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

```
H = np.loadtxt("Hmatrix2.txt",delimiter=',')
Axes0 = H.shape[0] #Rows
Axes1 = H.shape[1] #Cols
print(Axes0,Axes1)
# print(H)
```

```
    3000 5000
```

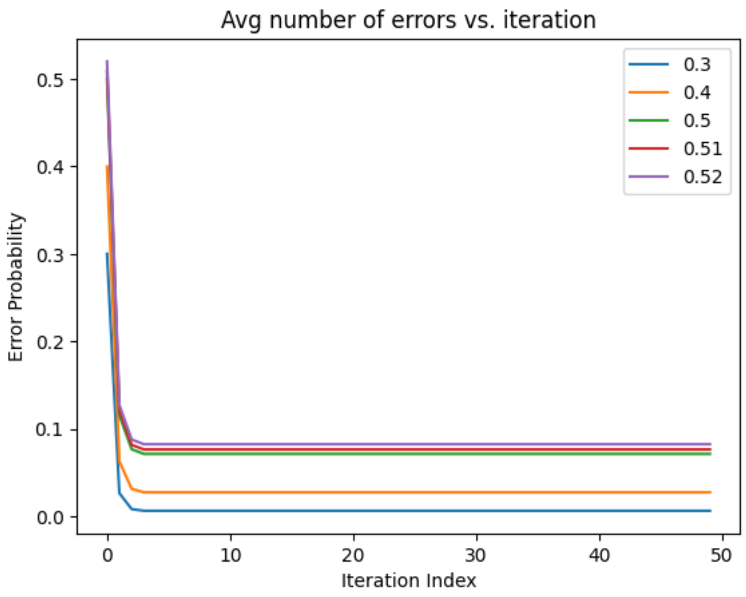
```
class Node:
    def __init__(self, nodeID, connected_nodeList, bitValue,*args,**kwargs):
        self.nodeID = nodeID #Helps to locate node in general list
        self.adjNodes = connected_nodeList # List of connected nodes to a particular node
        self.prevMsg = 2 #Set 2 as there is no initial messsage when started
        self.currMsg = bitValue
        self.pr1 = 0.5 if bitValue==-1 else bitValue #Stores probab=0.5 for erasure else same as bit.
        self.FrmAdjNode = [None]*len(connected_nodeList) #Msg received from adj nodes

# Pi = Pr(ci = 1|yi)
# qij is a message sent by the variable node to the check node
# Every message contains always the pair qij(0) and qij(1) which stands for the amount of belief that yi is a '0' or a '1'.
# rji is a message sent by the check node fj to the variable node. Again there is a rji(0) and rji(1) that indicates the (current) amount
    def varToChk(self,chkNodes):
        #For 1st time sends whatever probability it has otherwise it makes a decision based on what it received from other check nodes (e
        # They use this equation to update their response messages:
        # qij(0) = Kij*(1-Pi)*PI[rj'i(0)] where PI is multiplication of all values except that of the checknode it is currently sending r
        #K must be determined in such a way that qij(0) + qij(1) = 1.
        # Then, it considers decisions of all checknodes to update is currMsg using hard deicision.
        if self.prevMsg==2:
            # print("First Iteration\n")
            for nodeInd in self.adjNodes: # From variable nodes adjacent node list
                idInd = chkNodes[nodeInd].adjNodes.index(self.nodeID) #Index of checknodes in chkNodes list
                chkNodes[nodeInd].FrmAdjNode[idInd] = self.pr1
            self.prevMsg = self.currMsg
        else:
            # print("2+ Iteration\n")
            self.prevMsg = self.currMsg
            for chkNodeInd in self.adjNodes:
                pr1 = self.pr1
                # print(pr1)
                pr0 = 1 - pr1
                for prob1 in self.FrmAdjNode: #Iterate through probab list sent by chk nodes
                    if  self.FrmAdjNode.index(prob1) == self.adjNodes.index(chkNodeInd):
                        continue
                    else:
                        pr1 *= prob1
                        pr0 *= (1 - prob1)
                # if(pr0+pr1)==0:
                #     print(pr0,pr1)
                try:
                    k = 1/(pr0 + pr1)
                except:
                    print("Received message cannot be decoded")
                    raise ZeroDivisionError()
                varNodeInd = chkNodes[chkNodeInd].adjNodes.index(self.nodeID)
                chkNodes[chkNodeInd].FrmAdjNode[varNodeInd] = k*pr1

            new_pr1 = pr1*self.FrmAdjNode[-1] #After last iteration pr1 will have multiplication of all probs except last element
            new_pr0 = pr0*(1 - self.FrmAdjNode[-1])
            # print(self.nodeID, " P(1): ",new_pr1," P(0): ",new_pr0)
            if(new_pr1 > new_pr0):
                self.currMsg = 1
            elif (new_pr0 > new_pr1):
                self.currMsg = 0
            else:
                self.currMsg = -1


    def chkToVar(self,varNodes):
        #check node considers message from variable nodes and calculates response for particular variable node using equation rji(0) = 0.
        for varNodeInd in self.adjNodes:
            prob0 = 0.5
            for pr1 in self.FrmAdjNode: #Iterate through probab list sent by var nodes
                if self.FrmAdjNode.index(pr1) == self.adjNodes.index(varNodeInd):
                    continue
                else:
                    prob0 = prob0*(1-2*pr1)
            chknodeInd = varNodes[varNodeInd].adjNodes.index(self.nodeID)
            varNodes[varNodeInd].FrmAdjNode[chknodeInd] = 0.5 - prob0 #1-(0.5+0.5*(1-2qi'j))
```

```
def soft_decode_bec(varNodes,chkNodes):
    t = 0 #Iteration index
    errorList = [0]*50
    while(t<50):
        #variable node to check node logic
        for varNode in varNodes:
            varNode.varToChk(chkNodes)

        #check node to variable node logic
        for chkNode in chkNodes:
            chkNode.chkToVar(varNodes)

        #Loop break condition and error counting
        currWord = []
        errCnt = 0
        for parity in varNodes:
            msg = parity.currMsg
            if msg == -1:
                errCnt+=1
            currWord.append(parity.currMsg)
        errorList[t] = (errCnt/len(currWord))


        if -1 not in currWord:
            # print(currWord)
            if sum(currWord)%2 == 0:
                return errorList,currWord


        t += 1
    return errorList,currWord
```

```
def generate_msg(p):
    # p = 0.3 #Probability of a bit to be an erasure
    Msg = np.random.randint(0, 1, size=Axes1)
    receivedMsg = Msg.copy()
    receivedMsg[np.random.rand(Axes1) < p] = -1
    # print(receivedMsg)
    return list(Msg),list(receivedMsg)
```

```
def initNodes(receivedMsg):
    varNodes = []
    chkNodes = []
    for i in range(Axes0):
        temp1 = list(H[i].nonzero()[0]) #Finding adj nodes
        chkNodes.append(Node(nodeID=i, connected_nodeList=temp1, bitValue=2)) #Checknode initially has no message

    for i in range(Axes1):
        temp2 = list(H[:,i].nonzero()[0])
        varNodes.append(Node(nodeID=i, connected_nodeList=temp2, bitValue=receivedMsg[i])) #Initializing variable nodes with received mes

    return varNodes,chkNodes
```

```
L = [0, -1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0] #Error case
Nsim = 250
probab_erasure = [0.3, 0.4, 0.5, 0.51, 0.52]
avgError = np.zeros(50)
for p in probab_erasure:
    err_allSim = [0]*Nsim
    for l in range(Nsim):
        originalMsg,errorMsg = generate_msg(p)
        varNodes,chkNodes = initNodes(receivedMsg=errorMsg)
        errorList,decoded_wrd = soft_decode_bec(varNodes,chkNodes)
        err_allSim[l] = errorList

    npError = np.array(err_allSim)
    avgError = np.mean(npError,axis=0)
    plt.plot(np.arange(0, 50), avgError,label=f'{p}')


# print(npError,"\n",avgError)
plt.title('Avg number of errors vs. iteration')
plt.xlabel('Iteration Index')
plt.ylabel('Error Probability')
plt.legend()
plt.show()
```


Avg number of errors vs. iteration
Error Probability
Iteration Index