Problem Set 2

December 4, 2021

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

import scipy.stats as sts

import networkx as nx

from PIL import Image, ImageDraw, ImageFont

from random import random
from numpy import arctan2, pi
from scipy.stats import norm

from scipy import signal
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

1 Problem 1: Rainfall Records

According to Ru Geld, Berlin received 511.12 mm of rainfall in 2020. Historical records stretching further back for the city were hard to come by, but a timeseries plot of annual rainfall in Germany between 2011 and 2020 suggested a random walk-like behavior: roughly even number of steps up and down, and a symmetrical pattern of step sizes (largely the same, but with few very large/very small exceptions).

As a result, we will assume that rainfall is normally distributed with mean 511, and standard deviation matched with the standard deviation of rainfall in Germany between 2011 and 2020. This assumes that the variation in weather in the city of Berlin will approximately follow the variation in weather in the country as a whole, albeit with a mean nudged to the 2020 recorded value.

```python
# Centererd on 511.1, rainfall in 2020
berlin_2020 = 511.12

# 2011 - 2020
annual_germany = [
    732.9, 767.5, 778.7, 727.1,
    701.3, 733.3, 858.7, 586.3,
    586.3, 735, 704.9]
```
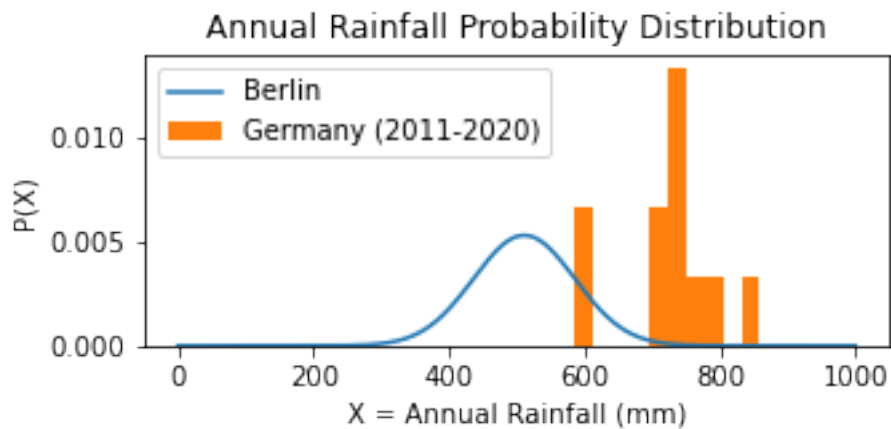
1

```
variation   = np.std(annual_germany)

x = np.linspace(0,1000, 1000)
berlin_rainfall = sts.norm(berlin_2020, variation)

plt.figure(figsize = (5,2))
plt.plot(x, berlin_rainfall.pdf(x), label = "Berlin")
plt.hist(annual_germany, density=True, label = "Germany (2011-2020)")
plt.xlabel("X = Annual Rainfall (mm)")
plt.ylabel("P(X)")
plt.title("Annual Rainfall Probability Distribution")
plt.legend()
plt.show()
```


Content with the approximation, we can now simulate a century of rain by draing samples from this probability density function,

```
[ ]: def sim_century(n = 101):
        """
        Simulates annual rainfall over n years.

            Parameters:
                n (int): Years to simulate

            Returns:
                record_years (int) : Number of record years observed (high and low)
                recorded_rain (arr): Rainfall for each year, for plotting
                first_record (int) : Year of first high/low record
        """

        recorded_rain = np.zeros(n - 1)
```

```python
    first_year = berlin_rainfall.rvs()
    recorded_rain[0] = first_year

    record_high = first_year
    record_low = first_year

    record_years = 0
    first_record = 0

    for i in range (n - 1):
        # Simulate rain in current year
        this_year =  berlin_rainfall.rvs()
        recorded_rain[i] = this_year

        # Check is this is a record year
        if this_year > record_high:
            record_high = this_year
            record_years += 1
            if not first_record:
                first_record = i + 1

        if this_year < record_low:
            record_low = this_year
            record_years += 1

    return record_years, first_record, recorded_rain

plt.figure(figsize = (5,2))
plt.plot(sim_century()[2])
plt.ylabel("Annual Rainfall (mm)")
plt.xlabel("Year #")
plt.title("Simulated Century")
plt.show()
```
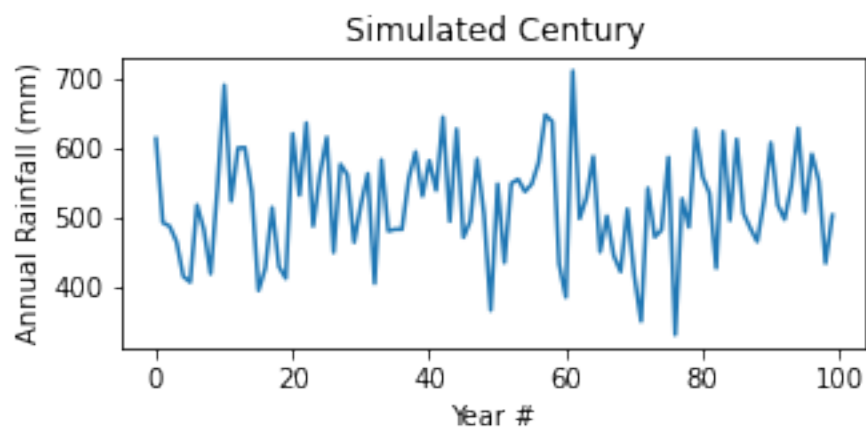
2. Next, simulate the annual rainfall in Berlin for 101 consecutive years (labeled Year 0 up to Year 100) by generating i.i.d. samples from your chosen probability distribution. A rainfall value is a record high if it is greater than those in all previous years (starting with Year 0), and a record low if it is lower than those in all previous years. In the century starting from Year 1 and ending with Year 100 (including those two years), produce a histogram to show the distribution over the number of "record" years – i.e., the number of years that have either a record high or a record low. Show empirically that the expected value of this distribution is,

$$2\sum_{i=1}^{100}\frac{1}{i+1}$$

```python
# MC simulation
def mc_records(iter = 10000):
    """Simulate iter (int) centuries and return number of record years (list)"""
    records = []
    firsts  = []

    for i in range(iter):
        record, first, years = sim_century()
        records.append(record)
        firsts.append(first)

    return records, firsts

sim_records, sim_firsts = mc_records()

# Plot Results
plt.figure(figsize = (5,4))
i = np.arange(1,101)

# Compute theoretical expected value
theoretical_avg_record = round(2*sum(1/(i+1)),3)

plt.axvline(
    theoretical_avg_record,
    label = f'Theoretical Mean = { theoretical_avg_record}',
    color = 'black')
plt.axvline(
    np.mean(sim_records),
    label = f'Empirical Mean = {np.mean(sim_records)}',
    color = 'black'
)
plt.hist(
```
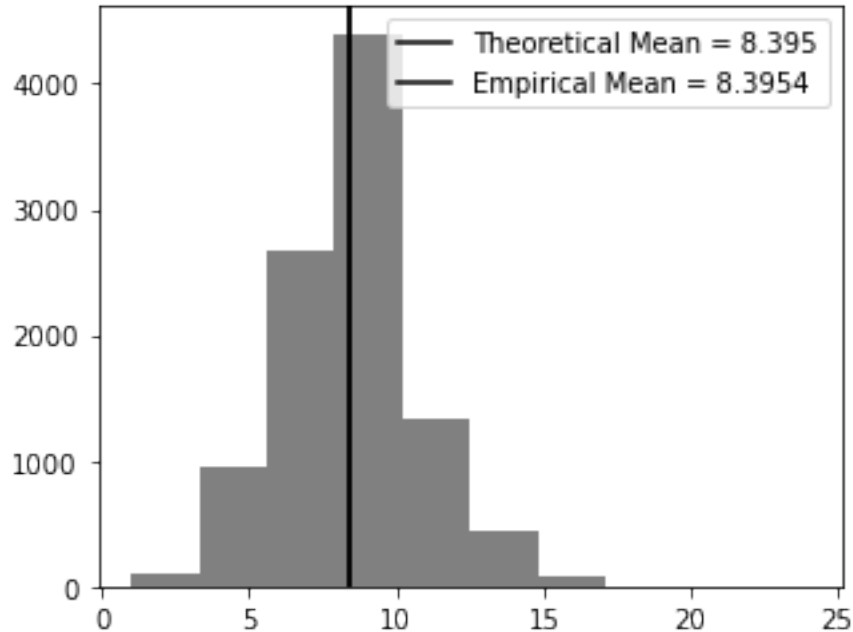
```
    sim_records,
    color = 'grey')
plt.legend()
plt.show()
```


The theoretical and empirical expected values are identical to at least 3 decimal places.

3. Next, use your rainfall distribution to simulate the first year after Year 0 when a record high (not low) rainfall occurs. Produce a histogram of the distribution over the number of years until the first record high after Year 0. Show empirically, using a line plot on top of your histogram, that this distribution has probability mass function

$$P(k) = \frac{1}{k(k+1)}$$

```
[ ]:  # Plot First Record
      plt.figure(figsize = (5,4))
      i = np.arange(1,101)
      # Compute theoretical expected value
      def theoretical_first(k): return 1/(k*(k+1))

      cts, bins, _ =  plt.hist(
          sim_firsts,
          color = 'grey',
          density=True,
          bins = 50)
```
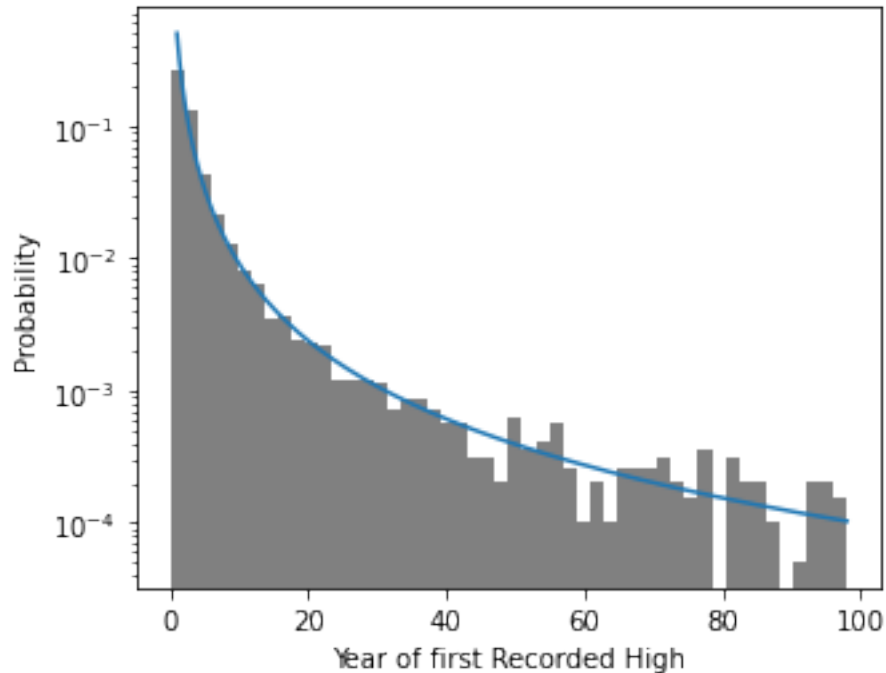
```
x = np.linspace(1 , max(bins), 1000)
plt.semilogy(x, theoretical_first(x))
plt.xlabel("Year of first Recorded High")
plt.ylabel("Probability")
plt.show()
```


After normalizing the histogram and overlaying the pmf, we see a close alignment between the theoretical and empirical distributions, even on a semi-log scale. There is some deviation towards later decades, though this might be because of undersampling and improper binning.

4. Explain why your results in Questions 2 and 3 do not depend on your choice of rainfall distribution in Question 1.

(TODO)

2   Problem 2: Metropolis-Hastings

- Implement the Metropolis-Hastings algorithm.
- Explain what your proposal distribution is and why you chose it.
- Explain the output from the algorithm, including how long it takes to reach equilibrium and how many steps you take between samples (and why).

In this problem, we attempt to draw samples from an unnormalized posterior distributed according to the following function,

```
[ ]: def unnormalized_distribution(x, y):

         measurements = [2.349080, -1.276010, -2.081865]
         buoys = [[-2.5, 2], [2, -5], [-3, -4]]
         sigma = 10/180*pi
         posterior = 1
         for i in range(3):
             bearing = arctan2(buoys[i][1] - y, buoys[i][0] - x)
             difference = (bearing - measurements[i] + pi) % (2 * pi) - pi
             posterior *= norm.pdf(difference, loc=0, scale=sigma)
         return posterior
```

To take iid samples from the posterior, we will use a Metropolis-Hastings sampler `custom_mh`. Each iteration in `n` total iterations, it draws a sample from a bivariate normal proposal distribution, comprising an `x` and `y` parameter draw. The x-normal has a scale parameter `eps_x` of 1, while the y-normal has a scale parameter `eps_y` of 2. These were set because of the oblong contours of the posterior distribution, where there is more variation along y than x. A larger `eps_y` will allow the sampler to cover the space of the distribution more easily, if at the cost of a lower acceptance rate (and in turn, convergence time). Overall, a bivariate normal was chosen because of how easy it was to encode priors about the shape and form of the posterior into its 2 parameters.

To account for the early drift from initialization (1,1) to the posterior, we discard the first 10 samples (a `burn_in` period). Additionally, samples were taken every `step` iterations, to decorrelate samples as much as possible and place some guarantees on the samples being iid,

```
[ ]: def custom_mh(n = 100000, step = 20, burn_in = 10):
         """
         Performs Metropolis-Hastings sampling over a poserior
         """
         samples = np.zeros((2,n//step), dtype = np.float16)
         rejections = np.zeros(n)

         # Initializations, which will be discarded
         curr_x = 1
         curr_y = 1

         eps_x = 1
         eps_y = 2

         for i in range(n):
             proposal_x = norm(curr_x, eps_x).rvs()
             proposal_y = norm(curr_y, eps_y).rvs()

             proposal = unnormalized_distribution(proposal_x,proposal_y)
             current = unnormalized_distribution(curr_x, curr_y)
             ratio = proposal/current

             accept = random() < min(1,ratio)
```

```
        if accept:
            curr_x = proposal_x
            curr_y = proposal_y
        else:
            rejections[i] = 1

        if i % step == 0 and i > burn_in:
            samples[:, i//step] = curr_x, curr_y

    return samples

samples = custom_mh()
```

```
fig, axs = plt.subplots(1,3, figsize = (10,3), sharey=True)

Z, x, y = np.histogram2d(
    samples[1,:],
    samples[0,:],
    bins = [50,50])
xx,yy = np.meshgrid(x,y)
c = unnormalized_distribution(xx,yy)

for ax, alpha, title in zip(axs.flatten(), [1,0.5,0], ["Target Density",␣
 ↪"Target Density/Samples", "Samples"]):
    ax.imshow(
        Z,
        interpolation= 'nearest',
        alpha = 1 - alpha,
        origin= 'lower',
        extent=[x[0], x[-1], y[0], y[-1]],
        cmap = "Greys")

    ax.contour(
        xx, yy, c,
        15,
        alpha = alpha,
        cmap = 'Greys'
    )
    ax.set_xlabel("x")
    if alpha == 1:
        ax.set_ylabel("y")
    ax.set_title(title)

plt.show()
```
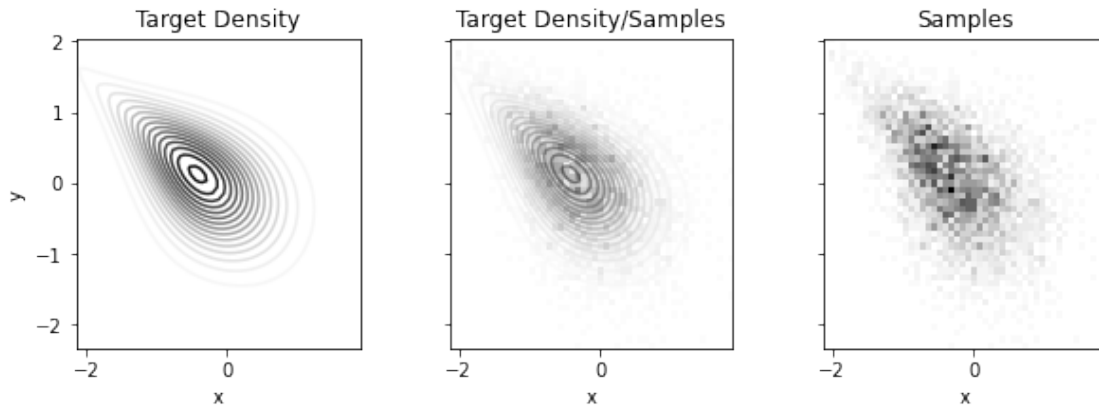
From the plot above, we can see that the MH sampler seems to have approximated the comet-like shape of the posterior (Target Density, Left), given by a tight overlap of the Target Density/Samples (Center). In Samples (right), we can see that the tall posterior mode at ~(-0.2,0) was also approximated well, with darker patches corresponding to that region. For more granular detail, the following plot was produced,

```python
fig, ax = plt.subplots(figsize=(5, 5), sharex = False)

# the scatter plot:
ax.plot(
    samples[0,:],
    samples[1,:],
    ',',
    alpha = 1,
    c = 'grey')

# Set aspect of the main axes.

# create new axes on the right and on the top of the current axes
divider = make_axes_locatable(ax)
# below height and pad are in inches
ax_histx = divider.append_axes("top", 0.8, pad=0.1, sharex=ax)
ax_histy = divider.append_axes("right", 0.8, pad=0.1, sharey=ax)

ax_histx.xaxis.set_tick_params(labelbottom=False)
ax_histy.yaxis.set_tick_params(labelleft=False)

ax_histx.hist(samples[0,:], bins=35, color = 'grey')
ax_histy.hist(samples[1,:], bins=35, orientation='horizontal', color = 'grey')

# Plot autocorrelation
plt.show()
```
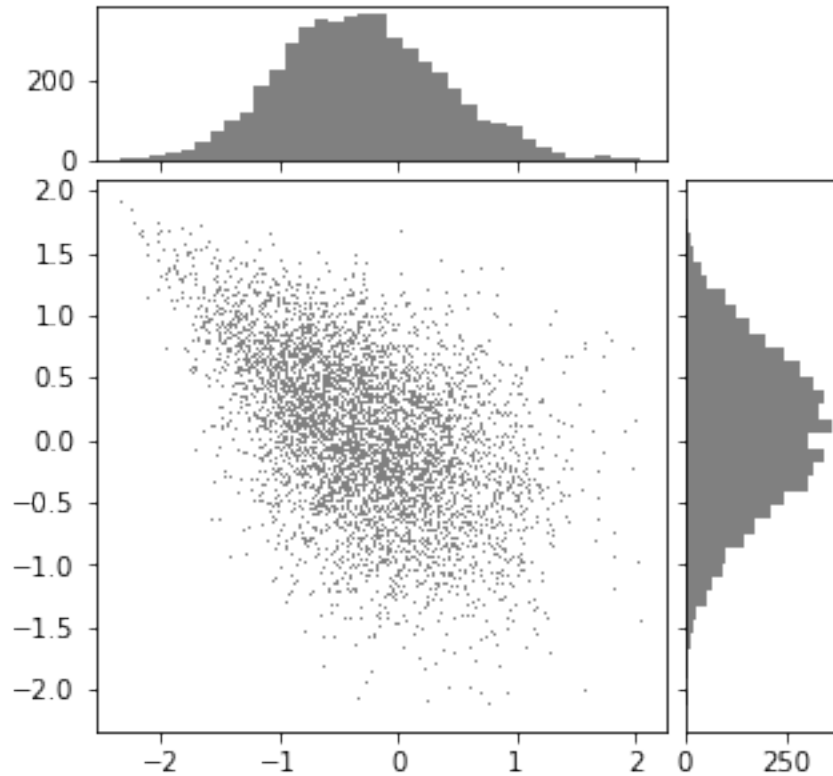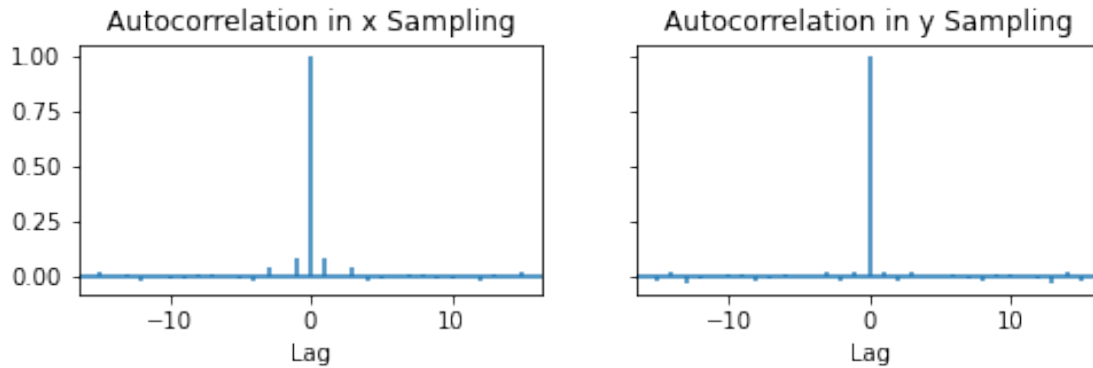
The last check we need to perform is on autocorrelation,

```python
fig, ax = plt.subplots(1,2, figsize=(8, 2), sharey = True)

for i,p in zip(range(2), ['x', 'y']):
    ax[i].acorr(
        samples[i,:],
        maxlags=15,
        detrend=lambda x: signal.detrend(x, type='linear'))
    ax[i].set_xlabel("Lag")
    ax[i].set_title(f"Autocorrelation in {p} Sampling")

plt.show()
```

In an autocorrelation plot, we guage how well a series predicts a lagged version of itself, or, if there is a **dependence** between the next set of points, given the current set.

Both plots indicate very low autocorrelation (the perfect correlation at Lag $= 0$ just implies that there is a series is perfectly correlated with itself, so it can be ignored), which loosely promises independence between samples.

3 Problem 3: Diameter and Average Distance

1. Provide an example of a graph with at least 6 nodes where the diameter is equal to the average distance. Explain why the metrics are equal.

```python
def get_shortest_paths(g):
    """ Return the shortest paths in g from each node to any other node."""
    a = nx.shortest_path(g)
    shortest_paths = []

    for target in a.values():
        for path in target.values():
            # exclude self to self paths
            shortest_path = len(path) - 1
            if shortest_path > 0:
                shortest_paths.append(len(path) - 1)

    return shortest_paths

def draw_paths(paths, ax):
    """Draw hist of shortest paths on ax"""
    ax.hist(paths)
    ax.axvline(
        np.mean(paths),
        label = "Average Distance")
    ax.axvline(
        np.max(paths),
```

```
        label = "Diameter")
    plt.legend()
```

Over a distribution of path lengths in a graph, the average path length corresponds to its mean, while the diameter corresponds to its maximum. As a result, we need a graph where the average is exactly equal to its mean.
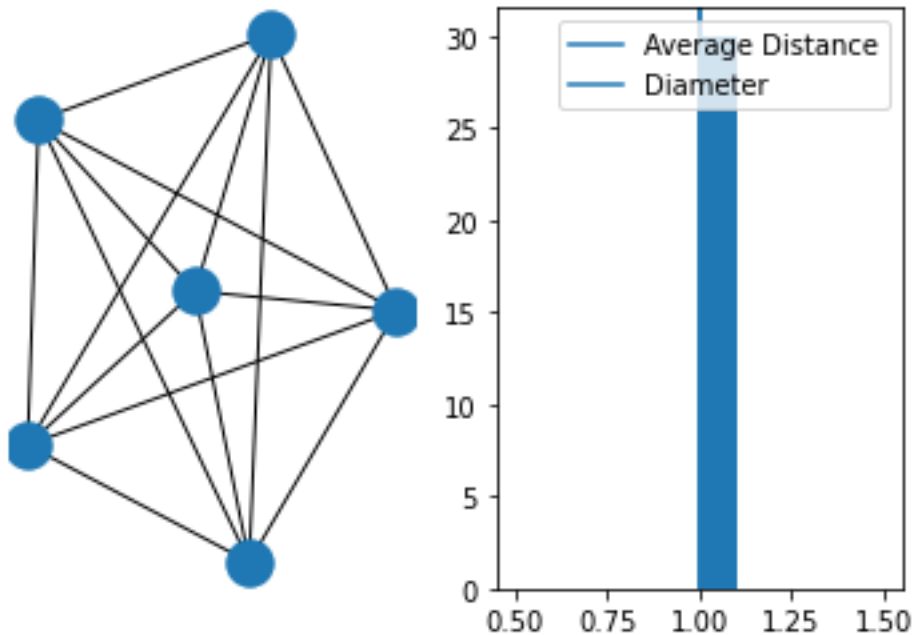
A complete graph is one such example, where there are $n(n-1)$ shortest paths with length 1. In a purely unimodal distribution, the mean must be the same as the max,

```
[ ]: fig, axs = plt.subplots(1,2)
     g = nx.complete_graph(n=6)
     nx.draw(g, ax = axs[0])

     draw_paths(get_shortest_paths(g), axs[1])

     ratio = nx.algorithms.distance_measures.diameter(g)/nx.
      ↪average_shortest_path_length(g)
     ratio
```

[ ]: 1.0

2. Describe an example of a graph with at least 6 nodes where the diameter is more than 3 times as large as the average distance. Explain why the diameter is more than 3 times the average distance.

Here, we begin with a complete graph and iteratively add nodes to a 'tail' until we reach the target ratio of 3:1,

```
[ ]: fig, axs = plt.subplots(1,2, figsize = (9,3))

     # For plotting
     k = 5
     ratios = np.zeros(k)
     diameters = np.zeros(k)
     avgs = np.zeros(k)

     g = nx.complete_graph(n=25)

     # Cache max node
     last_node = max(g.nodes())

     for i in range(k):
         g.add_edge(last_node, last_node + 1)
         last_node += 1
         diameters[i] = nx.algorithms.distance_measures.diameter(g)
         avgs[i] = nx.average_shortest_path_length(g)
         ratios[i] = diameters[i]/avgs[i]
         if ratios[i] >=3:
             break

     nx.draw(g, ax = axs[0])
     draw_paths(get_shortest_paths(g), axs[1])

     ratios
```
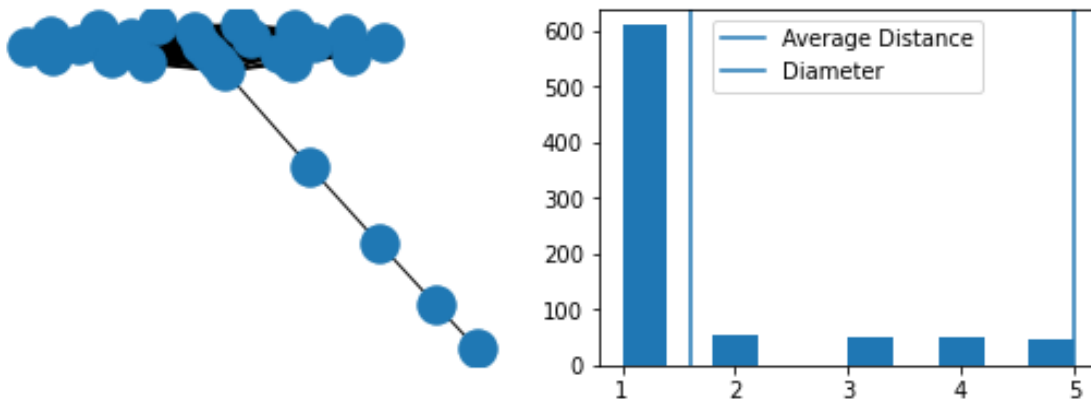
[ ]: array([1.86246418, 2.48349057, 2.87452471, 3.0945122 , 0.   ])

For a complete graph of $n = 25$ nodes, we need to add 4 nodes to the tail for the ratio of diameter to average path length to be greater than 3 (here, ~3.09). Whenever we add a new node p to the tail, the diameter increases by 1 but the average path length increases at a slower rate, since the vast majority of path lengths is concentrated in the connected component. So as long as the connected

component is large relative to the additions to the tail, we can see large diameters.

```python
def compute_ratios(n=25, k = 100, dual = True):

    """
    Parameters:
        dual (bool): store empirical values as well. Setting to False will␣
 ↪substantially
                     speed up computation.

    """

    # For plotting
    cols = dual + 1
    diameters = np.zeros((k,cols))
    avgs = np.zeros((k,cols))

    g = nx.complete_graph(n=n)
    # Cache max node
    last_node = max(g.nodes())
    # Initialize
    diameters[0,:] = nx.algorithms.distance_measures.diameter(g)
    avgs[0,:] = nx.average_shortest_path_length(g)

    for i in range(1,k):
        # Add new node to tail
        g.add_edge(last_node, last_node + 1)
        last_node += 1

        # Tally empirical values, if needed
        if dual:
            diameters[i, 1] = nx.algorithms.distance_measures.diameter(g)
            avgs[i, 1] = nx.average_shortest_path_length(g)

        # Count total min path lengths
        p = np.arange(1,i + 1)
        paths_from_k =  sum((n - 1)*(p + 1) + p/2 + (p**2)/2 + ((i - p)*(1 + i␣
 ↪- p))/2)
        paths_to_k =  n*(n-1) + (n - 1)*sum(p + 1) + sum(p)

        # Compute theoretical avg
        avg_path = (paths_from_k + paths_to_k) / ((n + i)*(n + i - 1))
        avgs[i,0] = avg_path

        # Theoretical diameter
        diameter = i + 1
        diameters[i,0] = diameter
```

```
    ratios = diameters/avgs

    return {"ratios":ratios, "avgs":avgs, "diameters":diameters}
```

```
[ ]: from matplotlib import cm

fig, axs = plt.subplots(1,3, figsize = (15,5))
n = np.arange(10,300,10)
cmap = cm.get_cmap('viridis_r' )

for i in range(len(n)):
    for ax,d in zip(axs.flatten(), compute_ratios(n = n[i], dual = False).
 →items()):
        ax.plot(
            d[1],
            ',',
            label = f"$n = {i}$",
            color = cmap(np.cumsum(n)[i]/sum(n)),

            )
        ax.set_title(d[0].upper())
        ax.set_xlabel('$k$ Nodes added to Tail' )
        ax.set_ylabel(d[0])

sm = plt.cm.ScalarMappable(cmap=cmap, norm = plt.Normalize(vmin = n.min(), vmax␣
 →= n.max()))
cb = plt.colorbar(sm)
cb.set_label("Nodes $n$ in Cluster")

plt.show()
```
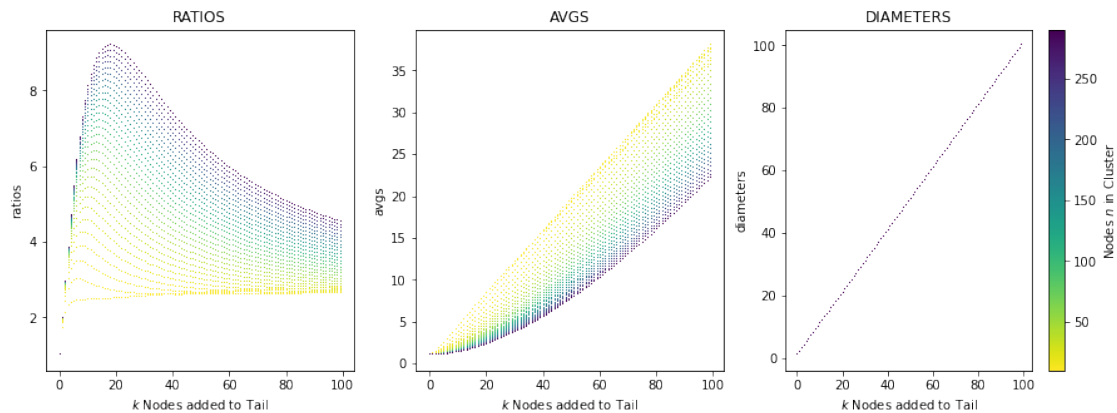

comment on computational complexity

$O((N + k)\hat{\,}2) :: O(1)$.

```python
# Contour over search space
def compute_ratios(nodes, tail):
    g = nx.complete_graph(n=nodes)

    # Cache
    last_node = max(g.nodes())
    for i in range(tail):
        g.add_edge(last_node, last_node + 1)
        last_node += 1

    diameter = nx.algorithms.distance_measures.diameter(g)
    avgs    = nx.average_shortest_path_length(g)
    ratios = diameter/avgs

    return ratios


compute_ratios_vec = np.vectorize(compute_ratios)


nodes = np.arange(10,100,10)
tail = np.arange(0,100,10)

xx, yy = np.meshgrid(nodes, tail)
Z = compute_ratios_vec(xx,yy)

fig = plt.figure(figsize=plt.figaspect(1), dpi = 100)

ax = fig.add_subplot(1,1,1,projection = '3d')
ax.set_ylabel("Nodes added to Tail")
ax.set_xlabel("Nodes in Graph")
ax.set_zlabel("Ratio")

ax.contour3D(xx,yy,Z, 100, cmap = 'viridis', alpha = 0.5)
ax.view_init(30,10)
ax.grid(False)
```
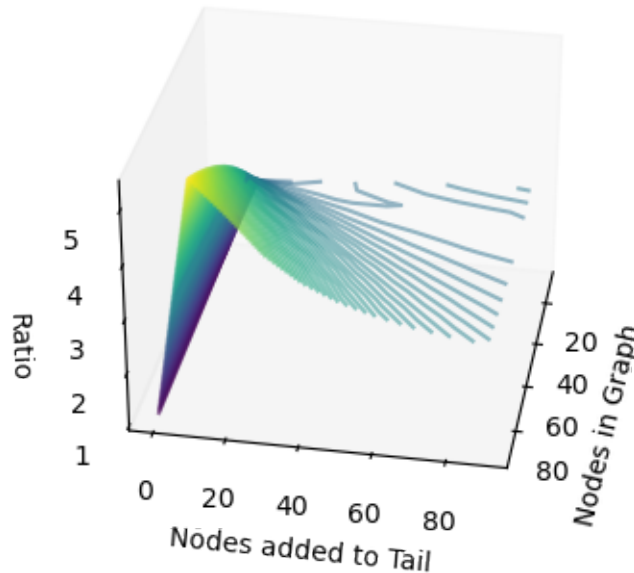
3. Describe how you can construct a graph in which the diameter is more than c times as large as the average distance – for any positive value of c. Briefly explain why or prove that the diameter is more than c times the average distance.

To calculate average path distance a_p we use,

$$a_p = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$$

$d(s,t)$ is the shortest path from node s to node t (see `get_shortest_paths()`),

V is the set of nodes in the graph G,

n is the number of nodes in G.

4 Problem 4: Hopfield Network

tallying coincident pixels (neurons that fire together wire together) fully connected - every neuron is connected to every other pixel, like the graphs in the previous problem. A fully connected graph also forms loops, which is why Hopfield networks can be thought of as the simplest case of Recurrent Neural Networks.

and updates (`Hopfield.update()`) can be thought of as a recall task by iteratively updating a state vector (`Hopfield.s`)

```
def generate_bitmap(text, size):
    """ Return a numpified image of text, with values in [-1,1]"""
```

17

```python
    pil_font = ImageFont.truetype(
        "/Library/Fonts/Arial.ttf",
         size = int(2*size//len(text)))

    text_width, text_height = pil_font.getsize(text)

    canvas = Image.new('RGB', [size, size], (255, 255, 255))
    # draw the text onto the canvas
    draw = ImageDraw.Draw(canvas)
    offset = ((size - text_width)//2,
              (size - text_height)//2)
    white = "#000000"
    draw.text(offset, text, font=pil_font, fill=white)

    # Clip Values
    img = (255 - np.asarray(canvas)) / 255.0

    # Constrain values to {-1, 1} selon McKay
    img = np.where(img > 0, -1, 1)

    # Return just 1 color channel
    return img[..., 1]

class Hopfield:
    def __init__(self, shape, rule = "Hebbian") -> None:
        """
        Initialise a hopfield network
        Params:
            shape (arr[2]): shape of memory in [w,h]
        """
        # Width
        self.w = shape[0]
        # Height
        self.h = shape[1]
        self.rule = rule

        self.truths = []

        # step*step image -> N = 2step
        self.N = self.w * self.h
        if rule == "Hebbian":
            self.capacity = 0.138*self.N

        # Fully connected, W_ij represents connections from
        # Node i to Node j
        self.W = np.zeros((self.N,self.N))
```

```python
        # Current State
        self.s = np.zeros((self.N,1))

    # Activation i = state.w[i]
    def activate(self, type = 'async'):

        if type == 'async':
            # Async: pick a random node
            i = np.random.randint(0,self.N)
            a = self.W[i,:]@self.s

            # if dot prod < 01
            if a < 0:
                # Activate to -1
                self.s[i] = -1
            else:
                # Activate to +1
                self.s[i] = 1
        elif type == 'sync':
            self.s = np.sign(self.W@self.s)

    def update(self,n_updates = 1000, step = 10, display = True, type =␣
↪'async'):
        """Update one node at a time (asyn) or all at once"""

        # Plot every few frames
        if display:
            frames = n_updates//step
            fig, axs = plt.subplots(
                2,
                frames//2,
                tight_layout = True,
                figsize = (10,3)
                )

            axes = axs.flatten()

            t = 0
            for i in range(n_updates):
                self.activate(type)
                if i%step == 0:
                    self.show(ax = axes[t], idx = i+1)
                    t += 1
        else:
            # Collect data on deviation from true
            loss = []
            energy = []
```

```python
        for i in range(n_updates):
            self.activate()
            try:
                loss.append(self.measure_loss(self.truths, self.s))
                energy.append(self.measure_energy())
            except:
                pass

        return loss, energy

    def measure_loss(self, truth, curr_state):
        """Sum of abs deviations from grouth truth"""
        return np.sum(np.abs(truth - curr_state), axis = 1)

    def measure_energy(self):
        return -0.5 * np.sum(np.multiply(np.outer(self.s,self.s), self.W))

    def store(self, mem):
        """Store a memory mem in Weights"""

        # Flatten pixel grid to array
        mem = mem.flatten()
        assert len(mem) == self.N
        self.truths.append(mem)

        # Learn Weights with Hebbian Rule
        # :: fully connected adjacency matrix
        # Add on top of last memory
        if self.rule == "Hebbian":
            self.W  += np.outer(mem,mem)/len(mem)

        elif self.rule == "Storkey":
            hebbian = np.outer(mem,mem)
            net_in = self.W.dot(mem)

            pre_syn = np.outer(mem, net_in)
            post_syn = pre_syn.T

            self.W += (1./len(mem))*(hebbian - pre_syn - post_syn)

        # as j != i in update
        np.fill_diagonal(self.W, 0)

    def show(self, ax = False, idx = 0):
        # Reshape representation
        representation = self.s.reshape(self.w, self.h)
```

```python
        if ax:
            ax.imshow(representation, cmap = 'Greys')
            ax.set_title(f'$s = {idx}$')
            ax.axis('off')
        else:
            plt.imshow(representation, cmap = 'Greys')

    def perturb_input(self, letter:np.array, amount:float = 0.4):
        """Add color-in noise to letter and store in state """
        size = letter.size
        n_pixels = round(size * amount)

        perturbed_letter = np.copy(letter)

        pixels = np.random.randint(0, size - 1, size = n_pixels)
        pixels = np.unravel_index(pixels, letter.shape)
        perturbed_letter[pixels] = 1

        self.s = perturbed_letter.flatten()

        return letter

def plot_states(data, e, net, labels = ["A","B","C"], ):
    """" Plot Pattern (in labels) Recall and Energy change over state updates.
↪"""
    data = np.array(data)

    fig, axs = plt.subplots(
        2,1,
        figsize = (5,3),
        gridspec_kw= {'height_ratios': [3,1]},
        tight_layout = True,
        sharex = True )

    # Pattern Recall
    axs[0].axhline(0, linestyle = '--', label = "Ground Truth", color = 'grey')
    for i,l in zip(range(len(labels)), labels):
        axs[0].plot(data[:,i], label = l)

    axs[0].set_xlabel("$i$ state updates")
    axs[0].set_ylabel("Loss")
    axs[0].set_title(f"Hopfield with {net.rule} Rule, {len(net.truths)}␣
↪patterns")
    axs[0].legend()

    axs[1].plot(e)
    axs[1].set_xlabel("Iterations")
```

```
    axs[1].set_ylabel("Energy (J)")

    plt.show()

A = np.array([
    [-1, 1, 1, -1],
    [1, -1, -1, 1],
    [1, 1, 1, 1],
    [1, -1, -1, 1],
    [1, -1, -1, 1]
])
B = np.array([
    [1, 1, 1, -1],
    [1, -1, -1, 1],
    [1, 1, 1, -1],
    [1, -1, -1, 1],
    [1, 1, 1, -1]
])
C = np.array([
    [-1, 1, 1, -1],
    [1, -1, -1, 1],
    [1, -1, -1, -1],
    [1, -1, -1, 1],
    [-1, 1, 1, -1]
])
```

We can

```
[ ]: y = generate_bitmap(":)", 40)
     h = generate_bitmap("</3", 40)

     hope = Hopfield(y.shape)
     hope.store(y)
     hope.store(h)

     hope.perturb_input(h, amount=0.9)
     hope.update(n_updates=5000, step = 500, type = 'async', display  = True)
```
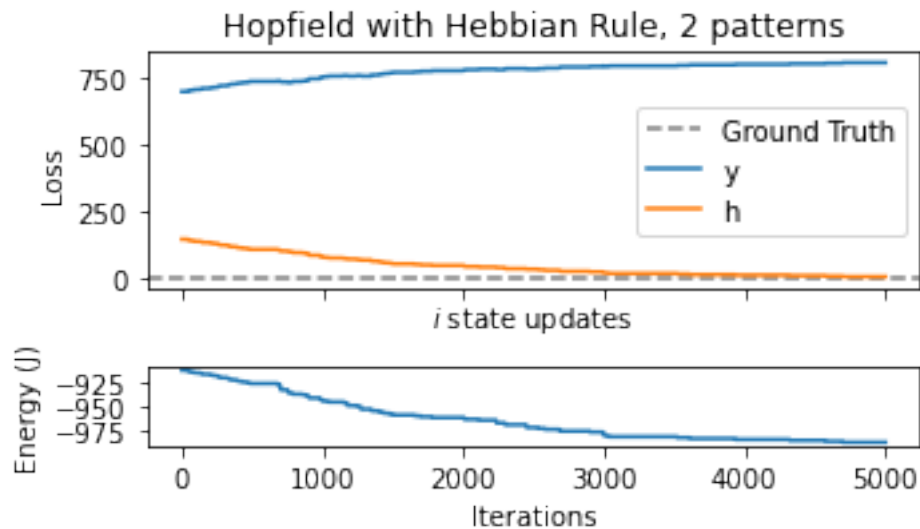
```
hope.perturb_input(h, amount=0.8)

data, e = hope.update(n_updates=5000, step = 100, type = 'sync', display  =␣
 ↪False)
plot_states(data, e, hope, labels = ["y", "h"])
```
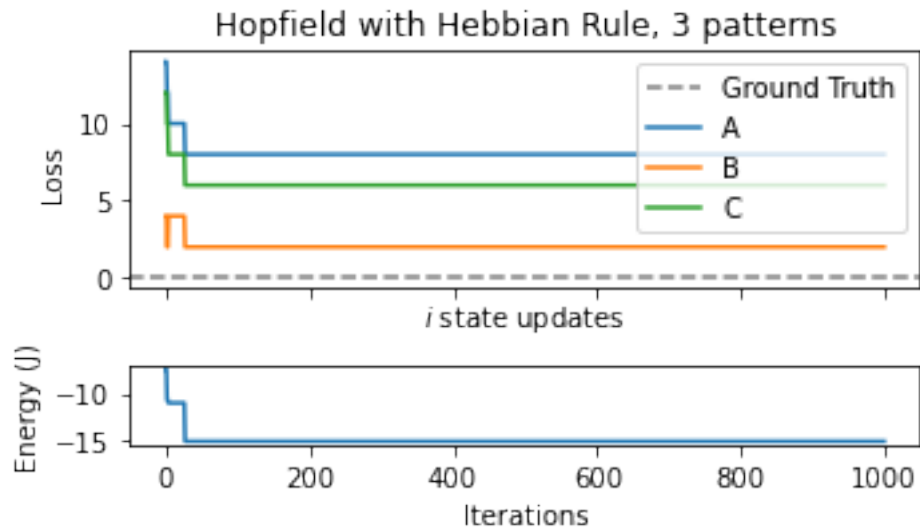

Fires if previous state is associated well enough with the current state.

```
hebbian = Hopfield(A.shape, rule = "Hebbian")
[hebbian.store(i) for i in [A,B,C]]

hebbian.perturb_input(B, amount=0.5)
data, e = hebbian.update(n_updates=1000, step = 10, type = 'async',␣
 ↪display=False)

plot_states(data, e, hebbian, labels = ["A", "B", "C"])
```
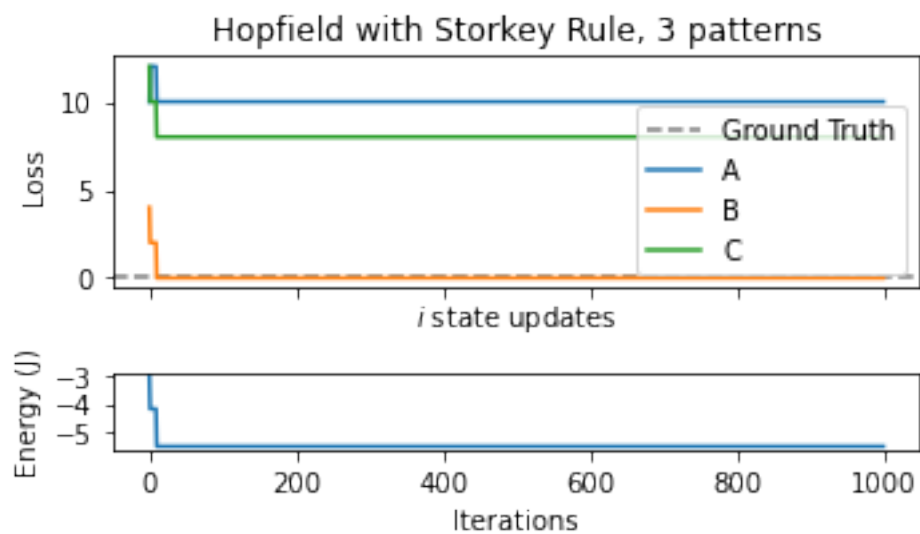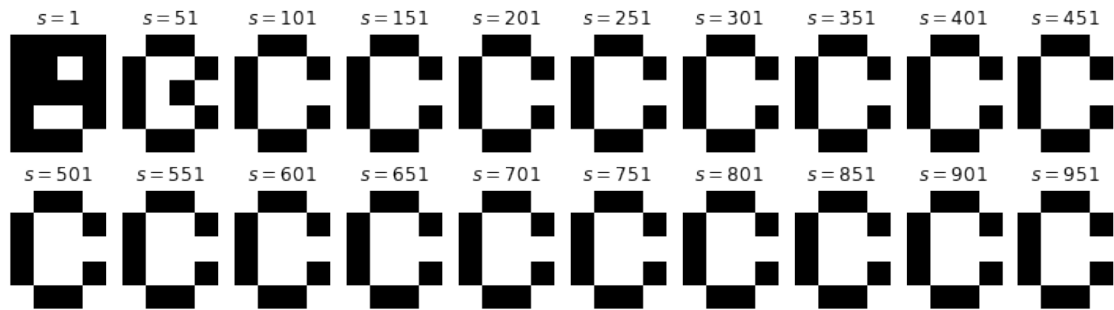
Hopfield with Hebbian Rule, 3 patterns


```
strokey = Hopfield(A.shape, rule = "Storkey")
[strokey.store(i) for i in [A,B,C]]

strokey.perturb_input(B, 0.9)
data, e = strokey.update(n_updates=1000, step = 10, type = "async", display =
  →False)

plot_states(data, e, strokey, labels = ["A", "B", "C"])
```

Hopfield with Storkey Rule, 3 patterns

```
strokey.perturb_input(B, 0.9)
strokey.update(n_updates=1000, step = 50, type = "async")
```



```
hebbian.perturb_input(B, 0.9)
hebbian.update(n_updates=1000, step = 50, type = "async")
```