     # Accept change if in square, otherw
        cnt += 1                      # Count change accepted
    plist.append(p)                   # Keep point in list even if not cha

# Convert list of arrays into a genuine numpy array
newplist = np.vstack(plist)

# Fraction of moves accepted
print(cnt/nb_steps, '% of moves accepted.')

# Plot points in (x,y) plane
plt.figure(figsize=(5, 5))
plt.plot(newplist[:, 0], newplist[:, 1], 'o', alpha=0.2)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
0.8585 % of moves accepted.
```



This looks uniform. To confirm, we plot the histogram of the x and y positions:

```
In [19]: plt.hist(newplist[:, 0], bins=50, alpha=0.5, density=True)
         plt.hist(newplist[:, 1], bins=50, alpha=0.4, density=True)
         plt.plot(np.linspace(-1,1,2), 0.5*np.ones(2), 'k--', alpha=0.5)
         plt.xlabel('x, y')
         plt.ylabel('pdf');
```



You can play with the standard deviation of the displacements and see that the acceptance ratio is high for low σ (tiny moves will necessarily be accepted) while the acceptance ratio is low for high σ (big moves are likely to fall outside the square and are therefore not accepted).

b)

We use the code before without displaying (or even keeping) the points and just add the estimator for π, known from CW1.

In [20]:
```
# Parameters
nb_steps = 10**5
var = 0.2
pi_est = 0.0

# Initial point
p = np.array([0., 0.])

# Counter for acceptance rate
cnt = 0

# Simulation
for i in range(nb_steps):
    d = np.random.normal(0, var, 2)
    p_try = p+d
    if p_try[0] <= 1.0 and p_try[0] >= -1.0 and p_try[1] <= 1.0 and p_try
        p = p_try
        cnt += 1

    # Check if point falls in circle
    if p[0]**2+p[1]**2 <= 1:
        pi_est += 4.0

print('Estimation of pi:', pi_est/nb_steps)
print('Acceptance ratio:', cnt/nb_steps)
```

```
Estimation of pi: 3.15444
Acceptance ratio: 0.84839
```

c)

No, we can't because the series of points generated by the Markov chain are *not* independent.