Goals:

- Test if ergodic, regular, absorbing

1. If absorbing:
    - Number of steps until absorption if start in some state
    - Probability of being absorbed in a certain state if we start in some state
    - Define some function for P(a -> b)
2. If ergodic:
    - Define some function for P(a -> b in n steps)
3. If regular (and thus also ergodic):
    - Calculate fixed vector
    - Mean first time to go from state a -> b

In [1]:
```python
import numpy as np
```

In [2]:
```python
# ergodic & regular
A = np.matrix([
    [0.1, 0.4, 0.5],
    [0.2, 0, 0.8] ,
    [0.1, 0.7, 0.2]
])

# absorbing 3x3
B = np.matrix([
    [0.1, 0.4, 0.5],
    [0, 1, 0] ,
    [0.1, 0.8, 0.1]
])

# ergodic not regular
C = np.matrix([
    [0, 1],
    [1, 0]
])

# absorbing 4x4
D = np.matrix([
    [1, 0, 0, 0],
    [.2, 0, .6, .2],
    [0, 0, 1, 0],
    [.1, .2, .3, .4],
])

# one node ergodic
E = np.matrix([
    [1]
])

# two node non-ergodic
F = np.matrix([
     [0,1],
     [0,1]
]
)

# complex non-ergodic (multiple equilibrium distributions)
G = np.matrix([
    [0, .5, 0, .5],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]
])

# ergodic periodic (not regular), more complex than C
H = np.matrix([
    [0,.5,.5],
    [1, 0, 0],
    [1, 0, 0]
])

# ergodic and regular
J = np.matrix([
    [0.6, 0.4],
    [0.5, 0.5],
])

# all columns sum to 1 (also called a doubly sotchastic matrix)
K = np.matrix([
    [.1,.4,.3,.2],
    [.2,.1,.4,.3],
    [.3,.2,.1,.4],
    [.4,.3,.2,.1],
])

# Drunken Walk Matrix
DW = np.matrix([
    [1, 0, 0, 0, 0],
    [.5, 0, .5, 0, 0],
    [0, .5, 0, .5, 0],
    [0, 0, .5, 0, .5],
    [0, 0, 0, 0, 1],
])

# Ehrenfest Model
Ehrenfest = np.matrix([
    [0, 1, 0, 0],
    [1/3, 0, 2/3, 0],
    [0, 2/3, 0, 1/3],
    [0, 0, 1, 0],
])
```

In [3]:
```python
# Define functions to see if valid markov chain, ergodic, regular, and absorbing

def isValid(matrix):
    # Ensure sums of each row is 1
    for s in np.sum(matrix, axis=1):
        if abs(s - 1) > 1e-9:
            return False
    # Ensure non negative values and correct shape (square)
    r, c = np.shape(matrix)
    return r == c and np.all(matrix >= 0)

# dfs, use modified Tarjan's algo, if find strongly connected component and root node
def isErgodic(matrix):
    # get adjacency list
    adj_list = adjacencyMatrixToList(matrix)

    # number of nodes and adjacency list
    n = len(matrix)

    tarj_ids = [-1] * n
    tarj_counter = 0

    # initialize lowlink values
    lows = [0] * n

    assumed_ergodic = [True]

    # start a dfs on a node keeping track of lowlink value
    # currently have: node_id, tarj_id(order visited), and lowlink value for a node
    node_idx = 0

    while node_idx < n and assumed_ergodic[0]:
        if tarj_ids[node_idx] == -1: # if node unvisited
            dfs(node_idx, tarj_ids, tarj_counter, lows, adj_list, assumed_ergodic)
        node_idx += 1

    result = assumed_ergodic[0]
    return result


def dfs(node_idx, tarj_ids, tarj_counter, lows, adj_list, assumed_ergodic):
    """
    tarjan dfs

    parameters:
        node_idx (int): current_node (associted with adj)
        tarj_ids (list) : list of the order in which each node visited with first node
            assigned -1 if unvisted
            ex. if visited node 0, then 2, then 1, but 3 remains unvisited:
            tarj_ids = [0,2,1,-1]
        tarj_counter (int): keeps track of the tarj_id/visited number of the next node
        lows (list) : list of lowlink values assigned to nodes
        adj_list (dict) : adj list that stores neigbors for directed graph
        assumed_ergodic (list) : list with one bool value, kind of like base case,
            if graph found to be non ergodic, will update to False and stack will unw

    returns:
        result (bool) : boolean indicating weather matrix is ergodic or not
    """
    # visiting a node, update relevant values
    tarj_ids[node_idx] = lows[node_idx] = tarj_counter
    tarj_counter += 1

    # counter for while loop
    neighbor_pointer = 0
    num_neighbors = len(adj_list[node_idx])

    while assumed_ergodic[0] and neighbor_pointer < num_neighbors:
        # get the node_id
        neighbor_idx = adj_list[node_idx][neighbor_pointer]
        # if unvisited, recurse
        if tarj_ids[neighbor_idx] == -1:
            dfs(neighbor_idx, tarj_ids, tarj_counter, lows, adj_list, assumed_ergodic)

        # if only doing one scc, could use the ids list and see if visited
        lows[node_idx] = min(lows[node_idx], lows[neighbor_idx])

        neighbor_pointer += 1

    # Check if tarj id the same as lowlink
    # if this is true, there is a strongly connected component detected
    # if scc detected on root node other than 0, not ergodic
    if tarj_ids[node_idx] == lows[node_idx] and node_idx != 0:
        assumed_ergodic[0] = False

# creates unweighted adjacency list from a given matrix # probably can just use the ma
def adjacencyMatrixToList(matrix):
    # declare as empty dictionary
    adjacencyList = {}

    for i in range(len(matrix)):
        neighbors = [j for j in range(len(matrix)) if matrix[i,j] > 0]
        adjacencyList[i] = neighbors

    return adjacencyList

def isAbsorbing(matrix):
    return 1 in matrix.diagonal()

# note: this is an approximation
def isRegular(matrix):
    return np.isclose((matrix ** 2000).A1, (matrix ** 2001).A1, atol=1e-8).all()
```

In [4]:
```python
# Tests
assert(isValid(A))
assert(isValid(B))
assert(isValid(C))
assert(isValid(D))

assert(isErgodic(A))
assert(not isErgodic(B))
assert(isErgodic(C))
assert(not isErgodic(D))
assert(isErgodic(E))
assert(not isErgodic(F))
assert(not isErgodic(G))
assert(isErgodic(H))

assert(not isAbsorbing(A))
assert(isAbsorbing(B))
assert(not isAbsorbing(C))
assert(isAbsorbing(D))

assert(isRegular(A))
assert(isRegular(B))
assert(not isRegular(C))
assert(isRegular(D))
```

Functions to Assist in Absorbing Markov Chain Questions

In [5]:
```python
# Helper function to abstract the code, see the next two functions
def getCanonicalCorner(matrix, corner):
    indexFirstOne = np.shape(matrix)[0] - 1

    # Find the first element on the diagonal that is a one and use it as reference
    for i, elem in enumerate(np.array(matrix.diagonal())[0]):
        if elem == 1:
            indexFirstOne = i
            break

    if corner == "Q": # Top Left Corner
        return matrix[:indexFirstOne, :indexFirstOne].copy()
    elif corner == "R": # Top Right Corner
        return matrix[:indexFirstOne, indexFirstOne:].copy()
    else:
        raise ValueError("Corner should be either Q or R")

def getQ(matrix):
    """
    Return Q from a canonical matrix.

    parameters:
    - matrix: The canonical form matrix

    returns:
    Q from the canonical form matrix
    """
    return getCanonicalCorner(matrix, "Q")

def getR(matrix):
    """
    Return R from a canonical matrix.

    parameters:
    - matrix: The canonical form matrix

    returns:
    R from the canonical form matrix
    """
    return getCanonicalCorner(matrix, "R")

def getCanonicalData(matrix):
    """
    Get the canonical form of a matrix and more information.

    Parameters:
    - matrix: The matrix that we wish to transform

    Returns:
    If absorbing, returns a tuple containing
    - new matrix that is in canonical form,
    - Q,
    - R,
    - transitive states,
    - and absorbing states
    Otherwise if not absorbing, throws an error.
    """

    if not isAbsorbing(matrix):
        raise ValueError("Supplemented Matrix is Not Absorbing")

    # Determine which states are absorbing or transitive
    absorbing_states = []
    transitive_states = []
    for i in range(len(matrix)):
        if matrix.item(i, i) == 1:
            absorbing_states.append(i)
        else:
            transitive_states.append(i)

    # The order of the states of a canonical matrix will be the transitive followed by
    ordered_states =  transitive_states + absorbing_states

    # Reorder the matrix according to canonical form
    canonicalM = matrix.copy()
    for i, row in enumerate(ordered_states):
        for j, col in enumerate(ordered_states):
            canonicalM.itemset((i, j), matrix.item(row, col))

    return (canonicalM, getQ(canonicalM), getR(canonicalM), transitive_states, absorbi

def getN(matrix):
    """
    Gets the N matrix where N = (I - Q)^{-1}.

    Parameters:
    - matrix: The Q matrix

    Returns:
    The matrix N
    """
    return np.linalg.inv(np.eye(len(matrix)) - matrix)
```

In [6]:
```python
print("Original Matrix\n", D)
canonicalD, QforD, RforD, transientStates, absorbingStates = getCanonicalData(D)
print("Canonical Form Matrix\n", canonicalD)
print("Q from Canonical Form\n", QforD)
print("R from Canonical Form\n", RforD)
print("N based on Q\n", getN(QforD))
```

```
Original Matrix
 [[1.  0.  0.  0. ]
 [0.2 0.  0.6 0.2]
 [0.  0.  1.  0. ]
 [0.1 0.2 0.3 0.4]]
Canonical Form Matrix
 [[0.  0.2 0.2 0.6]
 [0.2 0.4 0.1 0.3]
 [0.  0.  1.  0. ]
 [0.  0.  0.  1. ]]
Q from Canonical Form
 [[0.  0.2]
 [0.2 0.4]]
R from Canonical Form
 [[0.2 0.6]
 [0.1 0.3]]
N based on Q
 [[1.07142857 0.35714286]
 [0.35714286 1.78571429]]
```

In [7]:
```python
print("Original Matrix\n", B)
canonicalB, QforB, RforB, transientStates, absorbingStates = getCanonicalData(B)
print("Canonical Form Matrix\n", canonicalB)
print("Q from Canonical Form\n", QforB)
print("R from Canonical Form\n", RforB)
print("N based on Q\n", getN(QforB))
```

```
Original Matrix
 [[0.1 0.4 0.5]
 [0.  1.  0. ]
 [0.1 0.8 0.1]]
Canonical Form Matrix
 [[0.1 0.5 0.4]
 [0.1 0.1 0.8]
 [0.  0.  1. ]]
Q from Canonical Form
 [[0.1 0.5]
 [0.1 0.1]]
R from Canonical Form
 [[0.4]
 [0.8]]
N based on Q
 [[1.18421053 0.65789474]
 [0.13157895 1.18421053]]
```

In [8]:
```python
# Ensure non-absorbing matrices can't be used
try:
    getCanonicalData(A)
    raise ValueError("The error was not caught")
except ValueError as e:
    print(f"Correctly caught the error: {e}")
```

```
Correctly caught the error: Supplemented Matrix is Not Absorbing
```

Functions For Answering Questions

- ✅ Probability to go from state **a** to state **b** in **k** steps
- ✅ Average number of times in a state before being absorbed
- ✅ Average number of steps before being absorbed
- ✅ Probability that we are absorbed by state **b** if we start in state **a** (be absorbed)
- ✅ Find the fixed vector of a matrix
- ✅ Mean First Passage Time of ergodic matrix
- ✅ Mean first return time for state **a**

In [9]:
```python
def probGoToState(matrix, start=None, end=None, steps=1):
    """
    Find the probability to go from some state to another in a given amount of steps.

    Parameters:
    - matrix: numpy matrix that is a Markov Chain
    - start: Row index to start in
    - end: Column index to end in
    - steps: Number of steps to take in the Markov Chain

    Returns:
    P(From start To end in given number of steps).
    """
    if (not isValid(matrix)):
        raise ValueError("Invalid Markov Chain")

    return round((matrix ** steps)[start, end], 5)

def timesInStateBeforeAbsorption(matrix, start=None, state=None):
    """
    Find the number of times you are in a state before absorption if you start in some

    Parameters:
    - matrix: numpy matrix that is an absorbing Markov Chain
    - start: State in the Markov Chain that we start in
    - state: State in the Markov Chain that we wish to see how many times we visit on

    Returns:
    P(If we start on a state, how many times are we in some given state before absorpt
    """

    _, Q, _, transient_states, _ = getCanonicalData(matrix)

    # We need to adjust for the fact that the indices of Q are different
    start = transient_states.index(start)
    state = transient_states.index(state)

    N = getN(Q)

    return round(N[start, state], 5)

def avgStepsBeforeAbsorption(matrix, start=None):
    """
    Find the number of steps until absorption.

    Parameters:
    -matrix: Absorbing Markov Chain (np matrix)
    -start: What state are we starting in?

    Returns:
    Number of steps until absorption
    """

    _, Q, _, transient_states, _ = getCanonicalData(matrix)

    # We need to adjust for the fact that the indices of Q are different
    start = transient_states.index(start)

    N = getN(Q)

    return round(np.sum(N, axis=1).item(start), 5)

def probEndInAbsorbingState(matrix, start=None, absorbing=None):
    """
    Find the probability we are absorbed by some state.

    Parameters:
    -matrix: Absorbing Markov Chain (np matrix)
    -start: What state are we starting in?
    -absorbing: What absorbing state are we ending in?

    Returns:
    Probability that we are absorbed by some given absorbing state
    """

    _, Q, R, transient_states, absorbing_states = getCanonicalData(matrix)

    N = getN(Q)
    NR = np.matmul(N, R)

    # We need to adjust for the fact that the indices of Q are different
    start = transient_states.index(start)
    absorbing = absorbing_states.index(absorbing)

    return round(NR.item(start, absorbing), 5)

def getFixedVectors(matrix):
    """
    finds all fixed vectors/equilibrium distributions for a transition matrix
    An equilibrium distribution can be thought of as an eigenvector with eigenvalue 1
    to get the correct eigenvector, we can find the vector in the E1 space with non-ne

    Parameters:
        matrix (np.matrix) : right transition/stochastic matrix

    Returns:
        fixed_vectors (np.matrix) : equilibrium vectors for the matrix
    """
    # must transpose the matrix because it is a right stochastic matrix
    evals, evecs = np.linalg.eig(matrix.copy().T)

    # get eigenvectors where eigenvalues
    # where a matrix has complex eigenvalues, all eigenvalues are represented in comp
    if evals.dtype == "complex128":
        raw_fixed = evecs[:, np.where(np.round(evals, 8) == complex(1,0))[0]]
        raw_fixed = raw_fixed.real
    else:
        raw_fixed = evecs[:, np.where(np.abs(evals - 1) < 1e-8)[0]]

    # "normalize" the vectors so that they sum to 1 (for a valid distribution),
    # transpose so in right stochastic matrix form
    fixed_vectors = (raw_fixed / raw_fixed.sum(axis=0)).T

    return fixed_vectors

def getFixedVector(matrix):
    """
    Find the fixed vector of a regular Markov Chain.

    Parameter:
    -matrix: Regular Markov Chain

    Returns:
    Array with fixed vector probabilities
    """
    if not isRegular(matrix):
        raise ValueError("Supplemented Markov Chain is Not Regular")

    return (matrix ** 2000)[0, :]

def meanFirstPassageTime(matrix, start=None, end=None):
    """
    For ergodic matrix, average time to go from one state to another for the first tim

    Parameters:
    -matrix: Ergodic Markov Chain
    -start: State we start in
    -end: State we want to end in

    Returns:
    The mean number of steps to go from the start state to the end state for the first
    """
    if not isErgodic(matrix):
        raise ValueError("Supplemented Markov Chain is not Ergodic")

    # Make a copy of the matrix where the end state is absorbing
    new_matrix = matrix.copy()
    new_matrix[end, :] = 0
    new_matrix[end, end] = 1

    _, Q, _, transient_states, absorbing_states = getCanonicalData(new_matrix)

    N = getN(Q)

    # Account for the fact the index of our start may have changed when making canonic
    start = transient_states.index(start)

    return round(np.sum(N, axis=1).item(start), 5)

def meanFirstReturnTime(matrix, state=None):
    """
    For regular matrix, average time to leave a state and come back to it.

    Parameters:
    -matrix: Regular Markov Chain
    -state: State to return to

    Returns:
    The mean number of steps to get back to the supplemented state
    """
    if not isRegular(matrix):
        raise ValueError("Supplemented Markov Chain is not Regular")

    w = getFixedVector(matrix)

    print(w)

    return round(1 / w[0, state], 5)
```

In [10]:
```python
# tests for fixed vector
assert(np.isclose(getFixedVectors(G), np.matrix([[0. , 0.5, 0.5, 0. ],
                                                 [0. , 0. , 0. , 1. ]])).all())

assert(np.isclose(getFixedVectors(C), np.matrix([0.5,0.5])).all())
assert(np.isclose(getFixedVectors(K), np.matrix([0.25,0.25,0.25,0.25])).all())
```

Interfaceable Markov Chain Calculator

User can enter Markov Chains and answer questions based on the type of Markov Chain that it is.

In [11]:
```python
current_matrix = None
ergodic = False
regular = False
absorbing = False

while True:
    # Take Matrix as Input Here
    if current_matrix is None:
        # Take as input how many states are in the matrix
        num_states = int(input("How many states are there?: "))
        # Start with the identity matrix as the template
        current_matrix = np.asmatrix(np.eye(num_states))
        # For each row and column in the matrix, take input for what the probability i
        for r in range(num_states):
            for c in range(num_states):
                # Keep trying to take input in until the user enters a valid floating
                while True:
                    try:
                        user_input = str(input(f'Probability at position {r}, {c}'))
                        if "/" in user_input:
                            user_input = list(map(lambda x: float(x) ,user_input.split
                            current_matrix.itemset((r, c), float(user_input[0] / user_
                        else:
                            current_matrix.itemset((r, c), float(user_input))
                    except Exception as e:
                        print(f"Invalid Input: {e}")
                    else:
                        break
        # If the matrix entered by the user isn't a Markov chain, make them enter a ne
        if not isValid(current_matrix):
            print("The Following Matrix is not a Valid Markov Chain:\n", current_matri
            current_matrix = None
            ergodic = False
            regular = False
            absorbing = False
            continue
        else:
            print("The Current Markov Chain Is:\n", current_matrix)
            ergodic = isErgodic(current_matrix.copy())
            regular = isRegular(current_matrix.copy())
            absorbing = isAbsorbing(current_matrix.copy())

    # List the options available for each type of Markov chain
    options = [
        ("Probability to go from state a to state b in k steps", probGoToState, ["sta
    ]

    if ergodic:
        options += [
            ("Mean First Passage Time of ergodic matrix", meanFirstPassageTime, ["sta
        ]
    if regular:
        options += [
            ("Find the fixed vector of a matrix", getFixedVector, []),
            ("Mean first return time for some state", meanFirstReturnTime, ["state"])
        ]
    if not regular:
        options += [
            ("Get all fixed vectors of a matrix", getFixedVectors, [])
        ]
    if absorbing:
        options += [
            ("Average number of times in a state before being absorbed", timesInStateB
            ("Average number of steps before being absorbed", avgStepsBeforeAbsorption
            ("Probability that we are absorbed by state b if we start in state a (be a
        ]

    user_input = ""

    # Have the user input what operation they would like to perform
    user_quit = False

    while True:
        print("Enter Operation Indicated by Value Inside Parentheses")
        for i, option in enumerate(options):
            print(f"({i}): {option[0]}")
        print("(Q) Try New Markov Chain")
        print("(Quit) Exit the Program")
        user_input = input()
        if user_input == "Q":
            current_matrix = None
            ergodic = False
            regular = False
            absorbing = False
            break
        if user_input == "Quit":
            user_quit = True
            break
        args = []
        for arg in options[int(user_input)][2]:
            args.append(int(input(f"{arg}: ")))
        func = options[int(user_input)][1]
        result = func(current_matrix, *args)
        print("Result:\n", result)

    if user_quit:
        break
```

```
The Current Markov Chain Is:
 [[0.5 0.5]
 [0.5 0.5]]
Enter Operation Indicated by Value Inside Parentheses
(0): Probability to go from state a to state b in k steps
(1): Mean First Passage Time of ergodic matrix
(2): Find the fixed vector of a matrix
(3): Mean first return time for some state
(Q) Try New Markov Chain
(Quit) Exit the Program
Result:
 [[0.5 0.5]]
Enter Operation Indicated by Value Inside Parentheses
(0): Probability to go from state a to state b in k steps
(1): Mean First Passage Time of ergodic matrix
(2): Find the fixed vector of a matrix
(3): Mean first return time for some state
(Q) Try New Markov Chain
(Quit) Exit the Program
```