Assignment 1

Jens Ifver

September 2021

1 Task 1

```
import numpy as np
import scipy.integrate as integrate
import math

## Functions to estimate the one-step error probability
def gen_patterns(p, N):
    mat = np.random.randint(low = 0, high = 2, size = (p,N))
    mat[mat == 0] = -1
    return mat
def choose_bit(N):
    return np.random.randint(low = 0, high = N)
def choose_pattern(p):
    return np.random.randint(low = 0, high = p)
def asynch_update1(P, bit, pattern): # For w_ii = 0
    c = 0
    for j in range(np.shape(P)[1]):
        for mu in range(np.shape(P)[0]):
            if (j != bit and mu != pattern ):
                c += P[mu, bit] * P[mu, j] * P[pattern, j] * P[pattern, bit]
    return (-1/np.shape(P)[1]) * c
def isIncorrect(c):
    if c > 1: return 1
    else: return 0
def one_step_error1(n_trials, N, p): # For w_ii = 0
    c_sum = 0
    for it in range(n_trials):
        P = gen_patterns(p,N)
        bit = choose_bit(N)
        pattern = choose_pattern(p)
        c = asynch_update1(P, bit, pattern)
        c_sum += isIncorrect(c)
    print(c_sum)
    return c_sum/n_trials
```

```
# Functions for w_ii != 0
def hebbs(P):
    N = P.shape[1]
    W = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
                W[i,j] = 1/N * np.dot(P[:,i],P[:,j])
    return W
def asynch_update2(W, bit, s_vec): # For w_ii != 0
    return np.sign(W[bit,:].dot(s_vec) )
def one_step_error2(n_trials, N, p): # For w_ii != 0
    n_incorrect = 0
    for it in range(n_trials):
        P = gen_patterns(p,N)
        W = hebbs(P)
        bit = choose_bit(N)
        pattern = choose_pattern(p)
        s_m = asynch_update2(W, bit, P[pattern])
        if (s_m != P[pattern,bit] ):
            n_incorrect += 1
    print(n_incorrect)
    return n_incorrect/n_trials

## Functions for computing theoretical one-step error
def e(x):
    return np.exp(-(x**2))
def erf(z):
    temp = integrate.quad(e,0,z)[0]
    temp = temp * (2/np.sqrt(math.pi))
    return temp
def error_prob(N,p):
    return 1/2 * (1-erf(np.sqrt(N/(2*p))))

## Function for one-step error for all p when w_ii = 0
def one_step_error_pvec1(p):
    result = np.zeros((2,np.shape(p)[0]))
    for it in range(np.shape(p)[0]):
        result[0,it] = error_prob(120,p[it])
        result[1,it] = one_step_error1(10**5, 120, p[it])
    return result

## Function for one-step error for all p when w_ii != 0
def one_step_error_pvec2(p):
    result = np.zeros((2,np.shape(p)[0]))
    for it in range(np.shape(p)[0]):
        result[0,it] = error_prob(120,p[it])
        result[1,it] = one_step_error2(10**5, 120, p[it])
    return result

## Computing estimated one-step error for all values of p together with
## theoretical one-step error prob.
p = np.array([12, 24, 48, 70, 100, 120])
result_zero_diag = one_step_error_pvec1(p)
print(result_zero_diag)
result_non_zero_diag = one_step_error_pvec2(p)
print(result_non_zero_diag)
```

2 Task 2

```
import numpy as np

# Load patterns
x1= np.array([ [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1,
    1, -1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, 1, 1, 1, -1, -1, 1,
        1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1,
            1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1,
                1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1,
                    -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[
                        -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1,
                            1, 1, 1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[
                                -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1,
                                    -1, -1, -1, -1, -1, -1, -1, -1] ])
x2=np.array([ [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1,
    -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1,
        -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1,
            -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1,
                1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1,
                    -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1,
                        -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1,
                            -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1,
                                1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1,
                                    -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1] ])
x3=np.array([ [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1,
    -1, -1, -1, -1, 1, 1, 1, -1, -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],[
        -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1,
            -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1,
                -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ 1, 1, 1, -1, -1, -1,
                    -1, -1, -1, -1],[ 1, 1, 1, -1, -1, -1, -1, -1, -1, -1],[ 1,
                        1, 1, -1, -1, -1, -1, -1, -1, -1],[ 1, 1, 1, -1, -1, -1,
                            -1, -1, -1, -1],[ 1, 1, 1, -1, -1, -1, -1, -1, -1,
                                -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ 1, 1, 1,
                                    1, 1, 1, 1, 1, -1, -1] ])
x4= np.array([ [ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, 1, -1],[
    -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1,
        -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1,
            1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, 1, 1, 1,
                1, 1, 1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, -1,
                    -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1,
                        1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1,
                            -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1,
                                -1, 1, 1, 1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, 1,
                                    -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1] ])
x5= np.array([ [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1,
    -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1,
        -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1,
            1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, 1, 1, 1, 1,
                1, 1, -1],[ -1, 1, 1, 1, 1, 1, 1, 1, 1, -1],[ -1, -1, -1, -1,
                    -1, -1, -1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, -1, 1, 1,
                        -1],[ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1],[ -1, -1,
                            -1, -1, -1, -1, -1, 1, 1, -1],[ -1, -1, -1, -1, -1,
                                -1, -1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, -1,
                                    1, 1, -1],[ -1, -1, -1, -1, -1, -1, -1, 1,
                                        1, -1] ])
x_dist1 = np.array([[1, 1, -1, -1, -1, -1, -1, -1, 1, 1], [-1, -1, 1, 1, 1, 1, 1,
    1, 1, -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1, -1,
        1, 1, 1, -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1,
            -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1,
                -1, 1, 1, 1, 1, 1, 1, -1, -1], [-1, -1, 1, 1, 1, 1, 1, 1, -1,
                    -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1,
                        -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1,
                            -1], [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1,
                                -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, 1, 1, 1,
                                    1, 1, 1, 1, -1], [-1, -1, 1, 1, 1, 1, 1, 1,
                                        -1, -1]])
x_dist2 = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1]
    1, 1, 1, 1, 1, 1, -1, -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1,
        -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1,
            -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1], [-1, 1, 1,
                1, -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1,
                -1], [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1], [-1, -1, 1, 1, 1, 1, 1,
                    1, -1, -1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1], [-1, -1,
                        -1, -1, -1, -1, -1, -1, -1, -1]])
x_dist3 = np.array([[1, -1, -1, 1, -1, 1, -1, 1, 1, -1], [1, -1, -1, 1, -1, 1, -1, 1, -1
    -1, 1, -1, 1, -1, -1, 1, -1, -1], [1, -1, 1, -1, 1, -1, -1, 1, -1, -1], [1,
        -1, 1, -1, 1, -1, -1, 1, -1, -1], [1, -1, 1, -1, 1, -1, -1, 1, -1, -1],
    [1, -1, 1, -1, 1, -1, -1, 1, -1, -1], [1, -1, -1, 1, -1, 1, -1, 1, 1, -1],
    [1, -1, -1, 1, -1, 1, -1, 1, 1, -1], [1, -1, 1, -1, 1, -1, -1, 1, -1, -1],
    [1, -1, 1, -1, 1, -1, -1, 1, -1, -1], [1, -1, 1, -1, 1, -1, -1, 1, -1, -1],
    [1, -1, 1, -1, 1, -1, -1, 1, -1, -1], [1, -1, 1, -1, 1, -1, -1, 1, -1, -1],
    [1, -1, -1, 1, -1, 1, -1, 1, -1, -1], [1, -1, -1, 1, -1, 1, -1, 1, 1, -1]])

# Flatten arrays
x1 = x1.ravel()
x2 = x2.ravel()
x3 = x3.ravel()
x4 = x4.ravel()
x5 = x5.ravel()
x_dist1 = x_dist1.ravel()
x_dist2 = x_dist2.ravel()
x_dist3 = x_dist3.ravel()

# Construct matrix P
P = np.array([x1,x2,x3,x4,x5])

# Hebb's rule
def hebbs(P):
    N = P.shape[1]
    W = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
            if (j != i):
                W[i,j] = 1/N * np.dot(P[:,i],P[:,j])
    return W

# Asynchronous update functions
def asynch_update(W, s_in):
    s_out = np.sign(np.dot(W, s_in)).astype(int)
    return s_out
def asynch_T_times(W, s_in):
    s_new = s_in
    while True:
        s_old = s_new
        s_new = asynch_update(W,s_old)
        if (s_new == s_old).all():
            break
    return s_new

# Find attractor function
def find_attractor(P, s):
    attractors = np.concatenate((P,-P), axis = 0)
    indices = np.array([np.arange(1,6), -1*np.arange(1, 6)]).ravel()
    for it in range(attractors.shape[0]):
        nr_correct = sum(s == attractors[it,:])
        if (nr_correct == 160):
            return indices[it]
    return 6

# Calculate weight matrix
W = hebbs(P)

# Distorted pattern 1
s_dist1 = asynch_T_times(W,x_dist1)
print("Steady state for dist 1:", "\n",repr(np.reshape(s_dist1, (16,10))))
print("Dist 1 converges to pattern" ,find_attractor(P,s_dist1))

# Distorted pattern 2
s_dist2 = asynch_T_times(W,x_dist2)
print("Steady state for dist 2:", "\n", repr(np.reshape(s_dist2, (16,10))))
print("Dist 2 converges to pattern" ,find_attractor(P,s_dist2))

# Distorted pattern 3
s_dist3 = asynch_T_times(W,x_dist3)
print("Steady state for dist 3:", "\n", repr(np.reshape(s_dist3, (16,10))))
print("Dist 3 converges to pattern" ,find_attractor(P,s_dist3))
```

3 Task 3

```
import numpy as np

# Generate patterns
def gen_patterns(p, N):
mat = np.random.randint(low = 0, high = 2, size = (p,N))
mat[mat == 0] = -1
return mat
# Hebb's rule
def hebbs(P):
N = P.shape[1]
W = np.zeros((N, N)) for i in range(N):
        for j in range(N):
            if (j != i):
W[i,j] = 1/N * np.dot(P[:,i],P[:,j])
return W
# Asynchronous update
def p_(b):
beta = 2
return 1/(1+np.exp(-2*beta*b))
```

```
def stochastic_asynch_update(W, s_in, t):
t = t%W.shape[0]
b = W[t,:].dot(s_in) s_out = np.copy(s_in)
r = np.random.rand(1)[0] prob = p_(b)
if (r < prob):
        s_out[t] = 1
    else:
s_out[t] = -1 return s_out
def calc_order_param_for_t(s, x_mu): return s.dot(x_mu) / (x_mu.shape[0])
def m_experiment(p): T = 2*10**5
P = gen_patterns(p,200) W = hebbs(P)
s_old = P[0,:]
m_mu = 0
for it in range(T):
s_new = stochastic_asynch_update(W,s_old,it)
m_mu += calc_order_param_for_t(s_new, P[0,:]) s_old = s_new
return m_mu / T
def m_average(p,times):
    m_tot = 0
    for ix in range(times):
m_tot += m_experiment(p) return m_tot / times
# Computation for p = 7 m_avg_7 = m_average(7, 100)
print(m_avg_7)
# Computation for p = 45 m_avg_45 = m_average(45, 100) print(m_avg_45)
```