Evaluating the Impacts of Road/ Highway Network Design on Population Dynamics
Final Report

Team 73
Members: Kai Qu, Ge Zhang, Zhe Zheng
Git Repository: https://github.gatech.edu/gzhang60/cse6730Project

April 2021

1 Introduction

The conflict between the natural population and human activity is always the hottest topic during urbanization. Biodiversity conservation is the key to sustainable development, especially for those highly urbanized areas. Corridors and natural reserves have been considered an effective way to protect biodiversity. However, planning the space for natural reserves and corridors is challenging for city planners. Road/ Highway network is the core of the urbanization and would significantly affect the population. Thus, to find the space for corridors and natural reserves during the urbanization, and reduce the negative impacts of road/ highway network on population development, we aim to compare the population dynamics with varied road network design(e.g. location, number of segments or shape) by the simulations.

The Diffusion-reaction system has been used as the base model for simulating population dynamics[7, 11]. The model was originally designed for chemical reactions. However, considering the main behaviors in population dynamics are dispersal and reproduction, this model is suitable for simulating population dynamics. For the diffusion-reaction system-based population dynamics model, we assume that the population density at one location is affected by natural death and birth locally and by migration from the other locations. The population's death and birth rate are affected not only by the resources or space but also by the other species.

To detect the impacts of road/ highway network development on population dynamics, we used three models: Single-species model, Predator Vs. Prey model and Species Vs. Species model to simulate the population dynamics under different scenarios[11]. The scenario simulated by the Single-species model assumes that the species is the dominant population in a small area, and they are distributed homogeneously. The limitations on their behaviors are mainly from the physical environment, such as food and spatial distance. The scenario simulated by Predator Vs. Prey model assumes that the species has predators in the habitat. The production and dispersal of the species are affected by the physical environment and the predators' activity. The scenario simulated by Species Vs. Species model assumes that some other species in the habitat share the same food and resources with the species. Thus the population density of the species would be affected by other species. Also, we assumed the road network would kill the population during the dispersal. There are two behaviors for each population, including growth and dispersal. We assumed the growth process happens before the dispersal process for each time step.

2 Data Preparation

2.1 Data engine

There are three types of population distribution patterns, including clumped, even, and random[1]. Clumped distribution is the most common population pattern in reality, which is caused mainly by patched resources. It is a general pattern for prey. Ecologists use a neutral landscape model to simulate the spatial pattern in nature. A modified random clusters method has been developed to generate neutral landscape models[10]. Based on this method, there are four steps for generating population patterns in our case:

1 Mark the population location by the initial probability p. In this step, a probability value between 0 and 1 will be given. A random number between 0 and 1 will be generated from the uniform distribution and compared with the initial probability number for each grid. If the random number is smaller than the probability value, the grid will be marked as occupied by population. Otherwise, the grid would be skipped. The maximum value of p is 0.593 if 4-neigbor rule is applied in the step 2.

[3]:
```
def initialWorld(widthLengthOfWorld):
    A = np.zeros((widthLengthOfWorld, widthLengthOfWorld))
    return A
```

```
def locatePopulationOnProb(world, randomSeed,widthLengthOfWorld, initialProb):
    A = world.copy()
    random.seed(randomSeed)

    for cellIndex, cell in np.ndenumerate(A):
        if random.random() < initialProb:
            A[cellIndex] = 1
    return A
```

2 Find the population cluster based on marked grids. In this step, a 4-neighbor rule will be applied to identify the clusters. It means, if the marked grid is found on the right, left, top, and bottom side and neighborhood of another marked grid, they belong to the same cluster.

[ ]:
```
def findClusterByNeighbor(world,widthLengthOfWorld):
    currentClusterID = 1
    A = world.copy()
    for cellIndex, cell in np.ndenumerate(world):
        if A[cellIndex[0],cellIndex[1]] == 1:
           # print(cellIndex[0],cellIndex[1])
            currentClusterID += 1
            ␣
 ↪check4Neighbor(cellIndex[0],cellIndex[1],A,widthLengthOfWorld,currentClusterID)
           # display2DArray(A)

    return A

def check4Neighbor(x,y,world,widthLengthOfWorld,currentClusterID):
    found = 0

    if world[x,y] == 1:
        world[x,y] = currentClusterID
        #check left, right, top, bottom grids if  directions are available
        if y - 1 >= 0 and world[x,y - 1] == 1:
            found = 1
            check4Neighbor(x,y - 1,world,widthLengthOfWorld,currentClusterID)

        if y + 1 <= (widthLengthOfWorld - 1) and world[x,y + 1] == 1:
            found = 1
            check4Neighbor(x,y + 1,world,widthLengthOfWorld,currentClusterID)


        if x - 1 >= 0 and world[x - 1,y] == 1:
            found = 1
            check4Neighbor(x - 1,y,world,widthLengthOfWorld,currentClusterID)

        if x + 1 <= (widthLengthOfWorld - 1) and world[x + 1,y] == 1:
            found = 1
            check4Neighbor(x + 1,y,world,widthLengthOfWorld,currentClusterID)
```

3 Assign the class number to each cluster. In this step, each cluster will be assigned the class numbers based on the predefined class system. The occupancy rate of each class will be given. The cluster will randomly

assign the class number from a candidate group, which includes all the possible class numbers. If the cluster area assigned by one class number has reached the given rate of that class, the class number will be deleted from the candidate group and thus no longer used for any other clusters.

[ ]:
```
def assignClassToCluster(areasOfClasses, clusterWorldBeforeAssign,widthLengthOfWorld):
    clusterWorld = clusterWorldBeforeAssign.copy()
    #find number of grids for each cluster
    clusterDict = {}
    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell in clusterDict:
            clusterDict[cell] += 1
        else:
            clusterDict[cell] = 1

    classDict = {}
    classCandidates= []
    # estimate the grids of each class
    for index, areaOfClass in enumerate(areasOfClasses):
        classDict[index+1] = areaOfClass * (widthLengthOfWorld * widthLengthOfWorld -␣
↪clusterDict[0])
        classCandidates.append(index+1)

    clusterAssignment={}
    # calulate the assignment based on the area matching
    for key in clusterDict:
        if key > 0:
            chosenClass = random.choice(classCandidates)
            if clusterDict[key] < classDict[chosenClass]:
                clusterAssignment[key] = chosenClass
                classDict[chosenClass] = classDict[chosenClass] - clusterDict[key]
                if classDict[chosenClass] <= 0:
                    classCandidates.remove(chosenClass)
            else:
                clusterAssignment[key] = chosenClass
                classCandidates.remove(chosenClass)

    #assign the class to cluster based on matching list

    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell in clusterAssignment:
            clusterWorld[cellIndex] = clusterAssignment[cell]
    return clusterWorld
```

4 Fill the gap between the clusters. A class number will be assigned to those unmarked grids based on the most frequent class number of 8 neighborhood grids of the unmarked grid. If two classes have the same frequency, one class will be randomly chosen from them for assignment.

[ ]:
```
def findMajorityFromList(list):
    count = {}
    for key in list:
        if key in count:
```

```
            count[key] += 1
        else:
            count[key] = 1

    initialmaxSet = 0

    for key in count:
        if initialmaxSet == 0:
            initialmax = count[key]
            maxKey = key
            initialmaxSet = 1

        if count[key] > initialmax:
            maxKey = key
    return maxKey


def pickMajorFrom8Neighbor(cellIndex,A,world,widthLengthOfWorld):
    candidates = []
    #check left, right, top, bottom grids if  directions are available
    if cellIndex[1] - 1 >= 0 and world[cellIndex[0],cellIndex[1] - 1] > 0:
        candidates.append(world[cellIndex[0],cellIndex[1] - 1])

    if cellIndex[1] + 1 <= (widthLengthOfWorld - 1) and␣
 ↪world[cellIndex[0],cellIndex[1] + 1] > 0:
        candidates.append(world[cellIndex[0],cellIndex[1] + 1])


    if cellIndex[0] - 1 >= 0 and world[cellIndex[0] - 1,cellIndex[1]] > 0:
        candidates.append(world[cellIndex[0] - 1,cellIndex[1]])


    if cellIndex[0] + 1 <= (widthLengthOfWorld - 1) and world[cellIndex[0] +␣
 ↪1,cellIndex[1]] > 0:
        candidates.append(world[cellIndex[0] + 1,cellIndex[1]])

    if cellIndex[0] + 1 <= (widthLengthOfWorld - 1) and cellIndex[1] + 1 <=␣
 ↪(widthLengthOfWorld - 1) and world[cellIndex[0] + 1,cellIndex[1] + 1] > 0:
        candidates.append(world[cellIndex[0] + 1,cellIndex[1] + 1])

    if cellIndex[0] + 1 <= (widthLengthOfWorld - 1) and cellIndex[1] - 1 >= 0 and␣
 ↪world[cellIndex[0] + 1,cellIndex[1] - 1] > 0:
        candidates.append(world[cellIndex[0] + 1,cellIndex[1] - 1])

    if cellIndex[0] - 1 >= 0 and cellIndex[1] + 1 <= (widthLengthOfWorld - 1) and␣
 ↪world[cellIndex[0] - 1,cellIndex[1] + 1] > 0:
        candidates.append(world[cellIndex[0] - 1,cellIndex[1] + 1])

    if cellIndex[0] - 1 >= 0 and cellIndex[1] - 1 >= 0 and world[cellIndex[0] -␣
 ↪1,cellIndex[1] - 1] > 0:
```

```
        candidates.append(world[cellIndex[0] - 1,cellIndex[1] - 1])

    if len(candidates)>0:
        A[cellIndex[0],cellIndex[1]] = findMajorityFromList(candidates)


def fillGapOnUnmarked(clusterWorld,widthLengthOfWorld):
    A = clusterWorld.copy()
    for cellIndex, cell in np.ndenumerate(A):
        if cell == 0:
            pickMajorFrom8Neighbor(cellIndex,A,clusterWorld,widthLengthOfWorld)
    return A
```

Random distribution is a rare population pattern, in reality, caused mainly by continuous resources. There is no interaction between individuals in the population.

[ ]:
```
def genrateRandomDistribution(world, numberOfClasses, randomSeed,widthLengthOfWorld,␣
↪initialProb):
    A = world.copy()
    random.seed(randomSeed)
    candidates = list(range(0,numberOfClasses + 1 ))
    candidates.pop(0)

    for cellIndex, cell in np.ndenumerate(A):
        if random.random() < initialProb:
            A[cellIndex] = random.choice(candidates)
    return A
```

Even distribution is a less common population pattern in reality, mainly caused by the competition for resources within the population.

[ ]:
```
def genrateEvenDistribution(world, numberOfClasses, gapSpace, widthLengthOfWorld):
    A = world.copy()
    candidates = list(range(0,numberOfClasses + 1 ))
    candidates.pop(0)

    for cellIndex, cell in np.ndenumerate(A):
        if cellIndex[0] != 0 and cellIndex[1] != 0 and cellIndex[0] % gapSpace == 0␣
↪and cellIndex[1] % gapSpace == 0:
            A[cellIndex] = random.choice(candidates)
    return A
```

Since the project aims to evaluate the impacts of road network development on population dynamics, we developed the functions for generating road networks with different configurations. We assume that the road would start from the upper bound of the world and be extended based on the probability of three directions: west, east, and south. The probability of one cell is the end of the road, and the extension probability for three directions would be given.

[ ]:
```
def addHighway2World(clusterWorld, widthLengthOfWorld, probAsEndOfRoad,␣
↪probForDirection, highwayClass,randomSeed):
    random.seed(randomSeed)
```

```
    A = clusterWorld.copy()
    endOfHighway = (0,int(widthLengthOfWorld * probAsEndOfRoad))
    findNextForHighway(A,endOfHighway,probForDirection,highwayClass,␣
 ↪widthLengthOfWorld)
    return A

def findNextForHighway(A,cellIndex,probForDirection,highwayClass,widthLengthOfWorld):
    candidatesProb = []
    candidatesCoordinates = []
    #check left, right, bottom grids if  directions are available
    if cellIndex[1] - 1 >= 0 and A[cellIndex[0], cellIndex[1] - 1] < highwayClass:
        candidatesProb.append(probForDirection[0])
        candidatesCoordinates.append((cellIndex[0], cellIndex[1] - 1))

    if cellIndex[1] + 1 <= (widthLengthOfWorld - 1) and A[cellIndex[0], cellIndex[1] +␣
 ↪1] < highwayClass:
        candidatesProb.append(probForDirection[1])
        candidatesCoordinates.append((cellIndex[0], cellIndex[1] + 1))

    if cellIndex[0] + 1 <= (widthLengthOfWorld - 1) and A[cellIndex[0] + 1,␣
 ↪cellIndex[1]] < highwayClass:
        candidatesProb.append(probForDirection[2])
        candidatesCoordinates.append((cellIndex[0] + 1, cellIndex[1]))

    if len(candidatesProb)>0 and cellIndex[0] + 1 <= (widthLengthOfWorld - 1):
        recalProbForCandidates(candidatesProb)

        nextCoordinates = findNextLocation(candidatesProb,candidatesCoordinates)

        A[nextCoordinates] = highwayClass
        if (nextCoordinates[1] - 1) !=0 and (nextCoordinates[1] + 1) !=␣
 ↪(widthLengthOfWorld - 1) and (nextCoordinates[0] + 1) != (widthLengthOfWorld - 1):
            findNextForHighway(A,nextCoordinates,probForDirection,highwayClass,␣
 ↪widthLengthOfWorld)

def recalProbForCandidates(candidates):
    sum = 0
    for key in candidates:
        sum += key

    for index, key in enumerate(candidates):
        candidates[index] = candidates[index]/sum

def findNextLocation(candidates,candidatesCoordinates):
#
    # sort the probs
    candidatesCoordinatesS = [x for _,x in␣
 ↪sorted(zip(candidates,candidatesCoordinates))]
    candidates.sort()
```

```
    randomeValue = random.random()

    for index, key in enumerate(candidates):
        if randomeValue < key :
            return candidatesCoordinatesS[index]
    return candidatesCoordinatesS[len(candidatesCoordinatesS) - 1]
```

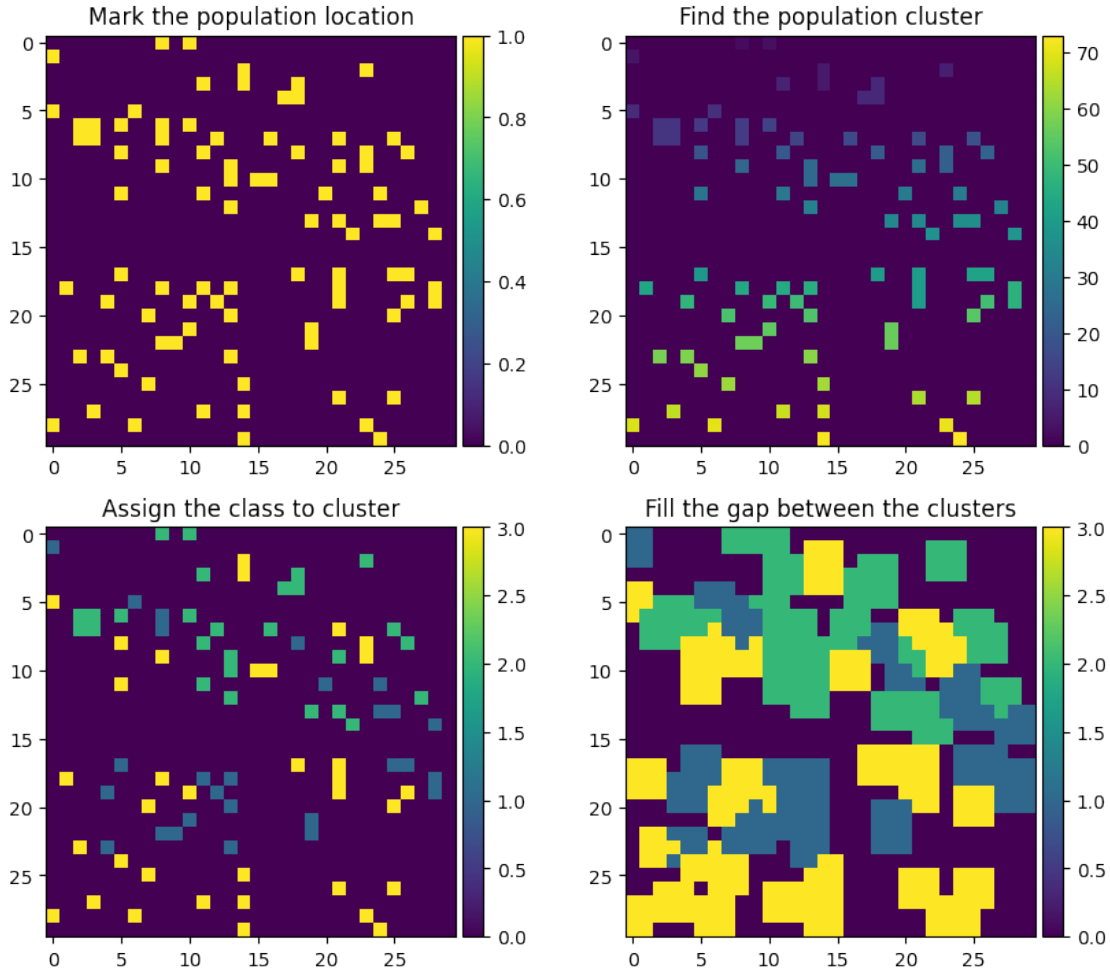
To better test the code for data generation, we developed a visualization component for checking the results visually.

[ ]:
```
def initial2DArrayForGeneratedPattern():
    fig = plt.figure(figsize=(10,10), dpi=100)
    fig.subplots_adjust(top=1.5)

    ax1 = fig.add_subplot(4, 2, 1)
    ax1.title.set_text('Mark the population location')
    ax2 = fig.add_subplot(4, 2, 2)
    ax2.title.set_text('Find the population cluster')
    ax3 = fig.add_subplot(4, 2, 3)
    ax3.title.set_text('Assign the class to cluster')
    ax4 = fig.add_subplot(4, 2, 4)
    ax4.title.set_text('Fill the gap between the clusters')

    return fig,ax1,ax2,ax3,ax4

def update2DArrayForGeneratedPattern(fig,ax,A):
    im1 = ax.imshow(A, interpolation='none')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im1, cax=cax, orientation='vertical')

def updateCharts(fig):
    display(fig)
    clear_output(wait = True)
    plt.pause(0.2)

def displayPattern(A,title):
    plt.rcParams["figure.figsize"] = (60,10)
    plt.imshow(A, interpolation='none')
    plt.colorbar()
    plt.title(title)
    plt.show()
```

2.2 Verification and visualization

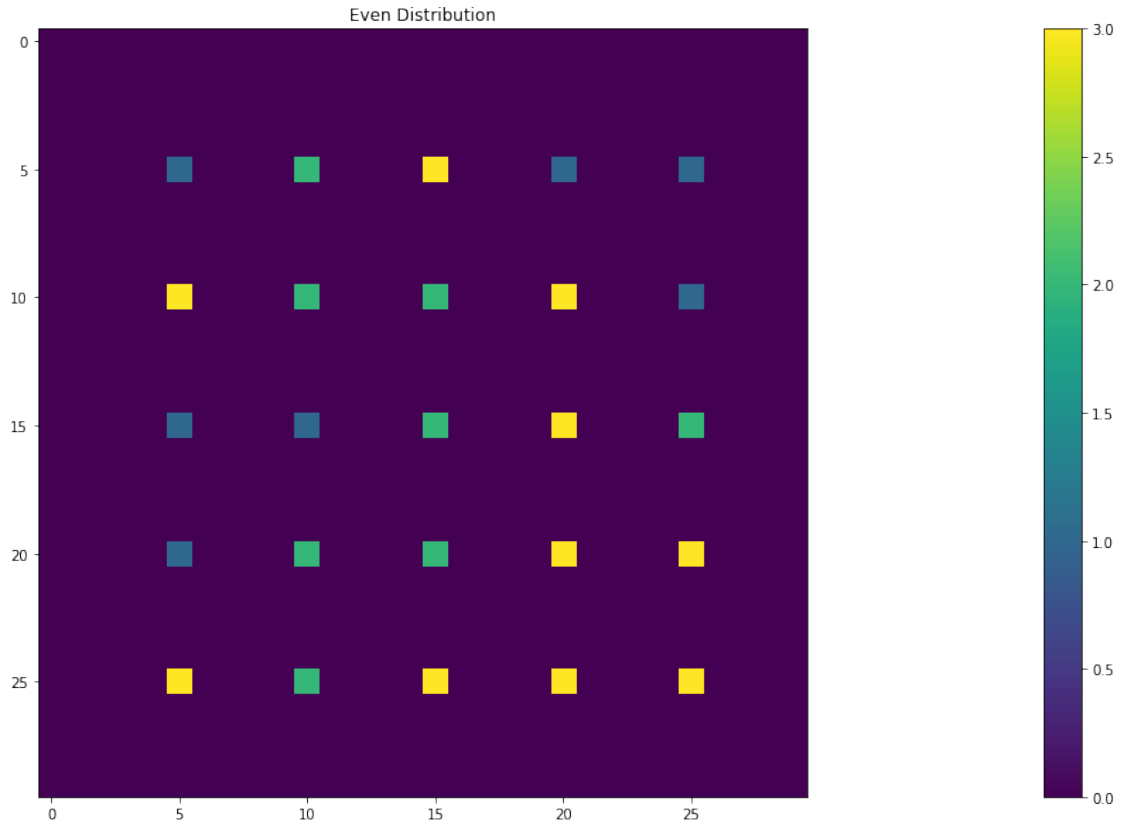
A clumped pattern was generated by the number of classes as 3, the percentage area of class as 30%, 30%, and 40%, respectively, occupation probability as 0.1, and the dimension of the world as 30. The procedure for generating clumped pattern was verified step by step with a 2D map.

[8]:
```
numberOfClasses = 3
areasOfClasses = [0.3,0.3,0.4]
widthLengthOfWorld = 30
initialProb = 0.1
randomSeedForPopulation = 100

I = dataForInput.initialWorld(widthLengthOfWorld)
C = dataForInput.locatePopulationOnProb(I,␣
 ↪randomSeedForPopulation,widthLengthOfWorld,initialProb)
D = dataForInput.findClusterByNeighbor(C,widthLengthOfWorld)
clusterWorld = dataForInput.assignClassToCluster(areasOfClasses, D, widthLengthOfWorld)
F = dataForInput.fillGapOnUnmarked(clusterWorld,widthLengthOfWorld)

fig,ax1,ax2,ax3,ax4 = visualization.initial2DArrayForGeneratedPattern()
visualization.update2DArrayForGeneratedPattern(fig,ax1,C)
visualization.update2DArrayForGeneratedPattern(fig,ax2,D)
visualization.update2DArrayForGeneratedPattern(fig,ax3,clusterWorld)
visualization.update2DArrayForGeneratedPattern(fig,ax4,F)
visualization.updateCharts(fig)
```



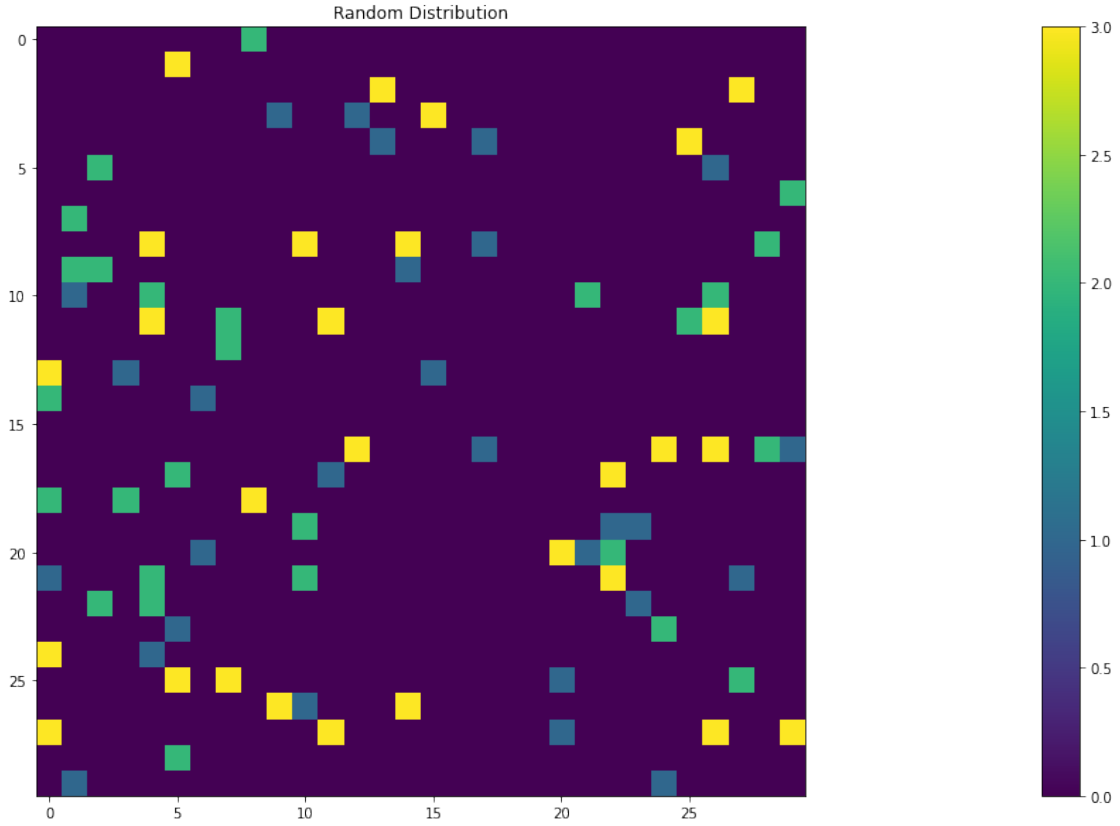
A even pattern was generated by the number of classes as 3, distance between population as 5, and the dimension of the world as 30. The procedure for generating even pattern was verified by a 2D map.

[28]:
```
gapSpace = 5
B = dataForInput.genrateEvenDistribution(I,␣
 ↪numberOfClasses,gapSpace,widthLengthOfWorld)
visualization.displayPattern(B,"Even Distribution")
```



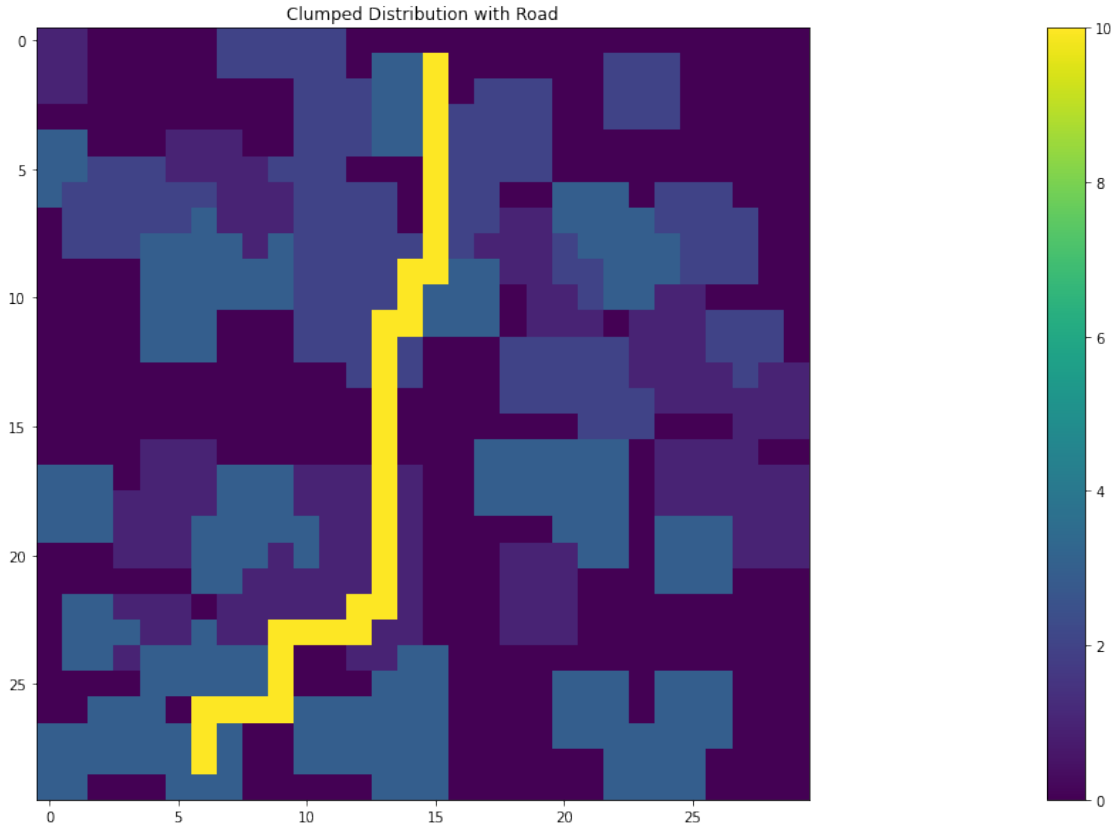
A random pattern was generated by the number of classes as 3, occupation probability as 0.1, and the dimension of the world as 30. The procedure for generating random pattern was verified by a 2D map.

[11]:
```
A = dataForInput.genrateRandomDistribution(I,␣
 ↪numberOfClasses,randomSeedForPopulation,widthLengthOfWorld,initialProb)
visualization.displayPattern(A,"Random Distribution")
```



A road/ highway was added to the clumped pattern by the probability of being the end of the road as 0.5, extension probability for the west, right, and south as 0.2,0.2, and 0.6, respectively, and the class of road as 10. A 2D map verified the procedure for generating the road.

[13]:
```
randomSeedForHighway = 20
probAsEndOfRoad = 0.5
probForDirection = [0.2,0.2,0.6]
highwayClass = 10

G = dataForInput.addHighway2World(F,widthLengthOfWorld, probAsEndOfRoad,␣
 ↪probForDirection,highwayClass,randomSeedForHighway)
visualization.displayPattern(G,"Clumped Distribution with Road")
```



2.3 Validation

Clark-Evans criterion has been used to measure the degree of the pattern's clustering[3]. If the criterion is equal to 1, it means the distribution is random. If the criterion is significantly larger than 1, it means the pattern is evenly distributed. If the criterion is smaller than 1, it means the distribution is clustered. The criterion is calculated as follows:

$$r_A = \frac{\sum r}{n}$$

$$r_E = \frac{1}{2\sqrt{\frac{n}{S}}}$$

$$R = \frac{r_A}{r_E}$$

where r is the nearest neighor distance of a marked grid, n is the total number of marked grid, S is the area of the whole world.

[ ]:
```
def measureDistanceToNearestMarked(locationListForMarked,cellIndex):
    distanceList = []
    #find all distances from marked to current grid
    for location in locationListForMarked:
        distanceList.append(caldistance(cellIndex, location))
```

```
    distanceList.remove(0)

    return min(distanceList)

def calNearestDistForGrids(clusterWorld,widthLengthOfWorld,markedClass):
    locationListForMarked = []
    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell == markedClass:
            locationListForMarked.append(cellIndex)

    A = clusterWorld.copy()
    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell == markedClass:
            currentDistance =␣
 ↪measureDistanceToNearestMarked(locationListForMarked,cellIndex)

            A[cellIndex[0],cellIndex[1]] = currentDistance
    return A

def caldistance(locationX, locationY):
    return math.sqrt((locationX[0]-locationY[0])**2 + (locationX[1]-locationY[1])**2)

def calRA(clusterWorld,distanceWorld, markedClass):
    distanceSum = 0
    count = 0
    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell == markedClass:
            distanceSum += distanceWorld[cellIndex]
            count += 1
    RA = distanceSum/count
    return RA

def calRE(clusterWorld,widthLengthOfWorld,markedClass):
    count = 0
    for cellIndex, cell in np.ndenumerate(clusterWorld):
        if cell == markedClass:
            count += 1
    RE = 1/(math.sqrt(count/(widthLengthOfWorld*widthLengthOfWorld))*2)
    return RE

def calR(RA,RE):
    return RA/RE

def calROnWorld(F,widthLengthOfWorld,markedClass):
    G = calNearestDistForGrids(F,widthLengthOfWorld,markedClass)
    RA = calRA(F,G, markedClass)
    RE = calRE(F,widthLengthOfWorld,markedClass)
    R = calR(RA,RE)
    return R
```

The R-value was calculated for three generated patterns. The result shows that the R-value of the random

sample is larger than 1. The R-value of the even sample is approximately equal to 1. The R-value of the clumped sample is smaller than 1.

[31]:
```
markedClass = 1
R = dataForInput.calROnWorld(A,widthLengthOfWorld,markedClass)
print('Random Sample:',R)
R = dataForInput.calROnWorld(B,widthLengthOfWorld,markedClass)
print('Even Sample:',R)
R = dataForInput.calROnWorld(F,widthLengthOfWorld,markedClass)
print('Clumped Sample:',R)
```

```
Random Sample: 1.1954019290238236
Even Sample: 1.0079052613579393
Clumped Sample: 0.8295367950698638
```

3 Single-species Model and Simulation

3.1 Conceptual Model

The main behaviors of the population include growth and dispersal. Based on the diffusion-reaction system, the population dynamics of one location can be described as:

$$\frac{d_N}{d_t} = D\Delta(N) + G(N)$$

where D is the dispersal speed, $\Delta(N)$ is the mathematical form of the dispersal, and G(N) is the function for growth. We assume that the population in one location would dispersal after reproduction. Based on the mathematical form of the population behaviors, we developed a population entity with location, birth rate, natural death rate, death rate during migration, dispersal speed, carrying capacity, and the dimension of the world as input. The initial population density at each cell is a randomly generated number following normal distribution and below K since K is the upper bound of the population density and the distribution of most biological data can be seen as normal distribution[5].

[33]:
```
class Population:
    def __init__(self,RC,BR,DRN,DRM,DS,K,WP,L):
        #property variables
        self.location = RC
        self.birthRate = BR
        self.deathRateNatural = DRN
        self.deathRateMigration = DRM
        self.dispersalSpeed = DS
        self.carryingCapacity = K
        #state variables
        self.growthComp = 0
        self.growthRate = 0
        self.dispersalComp = 0
        self.populationDensity = 0
        self.worldWithPopulation = WP
        self.widthLengthOfWorld = L
        self.immigtationComp = 0
        self.emigrationComp = 0
```

```
    def calGrowthComponent(self):
        growthComp = Growth(self.populationDensity, self.birthRate, self.
↪deathRateNatural, self.carryingCapacity)
        growthComp.calGrowthRate()
        growthComp.calChangedPopulationDensity()
        self.growthRate = growthComp.growthRate
        self.growthComp = growthComp.changedPopulationDensity

    def calDispersalComponent(self):

        dispersalComp = Dispersal(self.deathRateMigration, self.dispersalSpeed, self.
↪populationDensity, self.worldWithPopulation, self.location, self.widthLengthOfWorld)
        dispersalComp.calChangedPopulationDensity()
        self.dispersalComp = dispersalComp.changedPopulationDensity
        self.immigtationComp = dispersalComp.immigtationPopulation
        self.emigrationComp = dispersalComp.emigrationPopulation

    def updatePopulationDensity(self):
        self.populationDensity = self.populationDensity + self.growthComp + self.
↪dispersalComp

        #normal distribution
    def initialPopulationDensityByNormalDistribution(self):
        randomNumber = np.random.normal(0,1)
        probRandomNumber = scipy.stats.norm.cdf(randomNumber, loc=0, scale=1)
        self.populationDensity = probRandomNumber * self.carryingCapacity
```

Growth Component The growth dynamic at location i can be described as:

$$G(N) = N_i f(N_i)$$

where N_i is the population density at location i and $f(N_i)$ is the population's growth rate at location i. We assume that the location's population density at time t+1 is decided by the population density of the location at time t, birth rate, death rate, and individuals from other locations[6]. Then, the growth of the population can be described as:

$$f(N_i) = b - d_p$$

where N_t is the population density at time t, b is the birth rate of the population, and d_p is the death rate of the population within the location
We also assume that the location's population density at time t+1 is also affected by the available resource or food in the location. Then the growth of the population can be described as:

$$f(N_i) = (b - d_p)(1 - \frac{N_t}{K})$$

$$N_t \leq K$$

where K is the carrying capacity of the habitat, which is also the upper bound of the population density. Thus, the change of population density contributed by local growth can be described as:

$$\frac{d_N}{d_t} = N_t(b - d_p)(1 - \frac{N_t}{K})$$

Based on the mathematical form of the growth, we developed a growth entity with population density, birth rate, natural death rate, and carrying capacity as input.

[97]:
```
class Growth:
    def __init__(self, PD, BR, DRN, K):
        self.populationDensity = PD
        self.birthRate = BR
        self.deathRateNatural = DRN
        self.carryingCapacity = K
        self.growthRate = 0
        self.changedPopulationDensity = 0

    def calChangedPopulationDensity(self):
        self.changedPopulationDensity = self.populationDensity * self.growthRate

    def calGrowthRate(self):
        self.growthRate = (self.birthRate - self.deathRateNatural) * (1 - self.
↪populationDensity/self.carryingCapacity)
```

Dispersal Component There are two processes in population dispersal, including immigration and emigration. Immigration represents the population flow from the neighborhood locations to the current location, while emigration represents the population flows from the current location to the neighborhood locations. We assume that the population in one location would only move to north, south, east, and west by one step for dispersal at every generation.
The immigration process at location i can be described as[2]:

$$\Delta(N)_{im} = \int_a K_{j,i} N_j d_m d(j)$$

The emigraion process at location i can be described as:

$$\Delta(N)_{em} = \int_a K_{i,j} N_i d(j)$$

where $K_{j,i}$ is the probablity of population moving from location j to location i. a is the set of all neighborhood locations around location i, location j is one of the neighborhood locations of location i and it belongs to set a and d_m is the ratio of the died individuals during the migration.We assume that the population would move from the location with high population density to the location with low population density and number of immigrants is proportaional to the population density of the target location[4]. $K_{j,i}$ can be described as:

$$K_{i,j} = max\left\{f(N_j) - f(N_i), 0\right\} \frac{N_j}{N_i + N_j}$$

For technical reason to make sure there is a migration from location with popultion to location without population, N_i will be set as 10 if $N_i = 0$. Thus, the change of the population dynamics contributed by dispersal can be described as:

$$\frac{d_N}{d_t} = D(-\int_a max\left\{f(N_j) - f(N_i), 0\right\} \frac{N_j}{N_i + N_j} N_i d(j) + \int_a max\left\{f(N_i) - f(N_j), 0\right\} \frac{N_i}{N_i + N_j} N_j (1 - d_m) d(j))$$

where N_i is the population density at location i, $f(N_i)$ is the population growth rate at location i.
Thus, the population density at time t+1 and location i can be described as :

$$N_{i,t+1} = N_{i,t} + N_{i,t}(b - d_p)(1 - \frac{N_{i,t}}{K}) - D \int_a max\left\{f(N_{j,t}) - f(N_{i,t}), 0\right\} \frac{N_{j,t}}{N_{i,t} + N_{j,t}} N_{i,t} d(j) +$$

$$D \int_a max \left\{ f(N_{i,t}) - f(N_{j,t}), 0 \right\} \frac{N_{i,t}}{N_{i,t} + N_{j,t}} N_{j,t}(1 - d_m) d(j)$$

Based on the mathematical form of the dispersal, we developed a dispersal entity with death rate during migration, dispersal speed, population density, world with population entities, locaion and the dimension of the world as input.

[35]:
```
class Dispersal:
    def __init__(self,DRM,DS,PD,WP,RC,L):
        self.deathRateMigration = DRM
        self.dispersalSpeed = DS
        self.populationDensity = PD
        self.worldWithPopulation = WP
        self.location = RC
        self.widthLengthOfWorld = L
        self.changedPopulationDensity = 0
        self.immigtationPopulation = 0
        self.emigrationPopulation = 0

    def calChangedPopulationDensity(self):
        self.changedPopulationDensity = self.dispersalSpeed * (self.immigration() -␣
↪self.emigration())
        self.immigtationPopulation = self.dispersalSpeed * self.immigration()
        self.emigrationPopulation = self.dispersalSpeed * self.emigration()

    def immigration(self):
        immigrationAmount = 0

        if self.location[1] - 1 >= 0:
            immigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1] - 1].populationDensity * self.calMigrationProb((self.location[0], self.
↪location[1] - 1), self.location) * (1- self.deathRateMigration)

        if self.location[1] + 1 <= (self.widthLengthOfWorld - 1) :

            immigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1] + 1].populationDensity * self.calMigrationProb((self.location[0], self.
↪location[1] + 1), self.location) * (1- self.deathRateMigration)

        if self.location[0] - 1 >= 0 :
            immigrationAmount += self.worldWithPopulation[self.location[0] - 1][self.
↪location[1]].populationDensity * self.calMigrationProb((self.location[0] - 1, self.
↪location[1]), self.location) * (1- self.deathRateMigration)

        if self.location[0] + 1 <= (self.widthLengthOfWorld - 1) :
            immigrationAmount += self.worldWithPopulation[self.location[0] + 1][self.
↪location[1]].populationDensity * self.calMigrationProb((self.location[0] + 1, self.
↪location[1]), self.location) * (1- self.deathRateMigration)

        return immigrationAmount

    def emigration(self):
```

```
        emigrationAmount = 0

        if self.location[1] - 1 >= 0:
            emigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1]].populationDensity * self.calMigrationProb(self.location,(self.
↪location[0],self.location[1] - 1))

        if self.location[1] + 1 <= (self.widthLengthOfWorld - 1) :
            emigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1]].populationDensity * self.calMigrationProb(self.location,(self.
↪location[0],self.location[1] + 1))

        if self.location[0] - 1 >= 0 :
            emigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1]].populationDensity * self.calMigrationProb(self.location,(self.
↪location[0] - 1,self.location[1]))

        if self.location[0] + 1 <= (self.widthLengthOfWorld - 1) :
            emigrationAmount += self.worldWithPopulation[self.location[0]][self.
↪location[1]].populationDensity * self.calMigrationProb(self.location,(self.
↪location[0] + 1,self.location[1]))

        return emigrationAmount

    def calMigrationProb(self, locationI, locationJ):

        if self.calGrowthRate(locationJ) - self.calGrowthRate(locationI) > 0:
            if self.worldWithPopulation[locationJ[0]][locationJ[1]].populationDensity␣
↪== 0:
                return 10/(10 + self.worldWithPopulation[locationI[0]][locationI[1]].
↪populationDensity)
            else:
                return self.worldWithPopulation[locationJ[0]][locationJ[1]].
↪populationDensity/(self.worldWithPopulation[locationJ[0]][locationJ[1]].
↪populationDensity + self.worldWithPopulation[locationI[0]][locationI[1]].
↪populationDensity)
        else:
            return 0

    def calGrowthRate(self,location):
        return self.worldWithPopulation[location[0]][location[1]].growthRate
```

3.2 Simulation Application

A world entity was developed based on the functions developed in the data generation section. Global variables for world (Constant) includes:

Variable	Definition
p	Initial probability of the grid occupied by population
L	Length/width of the world
A_s	Percentage of area occupied by species s
T	Number of population types in the world
PM	Pattern mode of the population
GS	Space between populaiton habitats
HC	Class of road

[36]:
```
class World:
    def __init__(self, P, T, L, A, S, PM, GS, HC):
        self.initialProbOccupiedByPopulation = P
        self.numberOfPopulationTypes = T
        self.percentageOfPopulationTypes = A
        self.widthLengthOfWorld = L
        self.initialWorld = self.generateInitialWorld()
        self.randomSeed = S
        self.patternMode = PM
        self.gapSpace = GS
        self.pattern = self.generatePatternByMode()

        self.highwayClass = HC
        self.patternWithHighway = self.generatePatternByMode()

    def generateInitialWorld(self):
        return dataForInput.initialWorld(self.widthLengthOfWorld)

    def generateRandomPattern(self):
        return dataForInput.genrateRandomDistribution(self.initialWorld, self.
↪numberOfPopulationTypes, self.randomSeed,self.widthLengthOfWorld, self.
↪initialProbOccupiedByPopulation)

    def generateEvenPattern(self):
        return dataForInput.genrateEvenDistribution(self.initialWorld, self.
↪numberOfPopulationTypes, self.gapSpace, self.widthLengthOfWorld)

    def generateClumpedPattern(self):
        C = dataForInput.locatePopulationOnProb(self.initialWorld, self.
↪randomSeed,self.widthLengthOfWorld,self.initialProbOccupiedByPopulation)
        D = dataForInput.findClusterByNeighbor(C,self.widthLengthOfWorld)
        clusterWorld = dataForInput.assignClassToCluster(self.
↪percentageOfPopulationTypes, D, self.widthLengthOfWorld)
        return dataForInput.fillGapOnUnmarked(clusterWorld,self.widthLengthOfWorld)

    def generatePatternByMode(self):
        if self.patternMode == "random":
            return self.generateRandomPattern()
        elif self.patternMode == "even":
            return self.generateEvenPattern()
```

```
        elif self.patternMode == "clump":
            return self.generateClumpedPattern()

    def␣
↪generatePatternWithHghway(self,probAsEndOfRoad,probForDirection,randomSeedHighway):
        G = dataForInput.addHighway2World(self.patternWithHighway,self.
↪widthLengthOfWorld, probAsEndOfRoad, probForDirection,self.
↪highwayClass,randomSeedHighway)
        self.patternWithHighway = G
```

A simulation entity was developed to seed, run, and monitor the simulation, and handle the simulation verification and validation. Global variables for simulation (Constant) includes:

Variable	Definition
BR	Birth rate of the population
DRN	Death rate of the population staying in the same grid compared to the last generation
K	Carrying capacity of each cell
DRM	Death rate of the population during the migration
NG	Number of generations during the simulation
DS	Dispersal speed of the population

State variables includes:

Variable	Definition
populationDensity	Population density at cell i of the current generation
worldWithPopulation_ PlusMinus	changed population at cell i of the current generation
growthComp	Natural population growth at cell i of the current generation
emigrationComp	Emigrated population at cell i of the current generation
immigtationComp	Immigrated population at cell i of the current generation
killedByHighway	Accumulated population size killded by road
GrowthPopulationSizeArray	Natural population growth at step i
totalPopulationSizeArray	Total population at step i
EmigrationPopulationSizeArray	Emigrated population at step i

The simulation was designed as a time-stepped simulation. During the simulation, a world entity was initialized first as a 2D array of cells. Then, one population entity would be assigned to each cell. Each population entity has a growth component and dispersal component. For each step, the population grows first and then does dispersal. The previous step's population status determined the growth and dispersion at the current step, and the growth and dispersal entities defined the activities. We assume that road would kill the population during the simulation.

[37]:
```
class Simulation:
    def __init__(self,W , SB, HC, PT, BR, DRN, DRM, DS, K, NG, World):
        self.widthLengthOfWorld = W
        self.randomSeedForBreeding = SB
        self.highwayClass = HC
        self.populationType = PT
```

```
        self.birthRate = BR
        self.deathRateNatural = DRN
        self.deathRateMigration = DRM
        self.dispersalSpeed = DS
        self.carryingCapacity = K
        self.numberOfGenrations = NG
        self.worldWithoutPopulation = World
        #state
        self.step = 0
        self.worldWithPopulation = 0
        self.killedByHighway = 0
        self.worldWithPopulationPlusMinus = 0
        self.stepArray = []
        self.totalPopulationSizeArray = []
        self.GrowthPopulationSizeArray = []
        self.EmigrationPopulationSizeArray = []



    def breedPopulationOnWorld(self):
        worldWithPopulation = []
        worldWithPopulationPlusMinus = []

        for number in range(0,self.widthLengthOfWorld):
            worldWithPopulation.append([])
            worldWithPopulationPlusMinus.append([])

        random.seed(self.randomSeedForBreeding)
        for cellIndex, cell in np.ndenumerate(self.worldWithoutPopulation):
            if cell == self.populationType:

                population = Population(cellIndex,self.birthRate,self.
↪deathRateNatural,self.deathRateMigration,
                                        self.dispersalSpeed,self.
↪carryingCapacity,worldWithPopulation,
                                        self.widthLengthOfWorld)
                population.initialPopulationDensityByNormalDistribution()
                worldWithPopulation[cellIndex[0]].append(population)
                worldWithPopulationPlusMinus[cellIndex[0]].append(1)
            else:
                worldWithPopulation[cellIndex[0]].append(Population(cellIndex,self.
↪birthRate,self.deathRateNatural,self.deathRateMigration,
                                                                    self.dispersalSpeed,self.
↪carryingCapacity,worldWithPopulation,self.widthLengthOfWorld))
                worldWithPopulationPlusMinus[cellIndex[0]].append(0)

        self.worldWithPopulation = np.array(worldWithPopulation)
        self.worldWithPopulationPlusMinus = np.array(worldWithPopulationPlusMinus)
```

```
    def extractPopulationDensityFromPopulation(self):
        A = dataForInput.initialWorld(self.widthLengthOfWorld)
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A[cellIndex] = cell.populationDensity
        return A

    def extractImmigPopulationFromPopulation(self):
        A = dataForInput.initialWorld(self.widthLengthOfWorld)
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A[cellIndex] = cell.immigtationComp
        return A

    def extractEmigPopulationFromPopulation(self):
        A = dataForInput.initialWorld(self.widthLengthOfWorld)
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A[cellIndex] = cell.emigrationComp
        return A

    def extractGrowthFromPopulation(self):
        A = dataForInput.initialWorld(self.widthLengthOfWorld)
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A[cellIndex] = cell.growthComp
        return A

    def calTotalPopulationSize(self):
        A = 0
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A += cell.populationDensity
        return A

    def calTotalEmigPopulationSize(self):
        A = 0
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A += cell.emigrationComp
        return A

    def calTotalGrowthPopulationSize(self):
        A = 0
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            A += cell.growthComp
        return A

    def extractOccupiedFromPopulation(self):
        A = dataForInput.initialWorld(self.widthLengthOfWorld)
        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            if cell.populationDensity > 0:
                A[cellIndex] = 1
        return A

    def calPopulationDensityForEachPopulationBystep(self):
        tempWorldWithPopulation = self.worldWithPopulation.copy()
```

```
        tempWorldWithPopulation2 = self.worldWithPopulationPlusMinus.copy()

        for cellIndex, cell in np.ndenumerate(self.worldWithPopulation):
            prePopDensity = tempWorldWithPopulation[cellIndex].populationDensity

            tempWorldWithPopulation[cellIndex].calGrowthComponent()
            tempWorldWithPopulation[cellIndex].calDispersalComponent()
            tempWorldWithPopulation[cellIndex].updatePopulationDensity()
            # update plusminus

            currentPopDensity = tempWorldWithPopulation[cellIndex].populationDensity

            tempWorldWithPopulation2[cellIndex] = (currentPopDensity - prePopDensity)

        self.worldWithPopulation = tempWorldWithPopulation
        self.worldWithPopulationPlusMinus = tempWorldWithPopulation2

    def calPopDensityForHighway(self):
        tempWorldWithPopulation = self.worldWithPopulation.copy()

        for cellIndex, cell in np.ndenumerate(self.worldWithoutPopulation):
            if cell == self.highwayClass:
                self.killedByHighway += tempWorldWithPopulation[cellIndex].
↪populationDensity
                tempWorldWithPopulation[cellIndex].populationDensity = 0

        self.worldWithPopulation = tempWorldWithPopulation


    def run(self):

        self.breedPopulationOnWorld()

        self.step = 0
        while self.step < self.numberOfGenrations:
            self.step += 1
            self.calPopulationDensityForEachPopulationBystep()

            self.stepArray.append(self.step)
            self.totalPopulationSizeArray.append(self.calTotalPopulationSize())
            self.GrowthPopulationSizeArray.append(self.calTotalGrowthPopulationSize())
            self.EmigrationPopulationSizeArray.append(self.
↪calTotalEmigPopulationSize())


    def runWithHighway(self):
        self.breedPopulationOnWorld()

        self.step = 0
        fig,ax1,ax2,ax3,ax4,ax5,ax6,ax7,ax8,xdata,x2data,x3data,ydata,y2data,y3data =␣
↪visualization.initialobserver()
```

```
        while self.step < self.numberOfGenrations:
            self.step += 1
            self.calPopulationDensityForEachPopulationBystep()
            self.calPopDensityForHighway()

            visualization.updateDisplayPopulationDensity(fig,ax1, self.
↪extractPopulationDensityFromPopulation())
            visualization.updateDisplayPopulationDensityPlusMinus(fig, ax2, self.
↪worldWithPopulationPlusMinus)
            visualization.updateDisplayPopulationChanged(fig, ax3, self.
↪extractGrowthFromPopulation())
            visualization.updateDisplayPopulationChanged(fig, ax4, self.
↪extractEmigPopulationFromPopulation())
            visualization.updateDisplayPopulationChanged(fig, ax5, self.
↪extractImmigPopulationFromPopulation())


            visualization.updateLineChart(ax6,xdata,ydata,self.step,self.
↪calTotalPopulationSize(),'Population Size by Generation')
            visualization.updateLineChart(ax7,x2data,y2data,self.step,self.
↪killedByHighway,'Population Killed By Highway by Generation')
            visualization.updateLineChart(ax8,x3data,y3data,self.step,self.
↪calTotalEmigPopulationSize(),'Emigration Population Size by Generation')

            visualization.updateCharts(fig)

    def verifyPopulationPlusMinus(self):
        self.breedPopulationOnWorld()

        self.step = 0
        fig = visualization.initialFigWithoutCharts()
        while self.step < self.numberOfGenrations:
            self.step += 1
            self.calPopulationDensityForEachPopulationBystep()
            self.calPopDensityForHighway()

            ax = visualization.addChartToFig(fig,'Generation '+ str(self.
↪step),5,2,self.step)
            visualization.updateFigWithaddedChart(fig,ax,self.
↪worldWithPopulationPlusMinus,-20,50)

            visualization.updateCharts(fig)

    def verifyPopulationDensity(self):
        self.breedPopulationOnWorld()

        self.step = 0
        fig = visualization.initialFigWithoutCharts()
        while self.step < self.numberOfGenrations:
            self.step += 1
```

```
            self.calPopulationDensityForEachPopulationBystep()
            self.calPopDensityForHighway()

            ax = visualization.addChartToFig(fig,'Generation '+ str(self.
↪step),5,2,self.step)
            visualization.updateFigWithaddedChart(fig,ax,self.
↪extractPopulationDensityFromPopulation(),0,100)

            visualization.updateCharts(fig)

    def verifyEmigPopulation(self):
        self.breedPopulationOnWorld()

        self.step = 0
        fig = visualization.initialFigWithoutCharts()
        while self.step < self.numberOfGenrations:
            self.step += 1
            self.calPopulationDensityForEachPopulationBystep()
            self.calPopDensityForHighway()

            ax = visualization.addChartToFig(fig,'Generation '+ str(self.
↪step),5,2,self.step)
            visualization.updateFigWithaddedChart(fig,ax,self.
↪extractEmigPopulationFromPopulation(),0,50)

            visualization.updateCharts(fig)
```

3.3 Run Simulation and Verification

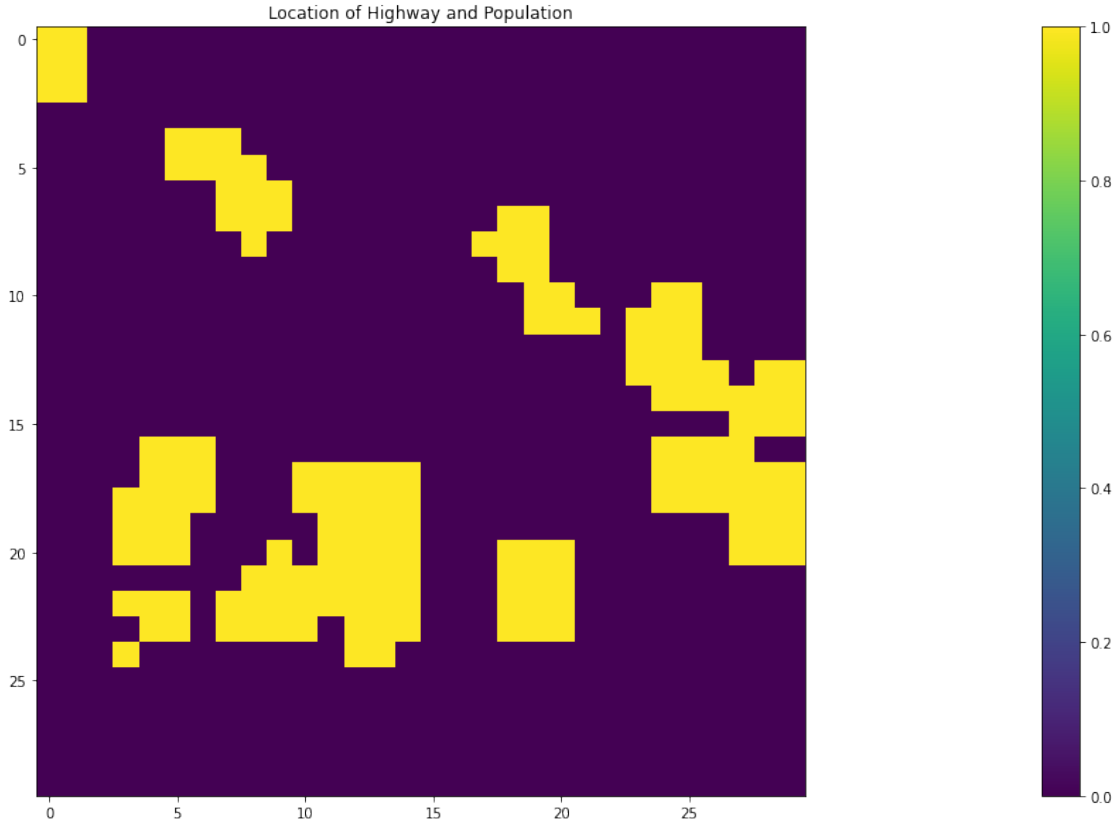
A visualization component was developed to show the results of the simulation verification generation by generation.

[73]:
```
def initialFigWithoutCharts():
    fig = plt.figure(figsize=(10,10), dpi=100)
    fig.subplots_adjust(top=1.5)
    return fig

def updateFigWithaddedChart(fig,ax,A,min,max):
    im1 = ax.imshow(A, interpolation='none',cmap='Greens', vmin=min, vmax=max)
    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im1, cax=cax, orientation='vertical')

def updateCharts(fig):
    display(fig)
    clear_output(wait = True)
    plt.pause(0.2)
```

We used a clumped world with the number of classes as 3, the percentage area of class as 30%, 30%, and

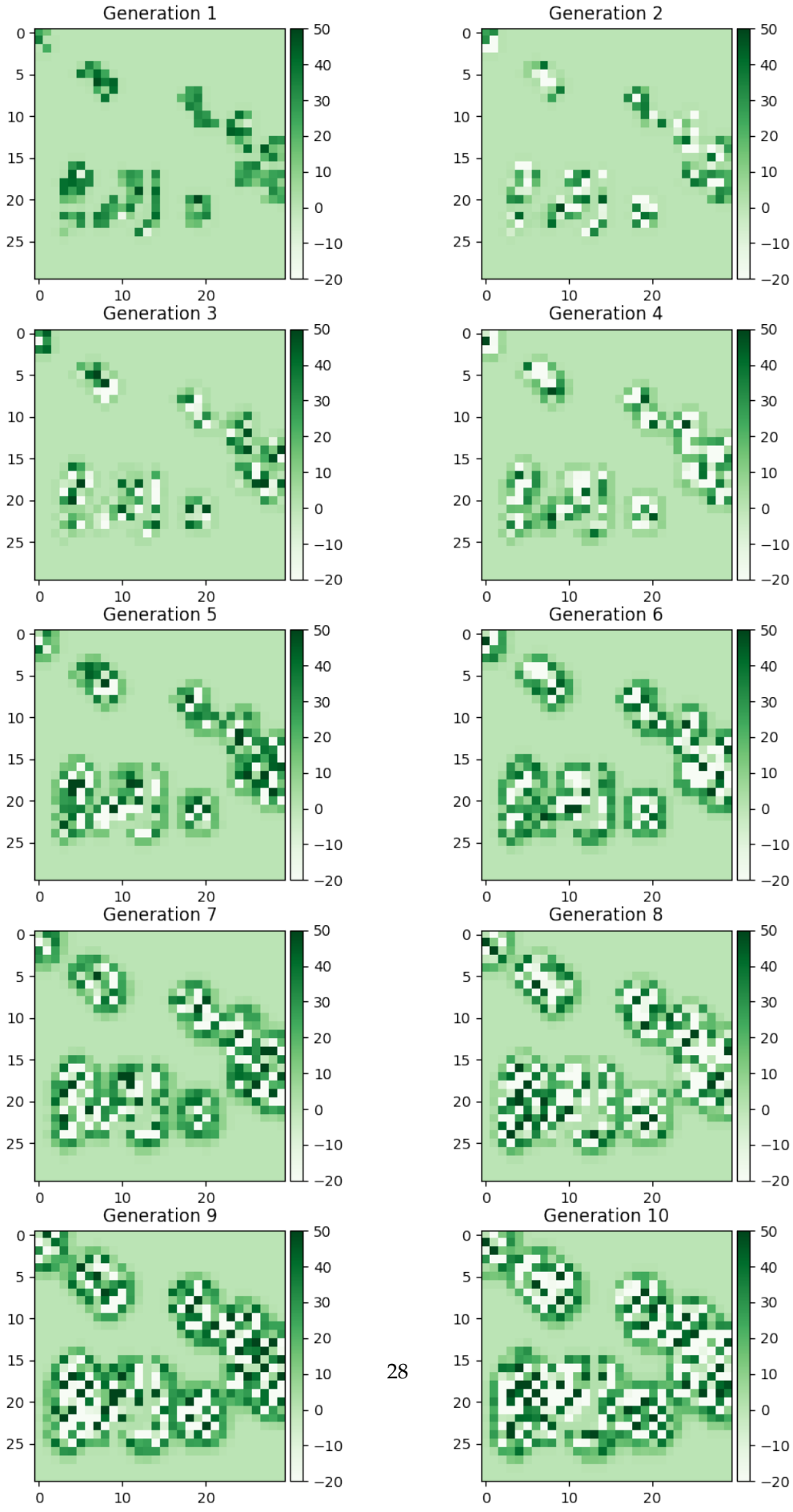
40%, respectively, occupation probability as 0.1, and the dimension of the world as 30 for verification. Based on the literature, the simulation on the world used birth rate as 1.7, natural death rate as 0.2, death rate during migration as 0.5, dispersal speed as 0.25, carrying capacity as 100, and the number of generations as 10[6]. Since we are testing the single-species model, we only pick the population with a class number of 1.

[38]:
```
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)

randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 10

simulation␣
 ↪=Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,populationType,
birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,carryingCapacity
,numberOfGenrations,world.patternWithHighway)

visualization.display2DArray(simulation.worldWithoutPopulation,1,10)
```



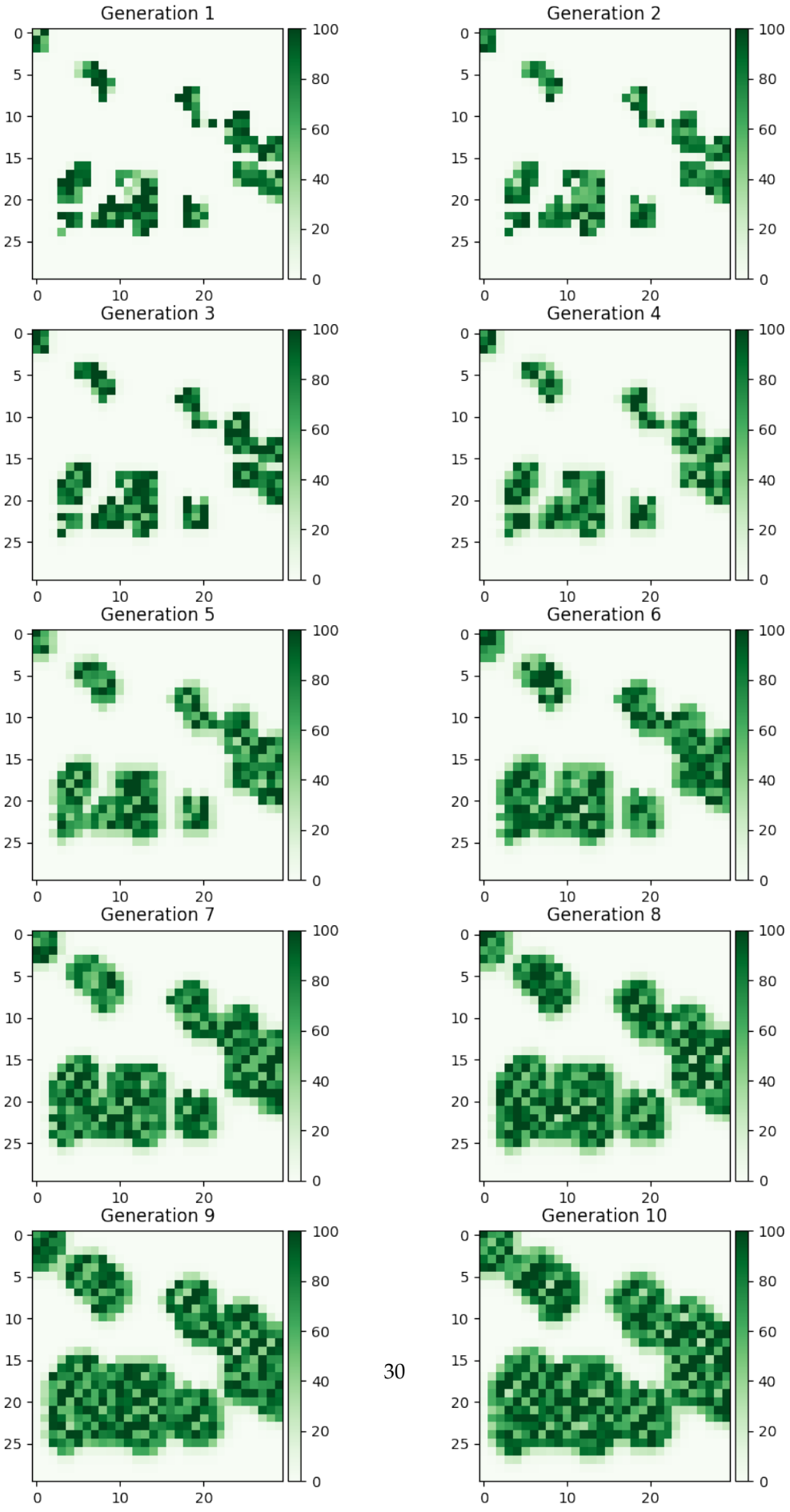
The population changes by the generations show that the population was increased initially and then decreased for the most cells with population. It can be explained by the high birth rate and low carrying capacity. After the first generation, a slight population increase happens mainly on the edge of the population clusters. In contrast, the significant population increase shows up in the middle of the population clusters. The population cluster gets bigger and bigger over time. It might be because of the high dispersal speed.

[75]:
```
simulation.verifyPopulationPlusMinus()
```


Generation 1
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6
Generation 7
Generation 8
Generation 9
Generation 10

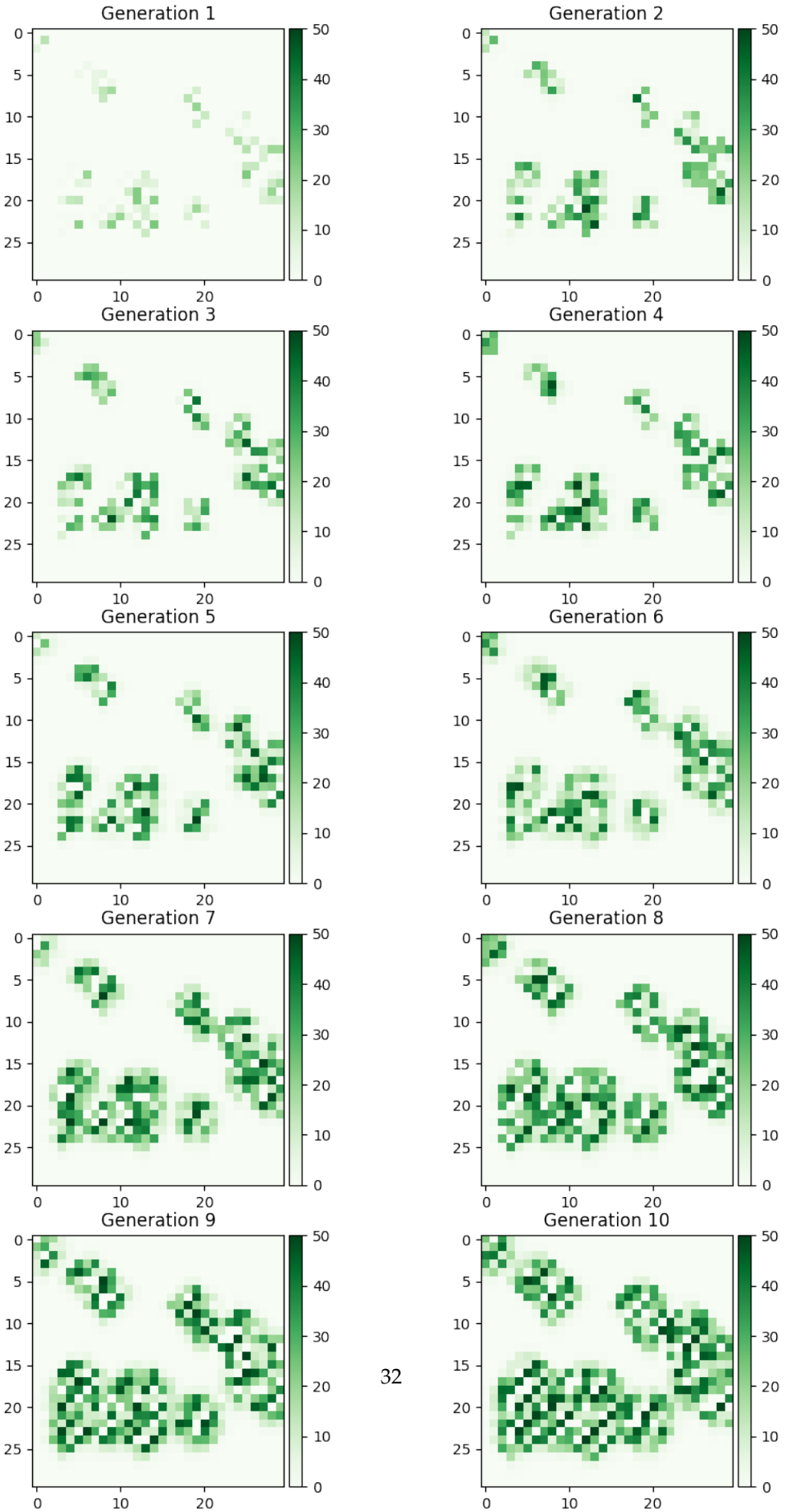
The population density by the generations shows that the cells near the high population density cells would increase faster than the cells near the low population density cells. It might be because of the population dispersion. Most of the cells with a population tend to have high population density, which is expected by the high birth rate and high dispersal speed. The population density of the cells on the edge of the population cluster is generally low. That is because the increased population of those cells is from other neighbor cells, and the death rate of the migration is high.

[40]:
```
simulation.verifyPopulationDensity()
```


Generation 1
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6
Generation 7
Generation 8
Generation 9
Generation 10

The emigration population by the generations shows that the most considerable emigration comes from the cells with the highest population density. It was expected as the population density determined the emigration. For the same cell across the generation, the emigration would decrease quickly during the simulation if the emigration were high. It is because the population density increases and exceeds the carrying capacity; thus, the population density would decrease much more than the natural process.

[41]:
```
simulation.verifyEmigPopulation()
```


Generation 1
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6
Generation 7
Generation 8
Generation 9
Generation 10

3.4 Sensitivity Analysis and Validation

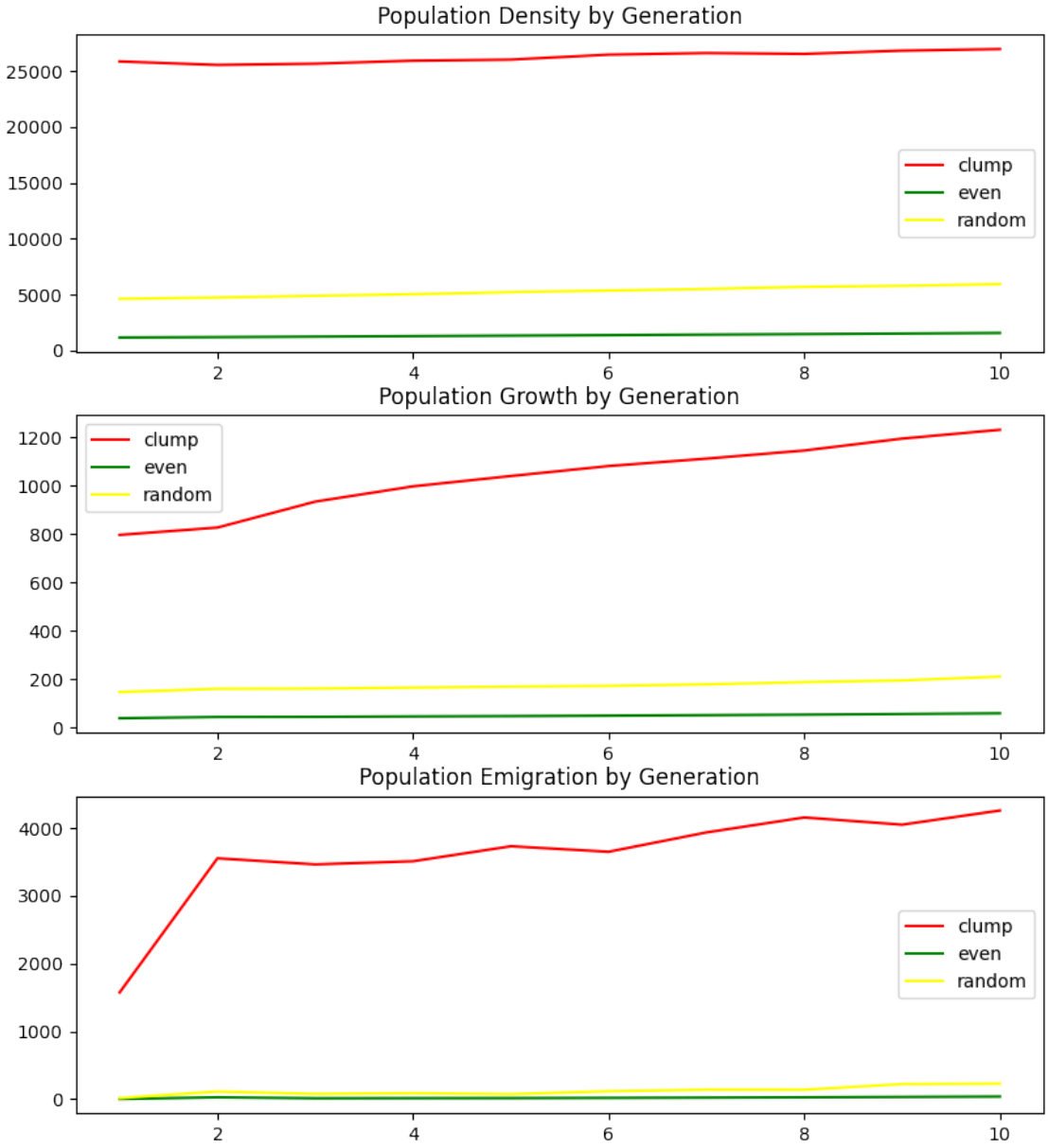
A visualization component was developed to show the results of the sensitivity analysis by multiple-line chart with capable of showing and comparing the results of three experiments in one chart.

[ ]:
```
def initialMultipleLineChartForSensitivityAnalysis():
    fig = plt.figure(figsize=(21,6), dpi=100)
    fig.subplots_adjust(top=1.5)
    ax1 = fig.add_subplot(3, 2, 1)
    ax2 = fig.add_subplot(3, 2, 3)
    ax3 = fig.add_subplot(3, 2, 5)
    return ax1, ax2, ax3

def addMultipleLineToChart(ax,␣
 ↪xdata,ydata,x2data,y2data,x3data,y3data,title,color,label,color2,label2,color3,label3):
 ↪
    ax.plot(xdata, ydata, color=color, label=label)
    ax.plot(x2data, y2data, color=color2, label=label2)
    ax.plot(x3data, y3data, color=color3, label=label3)
    ax.title.set_text(title)
    ax.legend()

def showChart():
    plt.show()
```

The initial pattern was set as clumped, even, and random, while other parameters were the same for the experiments. The results show that the clumped pattern has higher population density, natural population growth, and emigrant population. It was expected as the total population size of the clumped pattern is bigger than others, and the natural population growth and population emigration are partially determined by population density.

[43]:
```
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x1 = simulation.stepArray
y11 = simulation.totalPopulationSizeArray
y12 = simulation.GrowthPopulationSizeArray
y13 = simulation.EmigrationPopulationSizeArray

world = World(0.1,3,30,[0.3,0.3,0.4],100,"even",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x2 = simulation.stepArray
y21 = simulation.totalPopulationSizeArray
y22 = simulation.GrowthPopulationSizeArray
y23 = simulation.EmigrationPopulationSizeArray

world = World(0.1,3,30,[0.3,0.3,0.4],100,"random",5,10)
```

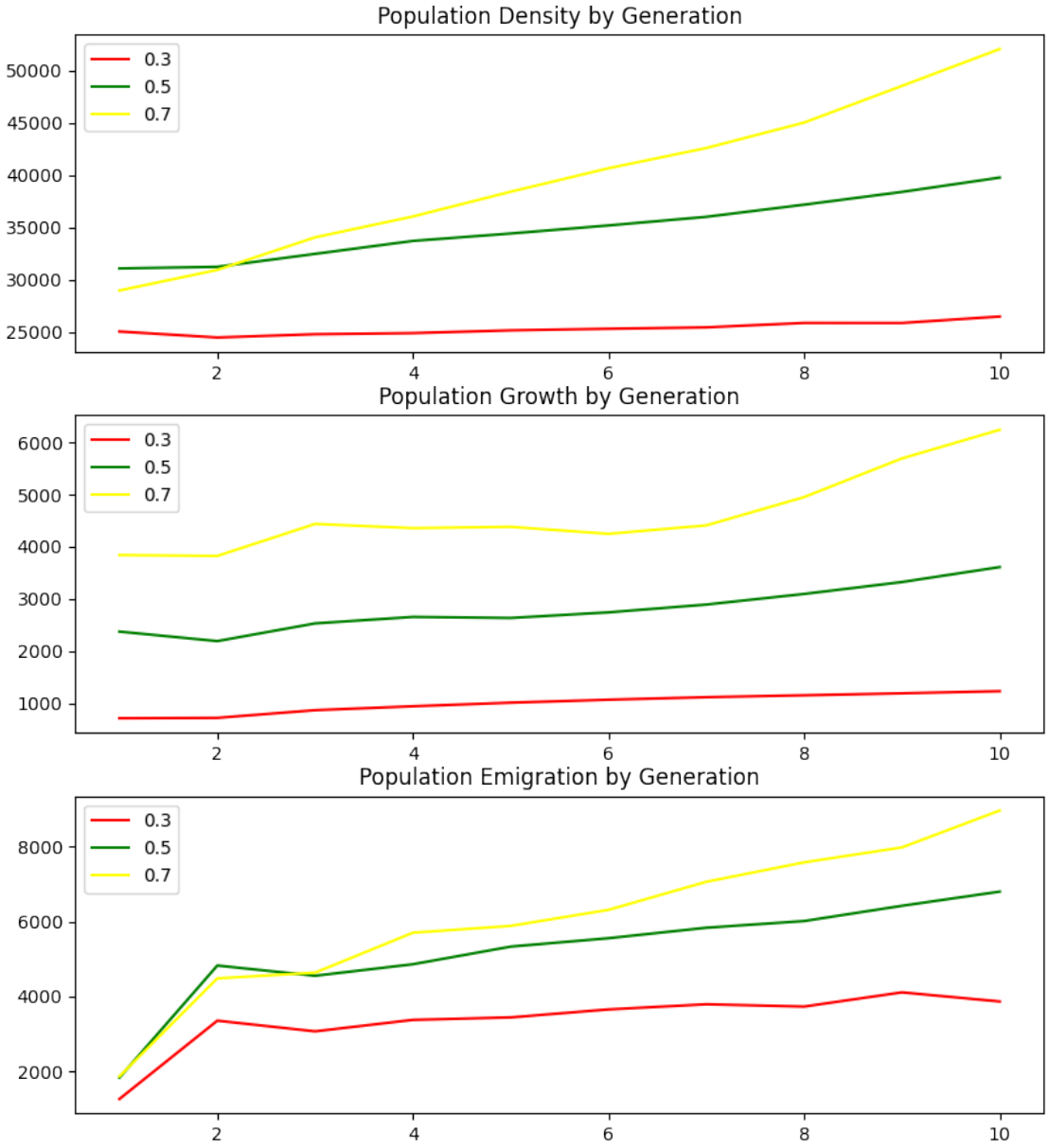
```
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x3 = simulation.stepArray
y31 = simulation.totalPopulationSizeArray
y32 = simulation.GrowthPopulationSizeArray
y33 = simulation.EmigrationPopulationSizeArray


ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addMultipleLineToChart(ax1, x1,y11,x2,y21,x3,y31,"Population Density by␣
 ↪Generation","red","clump","green","even","yellow","random")
visualization.addMultipleLineToChart(ax2, x1,y12,x2,y22,x3,y32,"Population Growth by␣
 ↪Generation","red","clump","green","even","yellow","random")
visualization.addMultipleLineToChart(ax3, x1,y13,x2,y23,x3,y33,"Population Emigration␣
 ↪by Generation","red","clump","green","even","yellow","random")
visualization.showChart()
```



The birth rate was set as 0.3, 0.5, and 0.7, while other parameters were the same for the experiments. The results show that the experiment with a birth rate of 0.7 has the highest population density, natural population growth, and emigrant population. In contrast, the experiment with a birth rate of 0.3 has the lowest ones.

[81]:
```
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x1 = simulation.stepArray
y11 = simulation.totalPopulationSizeArray
```

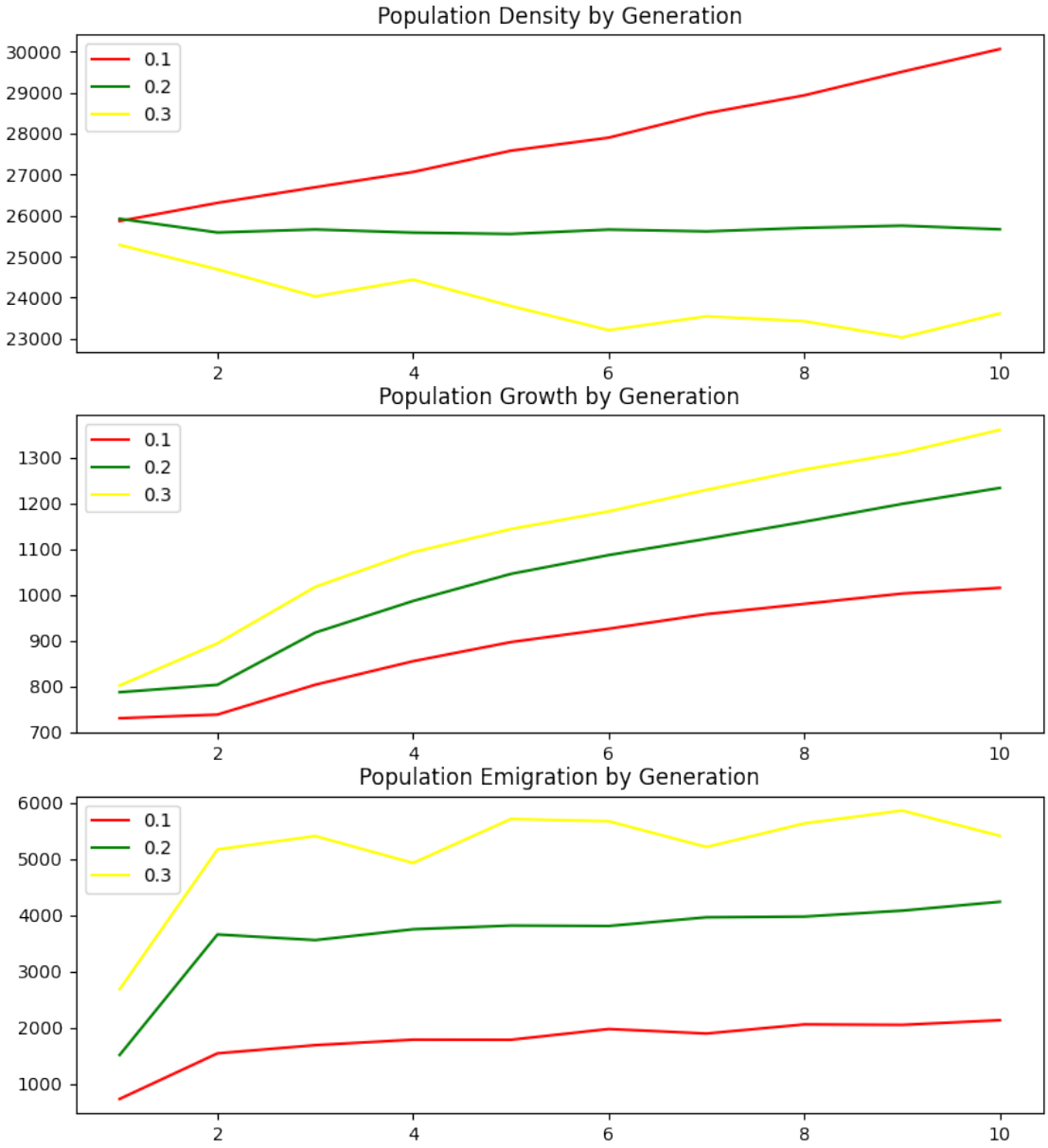
```
y12 = simulation.GrowthPopulationSizeArray
y13 = simulation.EmigrationPopulationSizeArray
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.5,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x2 = simulation.stepArray
y21 = simulation.totalPopulationSizeArray
y22 = simulation.GrowthPopulationSizeArray
y23 = simulation.EmigrationPopulationSizeArray
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.7,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x3 = simulation.stepArray
y31 = simulation.totalPopulationSizeArray
y32 = simulation.GrowthPopulationSizeArray
y33 = simulation.EmigrationPopulationSizeArray

ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addMultipleLineToChart(ax1, x1,y11,x2,y21,x3,y31,"Population Density by␣
 ↪Generation","red","0.3","green","0.5","yellow","0.7")
visualization.addMultipleLineToChart(ax2, x1,y12,x2,y22,x3,y32,"Population Growth by␣
 ↪Generation","red","0.3","green","0.5","yellow","0.7")
visualization.addMultipleLineToChart(ax3, x1,y13,x2,y23,x3,y33,"Population Emigration␣
 ↪by Generation","red","0.3","green","0.5","yellow","0.7")
visualization.showChart()
```



The dispersal speed was set as 0.1, 0.2, and 0.3, while other parameters were the same for the experiments. The results show that the experiment with a dispersal speed of 0.3 has the highest natural population growth and emigrant population but the lowest population density. In contrast, the experiment with a dispersal speed of 0.1 has the lowest natural population growth and emigrant population but the highest population density. It is because the natural population growth is not only controlled by the birth rate and natural death rate but also affected by the carrying capacity. If the population density exceeds the carrying capacity, the death rate would be higher than the natural death rate, which means the natural population growth would be lower. If the dispersal speed is higher, which means the emigrant population would be more, the population density would be less likely to hit the carrying capacity. Thus it increases the natural population growth globally. But since the higher dispersal speed would cause more populations

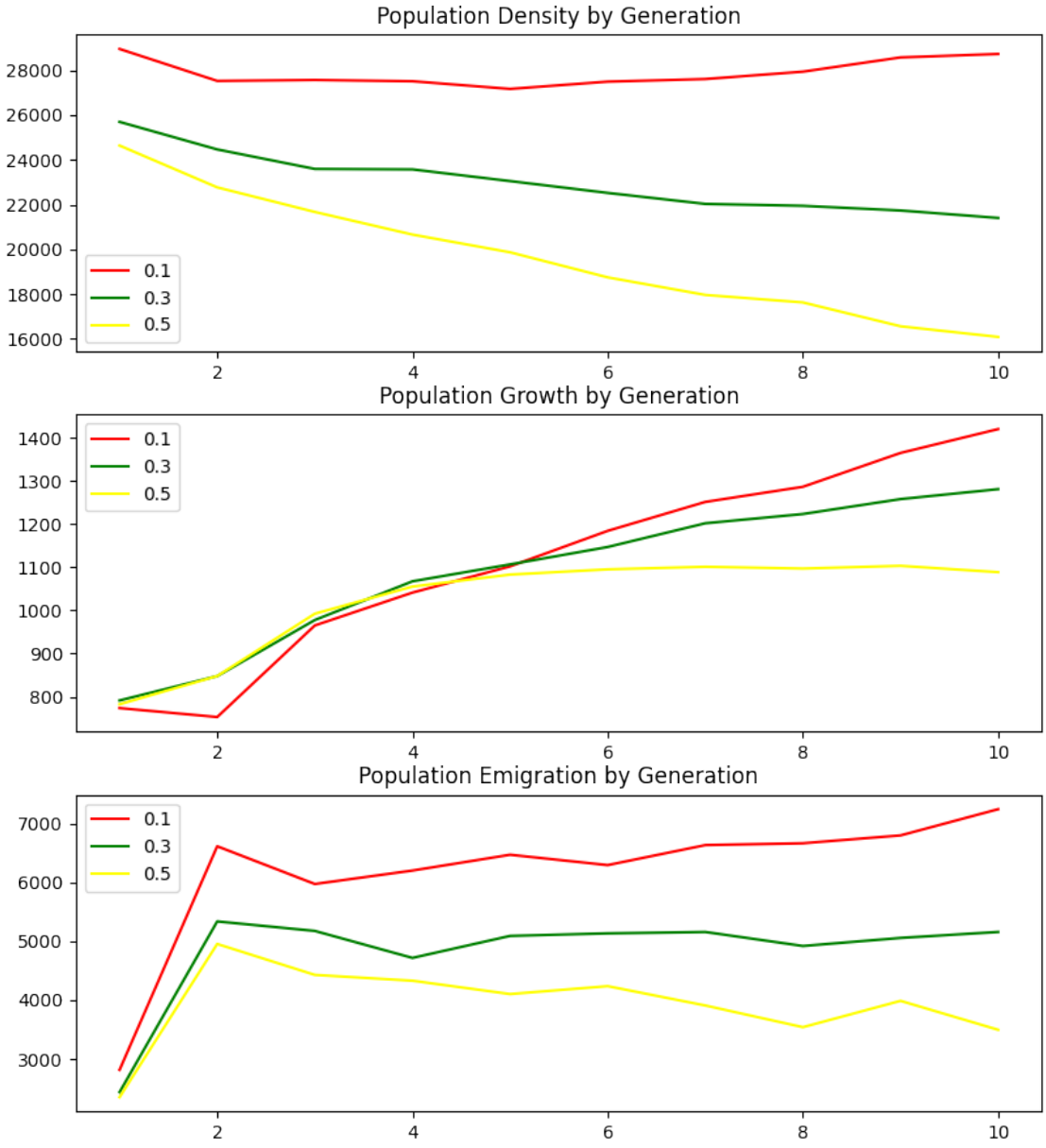
to die during the migration, the total population density with higher dispersal would be lower than the population density of others.

[99]:
```
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.1,300,10,world.patternWithHighway)
simulation.run()
x1 = simulation.stepArray
y11 = simulation.totalPopulationSizeArray
y12 = simulation.GrowthPopulationSizeArray
y13 = simulation.EmigrationPopulationSizeArray

world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.2,300,10,world.patternWithHighway)
simulation.run()
x2 = simulation.stepArray
y21 = simulation.totalPopulationSizeArray
y22 = simulation.GrowthPopulationSizeArray
y23 = simulation.EmigrationPopulationSizeArray

world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.2,0.3,300,10,world.patternWithHighway)
simulation.run()
x3 = simulation.stepArray
y31 = simulation.totalPopulationSizeArray
y32 = simulation.GrowthPopulationSizeArray
y33 = simulation.EmigrationPopulationSizeArray

ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addMultipleLineToChart(ax1, x1,y11,x2,y21,x3,y31,"Population Density by␣
↪Generation","red","0.1","green","0.2","yellow","0.3")
visualization.addMultipleLineToChart(ax2, x1,y12,x2,y22,x3,y32,"Population Growth by␣
↪Generation","red","0.1","green","0.2","yellow","0.3")
visualization.addMultipleLineToChart(ax3, x1,y13,x2,y23,x3,y33,"Population Emigration␣
↪by Generation","red","0.1","green","0.2","yellow","0.3")
visualization.showChart()
```



The death rate during migration was set as 0.1, 0.3, and 0.5, while other parameters were the same for the experiments. The results show that the experiment with a death rate during migration of 0.1 has the highest population density, natural population growth, and emigrant population. In contrast, the experiment with a death rate of 0.5 has the lowest ones. It is because higher death rate during migration would kill more populations and reduce the number of populations reach the destination. Thus, the population density of the destination cell would be lower than the population density in other scenarios. Considering the natural population growth and emigrant population were positively related to population density, lower population density means lower natural population growth and emigrant population.

[77]:
```
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.1,0.3,300,10,world.patternWithHighway)
simulation.run()
x1 = simulation.stepArray
y11 = simulation.totalPopulationSizeArray
y12 = simulation.GrowthPopulationSizeArray
y13 = simulation.EmigrationPopulationSizeArray
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.3,0.3,300,10,world.patternWithHighway)
simulation.run()
x2 = simulation.stepArray
y21 = simulation.totalPopulationSizeArray
y22 = simulation.GrowthPopulationSizeArray
y23 = simulation.EmigrationPopulationSizeArray
world = World(0.1,3,30,[0.3,0.3,0.4],100,"clump",5,10)
simulation = Simulation(30,120,10,1,0.3,0.2,0.5,0.3,300,10,world.patternWithHighway)
simulation.run()
x3 = simulation.stepArray
y31 = simulation.totalPopulationSizeArray
y32 = simulation.GrowthPopulationSizeArray
y33 = simulation.EmigrationPopulationSizeArray


ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addMultipleLineToChart(ax1, x1,y11,x2,y21,x3,y31,"Population Density by␣
 ↪Generation","red","0.1","green","0.3","yellow","0.5")
visualization.addMultipleLineToChart(ax2, x1,y12,x2,y22,x3,y32,"Population Growth by␣
 ↪Generation","red","0.1","green","0.3","yellow","0.5")
visualization.addMultipleLineToChart(ax3, x1,y13,x2,y23,x3,y33,"Population Emigration␣
 ↪by Generation","red","0.1","green","0.3","yellow","0.5")
visualization.showChart()
```



3.5 Simulation Analysis

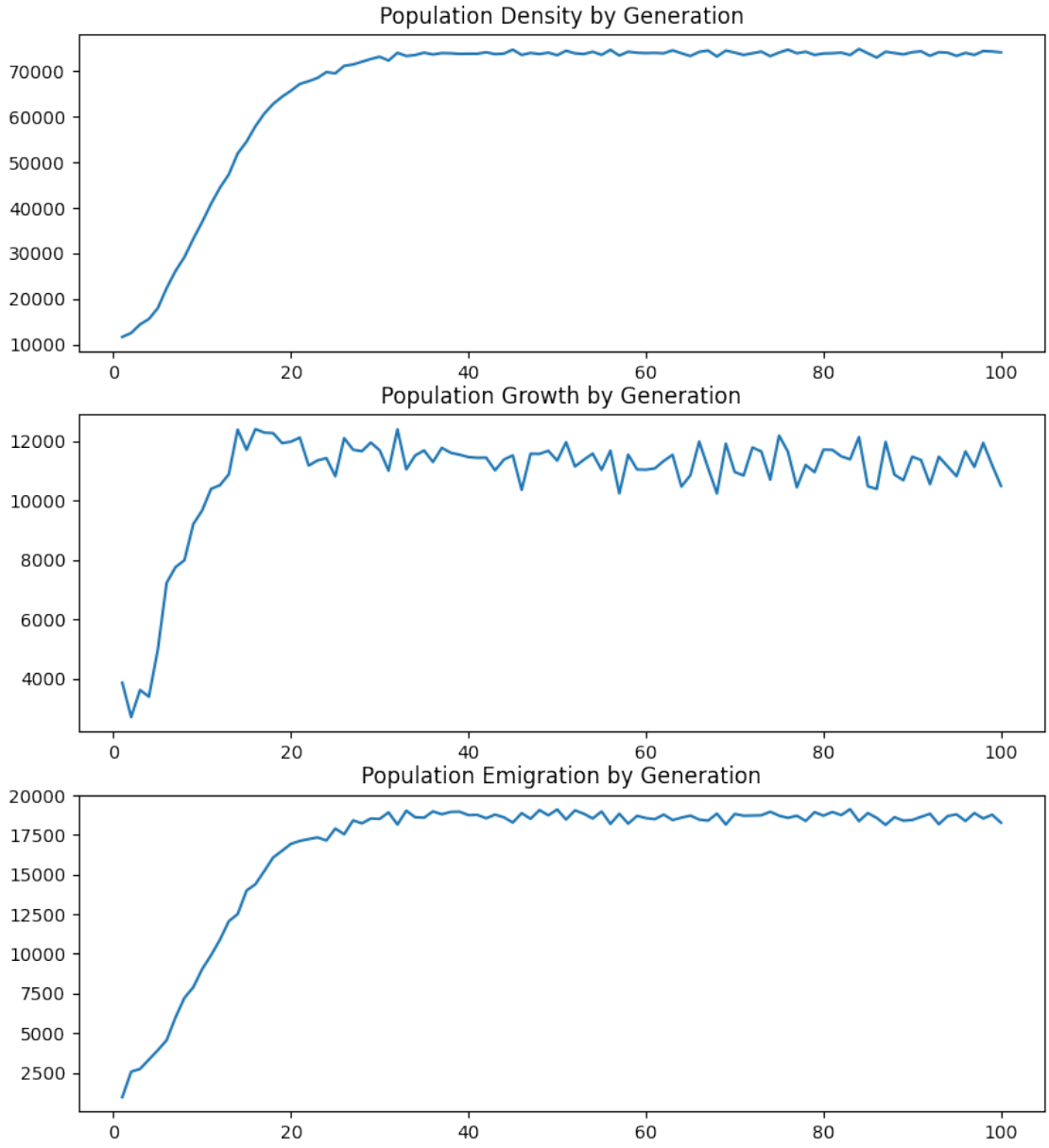
A visualization component was developed to show the results of the simulation analysis.

[ ]:
```
def addLineToChart(ax, xdata,ydata,title):
    ax.plot(xdata, ydata)
    ax.title.set_text(title)
```

To find out the steady-state of the system, we simulate by 100 generations and monitor population density,

natural population growth, and emigrant population. The results show that the system reaches a steady-state at the 40th generation.

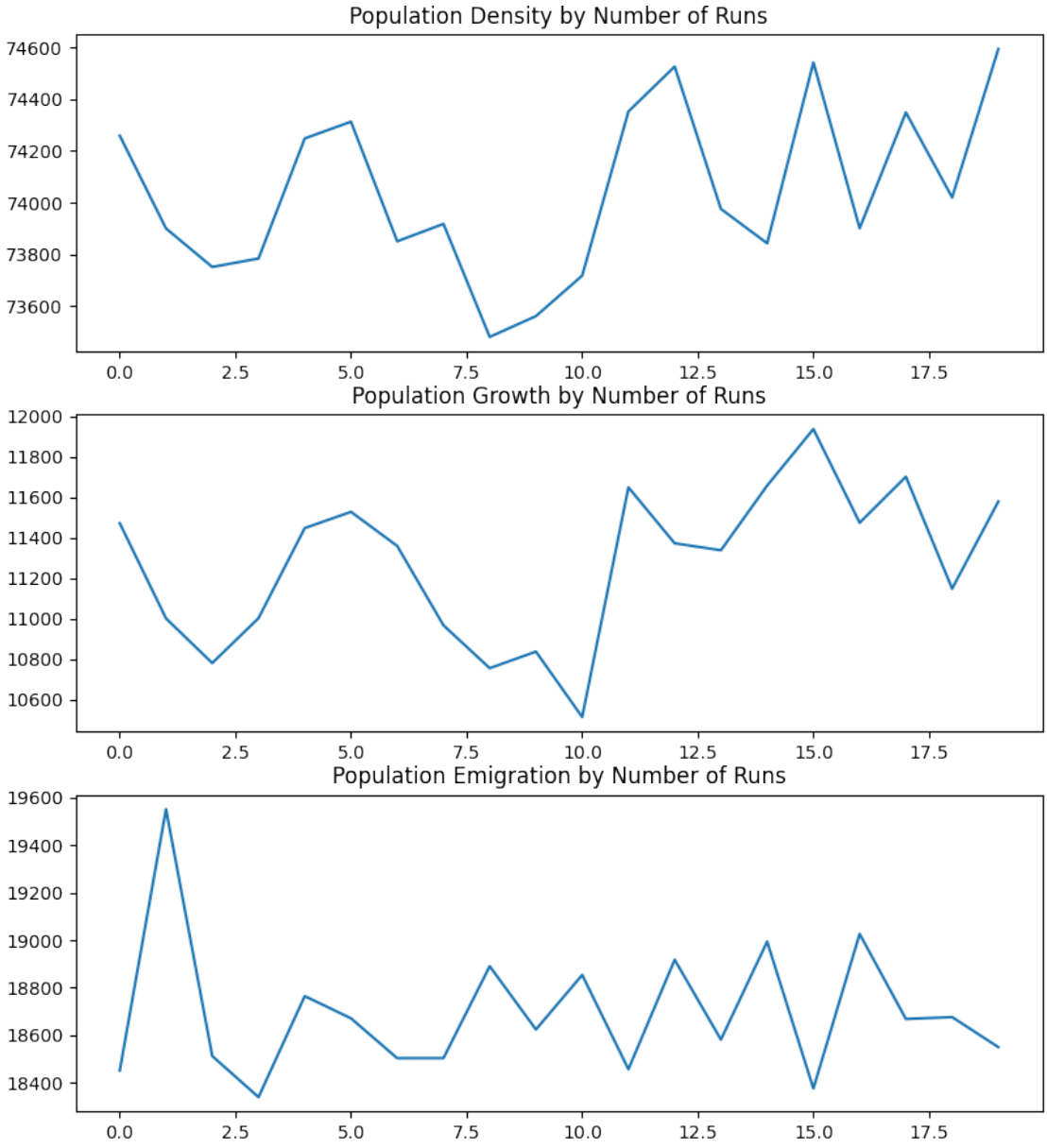
[48]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,
randomSeed,patternMode,gapSpace,highwayClass)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 100


simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
carryingCapacity,numberOfGenrations,world.patternWithHighway)


simulation.run()
x = simulation.stepArray
y11 = simulation.totalPopulationSizeArray
y12 = simulation.GrowthPopulationSizeArray
y13 = simulation.EmigrationPopulationSizeArray

ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addLineToChart(ax1, x,y11,"Population Density by Generation")
visualization.addLineToChart(ax2, x,y12,"Population Growth by Generation")
visualization.addLineToChart(ax3, x,y13,"Population Emigration by Generation")
visualization.showChart()
```



To better understand the range of the average value of the final outputs, we simulated the system until the steady-state for 20 times. We then calculated the confidence interval at the confidence level of 95% for population density, natural population growth, and emigrant population. The results show that the confidence interval of population density, natural population growth, and emigrant population are (73350.57, 74738.35), (10482.36, 12071.25), and (18100.61, 19291.12) respectively.

[49]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
```

```
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,randomSeed,patternMode,gapSpace,
highwayClass)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 40

numberOfRuns = 20
runIndexArray = []
finalTotalPopulationSizeArray = []
finalGrowthPopulationSizeArray = []
finalEmigrationPopulationSizeArray = []

for i in range(0,numberOfRuns):
    simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
    populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
    carryingCapacity,numberOfGenrations,world.patternWithHighway)
    simulation.run()
    runIndexArray.append(i)
    finalTotalPopulationSizeArray.append(simulation.totalPopulationSizeArray[39])
    finalGrowthPopulationSizeArray.append(simulation.GrowthPopulationSizeArray[39])
    finalEmigrationPopulationSizeArray.append(simulation.
 ↪EmigrationPopulationSizeArray[39])

x = runIndexArray
y11 = finalTotalPopulationSizeArray
y12 = finalGrowthPopulationSizeArray
y13 = finalEmigrationPopulationSizeArray

ax1, ax2, ax3 = visualization.initialMultipleLineChartForSensitivityAnalysis()
visualization.addLineToChart(ax1, x,y11,"Population Density by Number of Runs")
visualization.addLineToChart(ax2, x,y12,"Population Growth by Number of Runs")
visualization.addLineToChart(ax3, x,y13,"Population Emigration by Number of Runs")
visualization.showChart()
```



Functions were developed to calculate the confidence interval based on the confidence level, degree of freedom, average of the sample values and stand deviation of the sample values.

[50]:
```
import statistics
import scipy.stats

def mean(array):
    return statistics.mean(array)
```

```
def stdev(array):
    return statistics.stdev(array)

def confidence_interval(array):
    confidence_level = 0.95
    degrees_freedom = 19
    return scipy.stats.t.interval(confidence_level, degrees_freedom, mean(array),␣
 ↪stdev(array))
```

[53]:
```
interval = confidence_interval(finalTotalPopulationSizeArray)
print('Confidence Interval of population density at steady state on 95% confidence␣
 ↪level',interval)
interval = confidence_interval(finalGrowthPopulationSizeArray)
print('Confidence Interval of natural population growth at steady state on 95%␣
 ↪confidence level',interval)
interval = confidence_interval(finalEmigrationPopulationSizeArray)
print('Confidence Interval of emigrated population at steady state on 95% confidence␣
 ↪level',interval)
```

```
Confidence Interval of population density at steady state on 95% confidence
level (73350.57304412275, 74738.35033753858)
Confidence Interval of natural population growth at steady state on 95%
confidence level (10482.357614776198, 12071.25435483808)
Confidence Interval of emigrated population at steady state on 95% confidence
level (18100.606503335763, 19291.12283500747)
```

3.6 Scenario Analysis with Road Network

A visualization component was developed to show the results of the scenario analysis with the road network. Eight plots were designed to monitor the progress of the simulation. They are population density by the grid, population +/- by the grid, population growth by the grid, population emigration by the grid, population immigration by the grid, population size by generation, population killed by road by generation, and Emigrant population by generation.

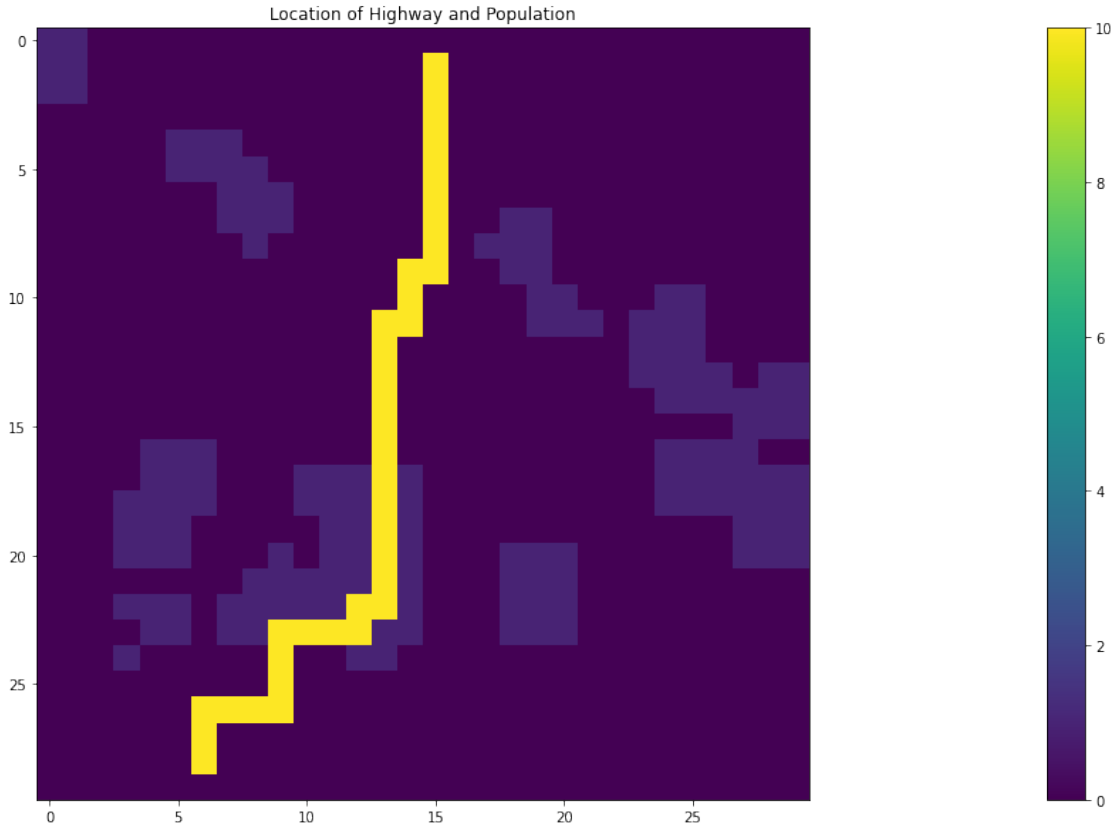
[ ]:
```
def initialobserver():
    fig = plt.figure(figsize=(10,10), dpi=100)
    fig.subplots_adjust(top=1.5)

    ax1 = fig.add_subplot(4, 2, 1)
    ax1.title.set_text('Population Density By Grid')
    ax2 = fig.add_subplot(4, 2, 2)
    ax2.title.set_text('Population +/- By Grid')
    ax3 = fig.add_subplot(4, 2, 3)
    ax3.title.set_text('Population Growth By Grid')
    ax4 = fig.add_subplot(4, 2, 4)
    ax4.title.set_text('Population Emigration By Grid')
    ax5 = fig.add_subplot(4, 2, 5)
    ax5.title.set_text('Population Immigration By Grid')

    ax6 = fig.add_subplot(4, 2, 6)
    ax7 = fig.add_subplot(4, 2, 7)
```

```
    ax8 = fig.add_subplot(4, 2, 8)

    xdata = []
    ydata = []
    x2data = []
    y2data = []
    x3data = []
    y3data = []
    return fig,ax1,ax2,ax3,ax4,ax5,ax6,ax7,ax8,xdata,x2data,x3data,ydata,y2data,y3data

def updateDisplayPopulationDensity(fig,ax,A):
    im1 = ax.imshow(A, interpolation='none',cmap='Greens', vmin=0, vmax=100)
    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im1, cax=cax, orientation='vertical')


def updateDisplayPopulationDensityPlusMinus(fig,ax,A):
    im1 = ax.imshow(A, interpolation='none',cmap='Greens', vmin=-20, vmax=50)
    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im1, cax=cax, orientation='vertical')

def updateDisplayPopulationChanged(fig,ax,A):
    im1 = ax.imshow(A, interpolation='none',cmap='Greens', vmin=0, vmax=50)
    divider = make_axes_locatable(ax)
    cax = divider.append_axes('right', size='5%', pad=0.05)
    fig.colorbar(im1, cax=cax, orientation='vertical')

def updateLineChart(ax,xdata,ydata,step,value,title):
    xdata.append(step)
    ydata.append(value)
    ax.set_xlim(0, step)
    ax.cla()
    ax.title.set_text(title)
    ax.plot(xdata, ydata)

def updateCharts(fig):
    display(fig)
    clear_output(wait = True)
    plt.pause(0.2)
```

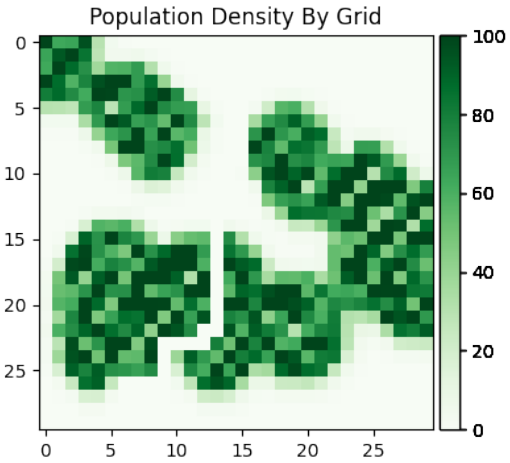
One road/ highway was added to clumped population world and overlapping with one population cluster. It was used as the world for the first scenario analysis. We simulated the system for ten generations and used the same global setting as the previous simulations. The results show that around 300 individuals were killed by road traffic, but the total population density increased significantly. The population cluster expanded, and some small patches have been merged into a new large one.

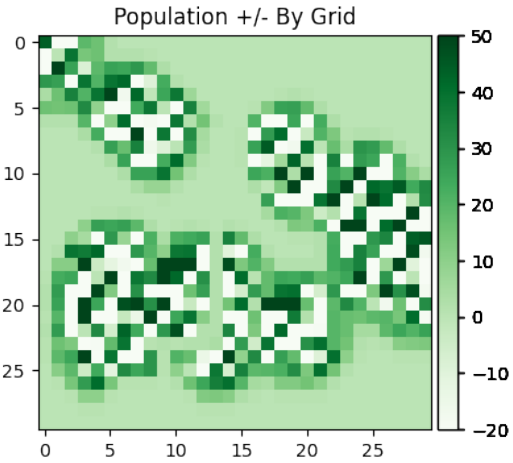
[71]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
```

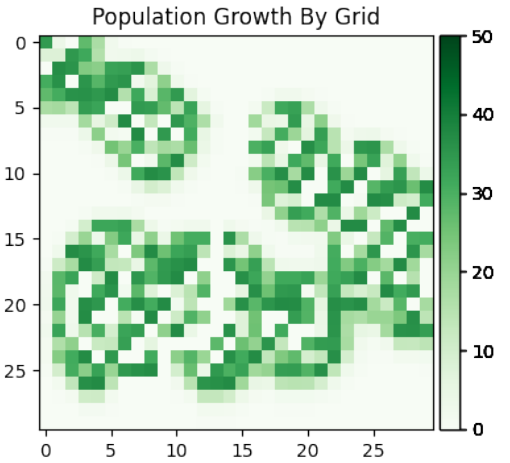
```
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,randomSeed,patternMode,gapSpace,
highwayClass)

# constant for road 1
probAsEndOfRoad = 0.5
probForDirection = [0.2,0.2,0.6]
randomSeedHighway = 20

world.generatePatternWithHghway(probAsEndOfRoad,probForDirection,randomSeedHighway)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 10


simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
carryingCapacity,numberOfGenrations,world.patternWithHighway)
visualization.display2DArray(simulation.
 ↪worldWithoutPopulation,populationType,highwayClass)
```

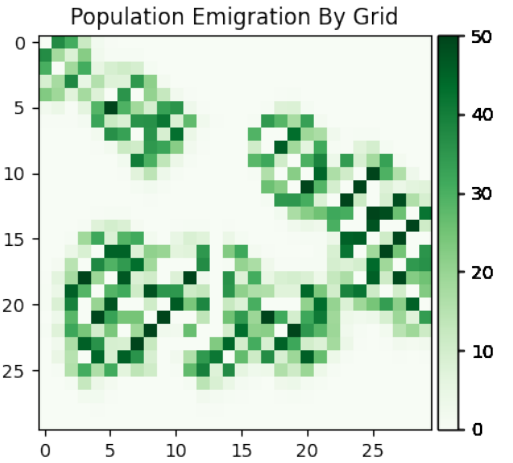


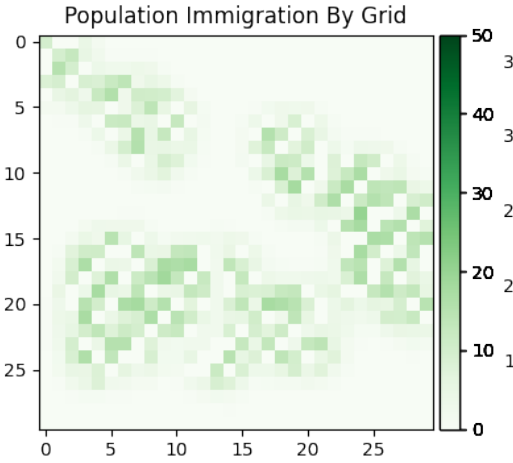
[72]:
```
simulation.runWithHighway()
```

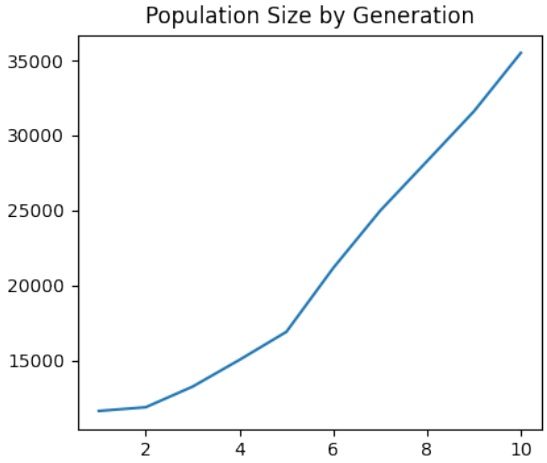

Population Density By Grid

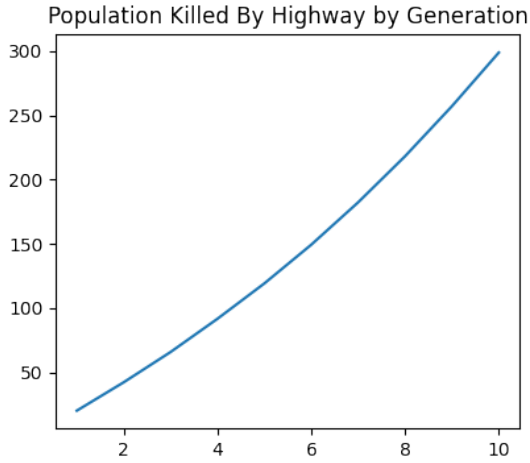

Population +/- By Grid

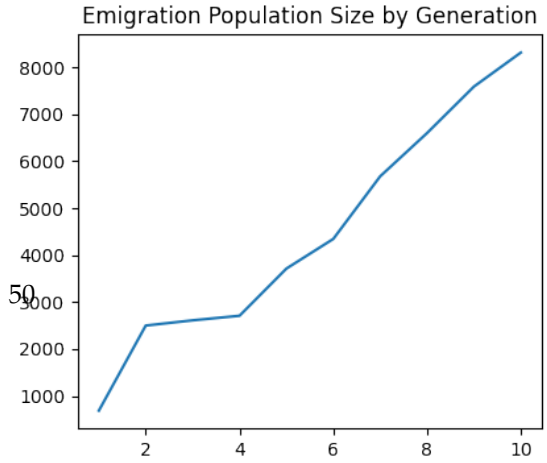

Population Growth By Grid


Population Emigration By Grid


Population Immigration By Grid


Population Size by Generation

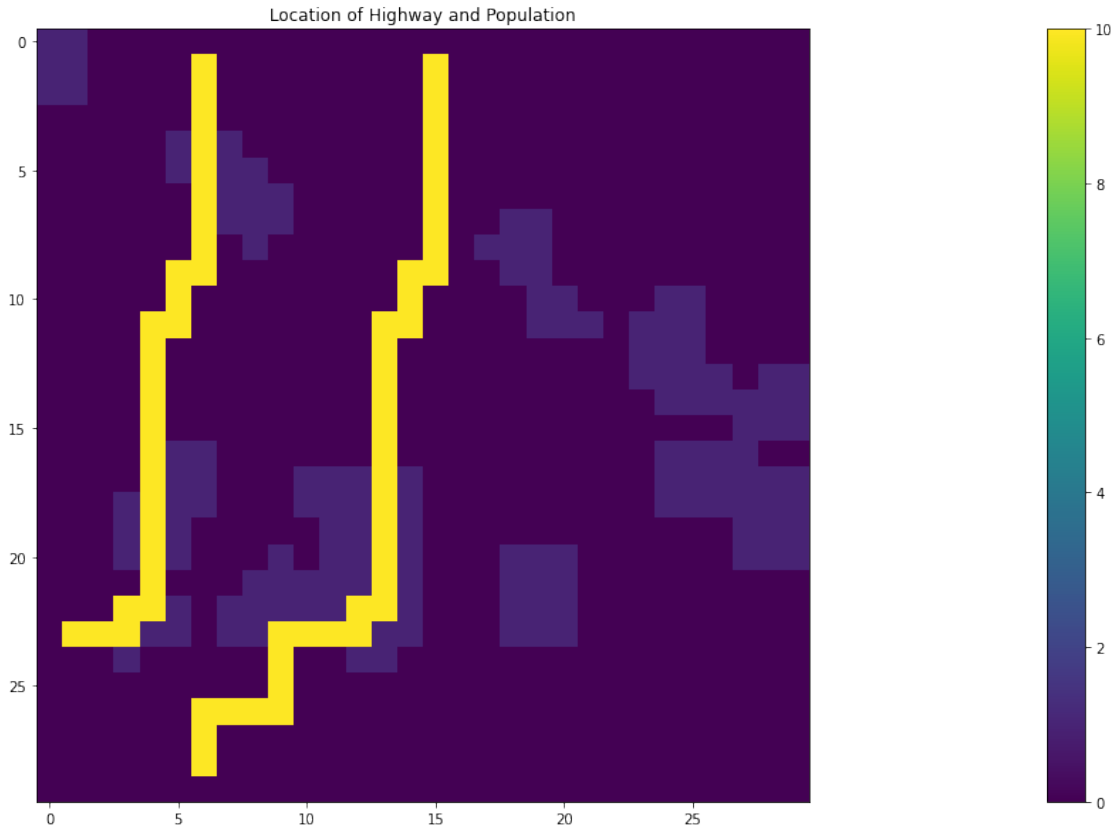

Population Killed By Highway by Generation


Emigration Population Size by Generation

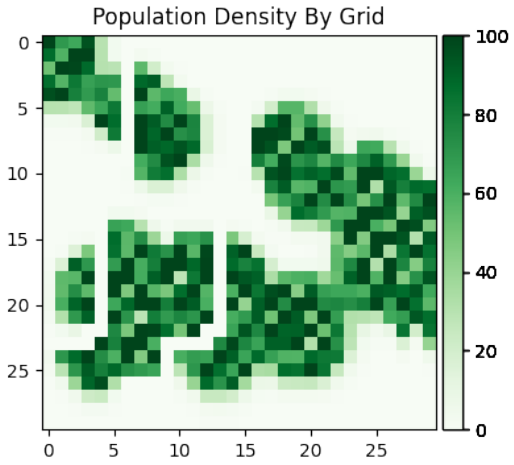
Two roads/highways were added to the clumped population world and overlapped with more population clusters than the previous scenario. It was used as the world for the second scenario analysis. We simulated the system for ten generations and used the same global setting as the previous simulations. The results show that around 600 individuals were killed by road traffic. The total population density increased but less than the last scenario. The population clusters expanded, but the road split the population clusters into more patches.

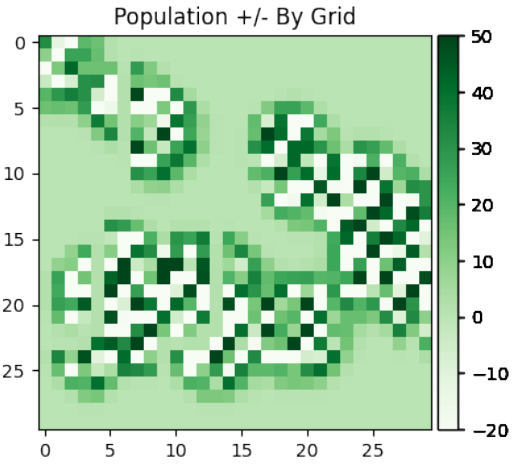
[61]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,randomSeed,patternMode,gapSpace,
highwayClass)

# constant for road 1
probAsEndOfRoad = 0.5
probForDirection = [0.2,0.2,0.6]
randomSeedHighway = 20

world.generatePatternWithHghway(probAsEndOfRoad,probForDirection,randomSeedHighway)

# constant for road 1
probAsEndOfRoad = 0.2
probForDirection = [0.2,0.2,0.6]
randomSeedHighway = 20

world.generatePatternWithHghway(probAsEndOfRoad,probForDirection,randomSeedHighway)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 10


simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
carryingCapacity,numberOfGenrations,world.patternWithHighway)
```

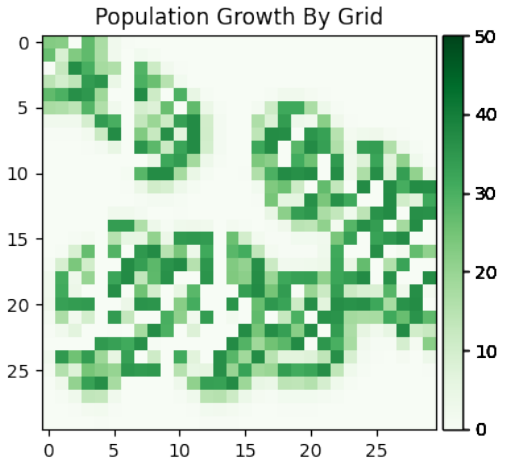
```
visualization.display2DArray(simulation.
 ↪worldWithoutPopulation,populationType,highwayClass)
```

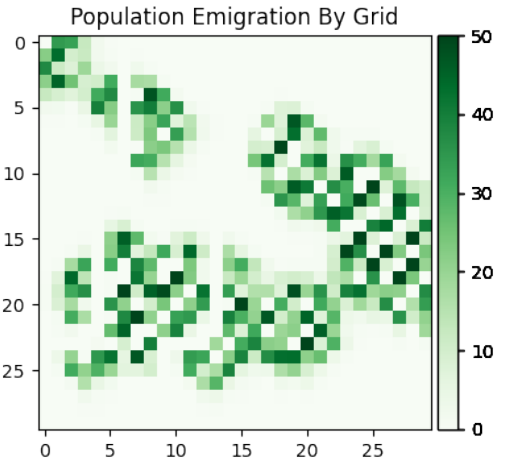


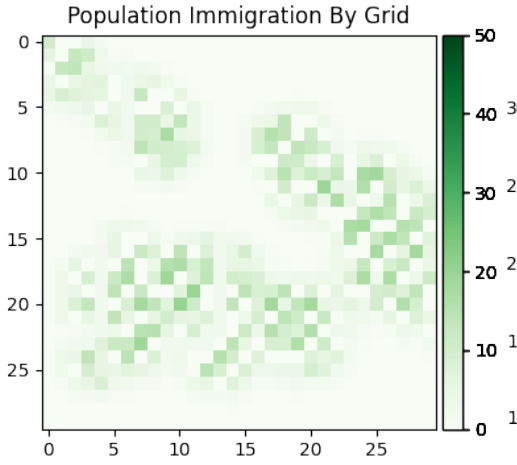
[62]:
```
simulation.runWithHighway()
```

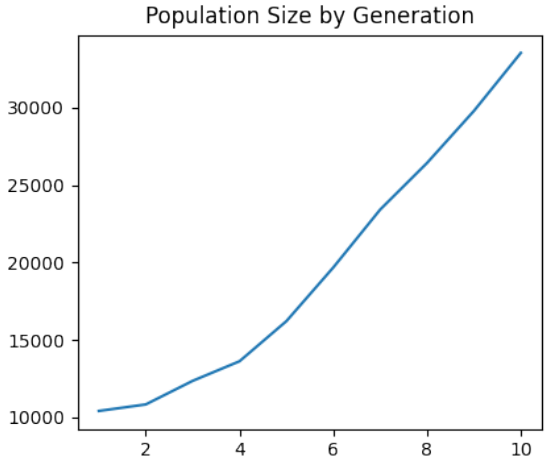

Population Density By Grid

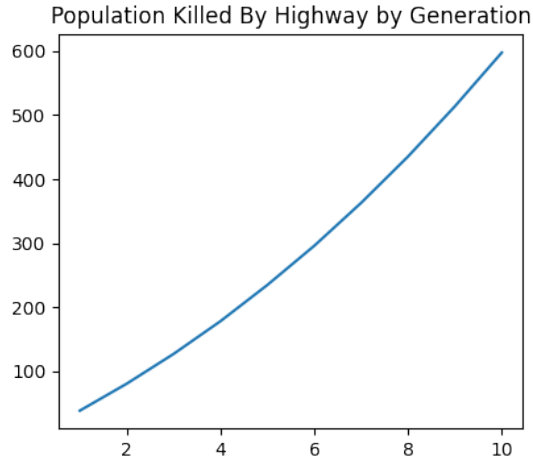

Population +/- By Grid

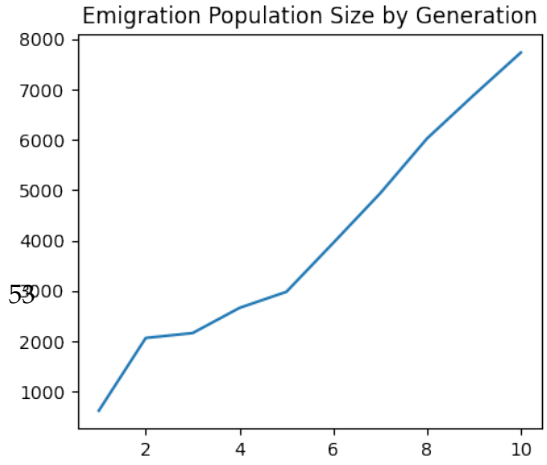

Population Growth By Grid


Population Emigration By Grid


Population Immigration By Grid

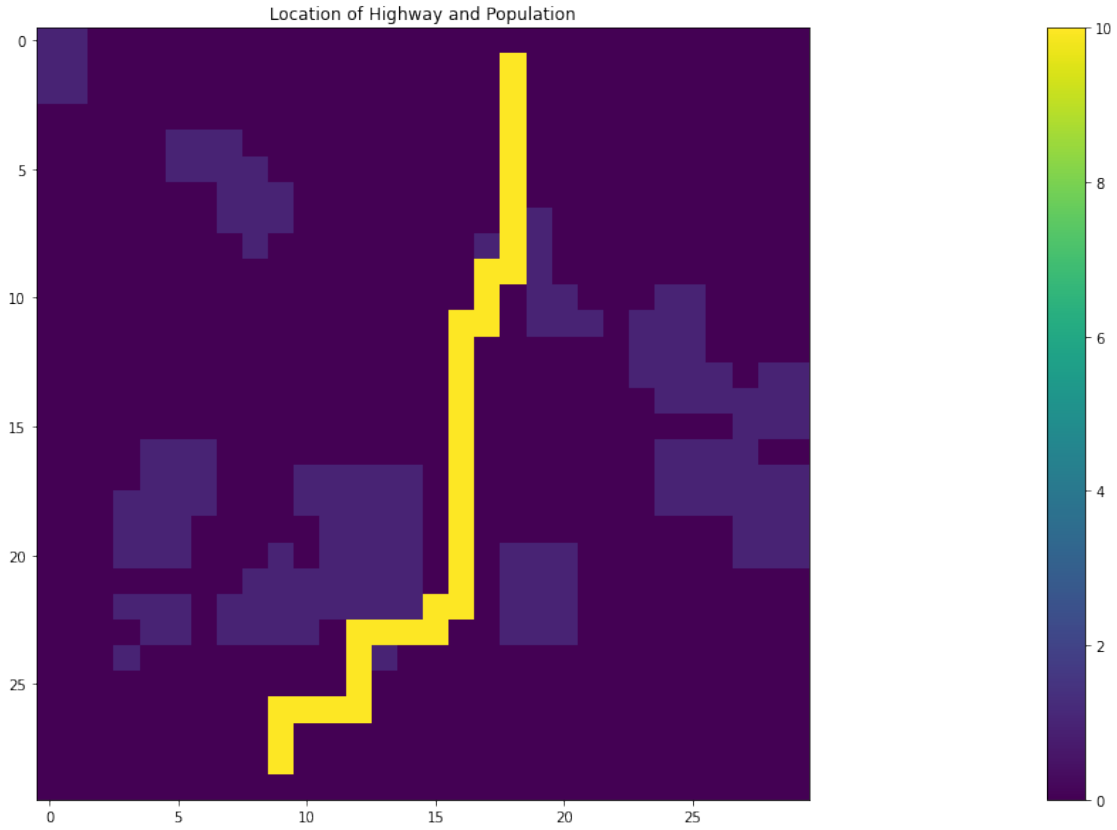

Population Size by Generation


Population Killed By Highway by Generation

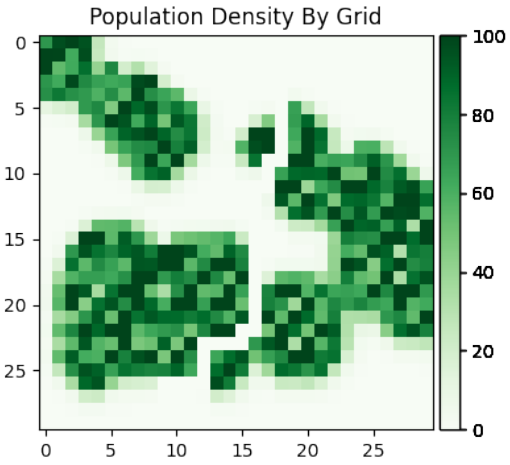

Emigration Population Size by Generation

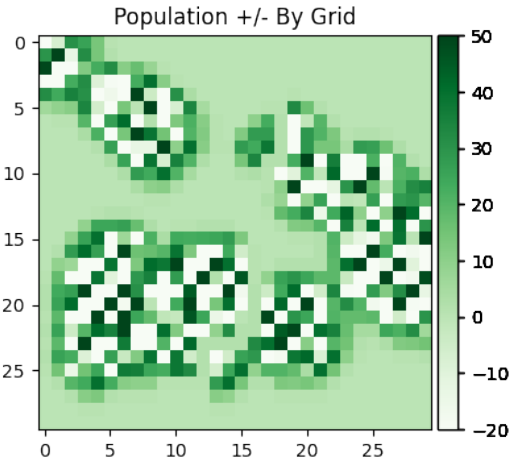
One road/ highway was added to clumped population world and go through the gap between population clusters. It was used as the world for the third scenario analysis. We simulated the system for ten generations and used the same global setting as the previous simulations. The results show that the number of individuals killed by road traffic, population density, and the emigrant population was similar to the first scenario. It suggests even if the road was not developed against the population directly, it would also affect the population density with the development of the population.

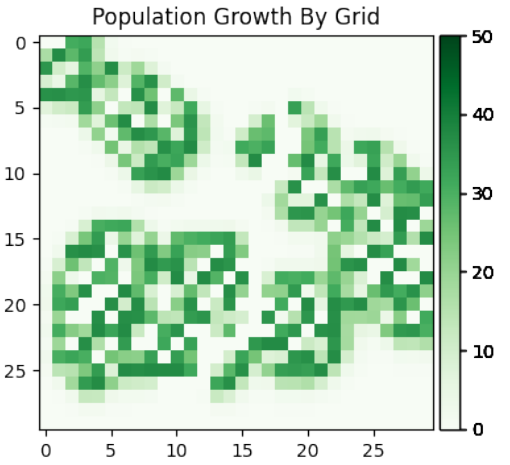
[63]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,randomSeed,patternMode,gapSpace,
highwayClass)

# constant for road 1
probAsEndOfRoad = 0.6
probForDirection = [0.2,0.2,0.6]
randomSeedHighway = 20

world.generatePatternWithHghway(probAsEndOfRoad,probForDirection,randomSeedHighway)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 10


simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
carryingCapacity,numberOfGenrations,world.patternWithHighway)
visualization.display2DArray(simulation.
 ↪worldWithoutPopulation,populationType,highwayClass)
```

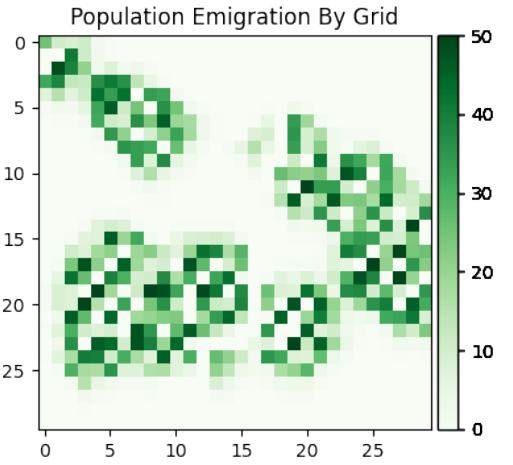


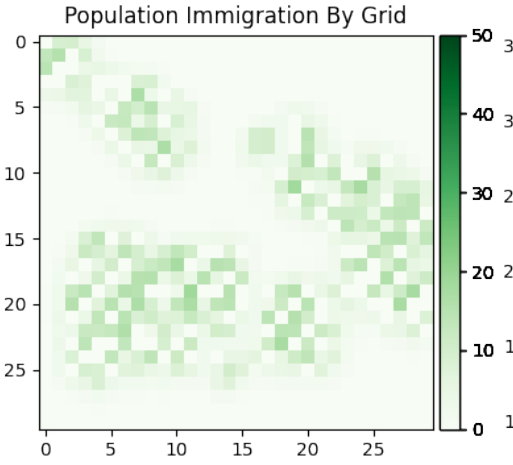
[64]:
```
simulation.runWithHighway()
```

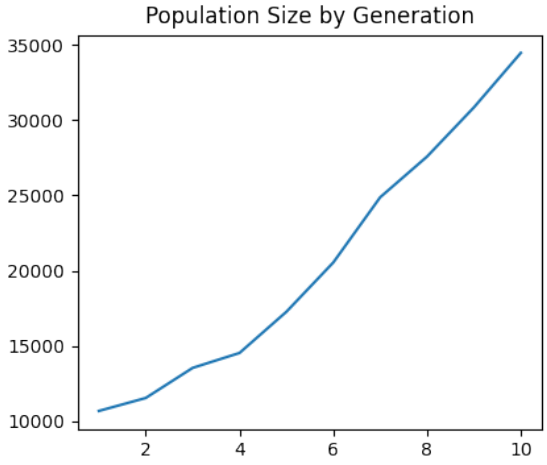

Population Density By Grid

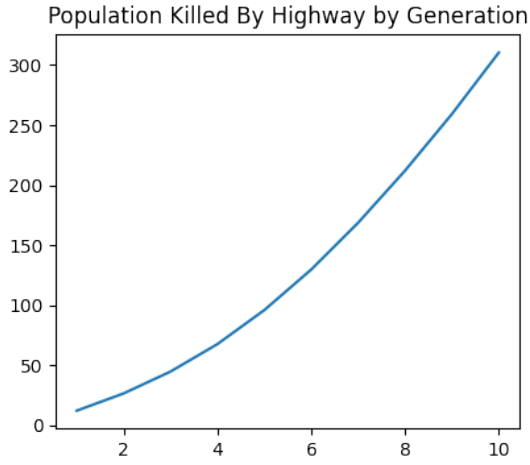

Population +/- By Grid

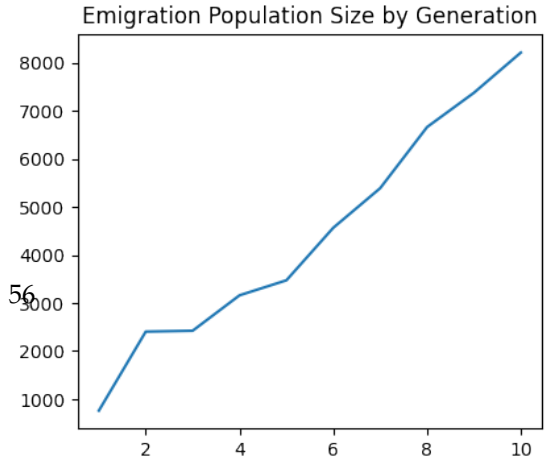

Population Growth By Grid


Population Emigration By Grid


Population Immigration By Grid

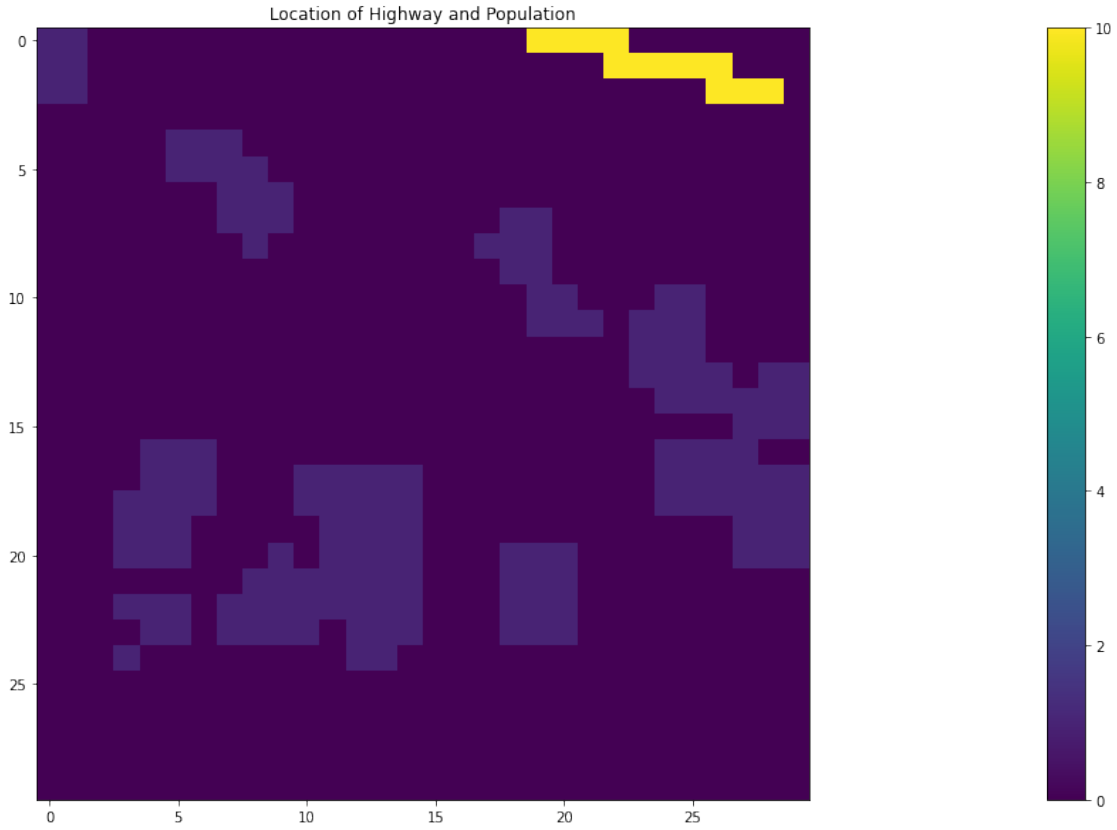

Population Size by Generation


Population Killed By Highway by Generation

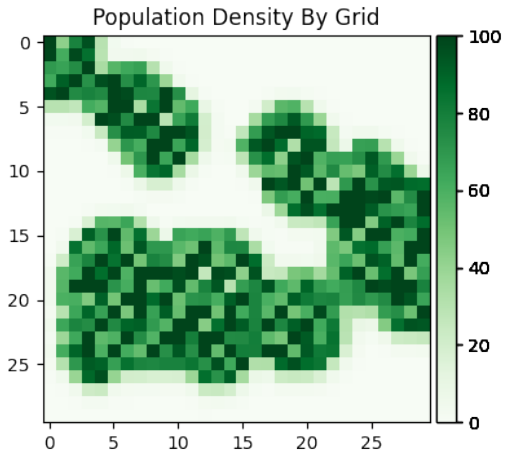

Emigration Population Size by Generation

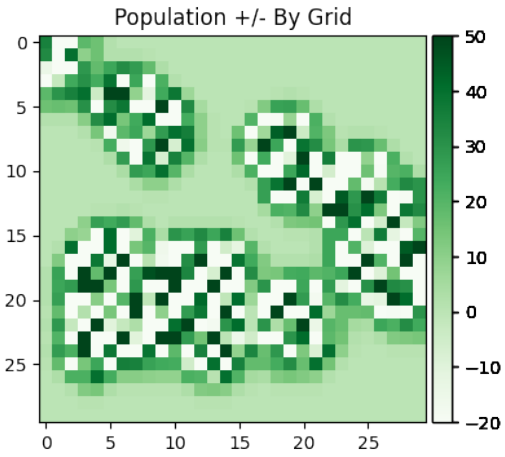
One road/ highway was added to clumped population world and located at the corner of the world. It was used as the world for the fourth scenario analysis. We simulated the system for ten generations and used the same global setting as the previous simulations. The results show that no individual was killed by road traffic. Small population clusters were merged into two big patches by corridors. Larger population size and emigrant population were observed at the end of the simulation compared to the previous scenarios. It suggests that if the developed road is not on the path of the dispersal and the population habitat, it will not affect the population dynamics.

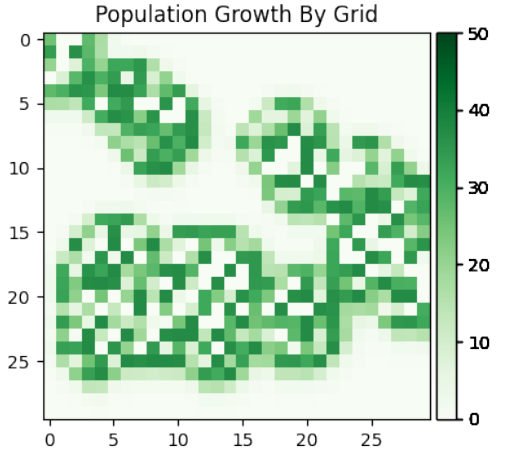
[65]:
```
#constant for world
initialProbOccupiedByPopulation = 0.1
numberOfPopulationTypes = 3
percentageOfPopulationTypes = [0.3,0.3,0.4]
widthLengthOfWorld = 30
randomSeed = 100
patternMode = "clump"
gapSpace = 5
highwayClass = 10

world = World(initialProbOccupiedByPopulation,numberOfPopulationTypes,
widthLengthOfWorld,percentageOfPopulationTypes,randomSeed,patternMode,gapSpace,
highwayClass)

# constant for road 1
probAsEndOfRoad = 0.6
probForDirection = [0.2,0.5,0.3]
randomSeedHighway = 20

world.generatePatternWithHghway(probAsEndOfRoad,probForDirection,randomSeedHighway)

# constant for simulation
randomSeedForBreeding = 120
populationType = 1
birthRate = 1.7
deathRateNatural = 0.2
deathRateMigration = 0.5
dispersalSpeed = 0.25
carryingCapacity = 100
numberOfGenrations = 10


simulation = Simulation(widthLengthOfWorld,randomSeedForBreeding,highwayClass,
populationType,birthRate,deathRateNatural,deathRateMigration,dispersalSpeed,
carryingCapacity,numberOfGenrations,world.patternWithHighway)
visualization.display2DArray(simulation.
 ↪worldWithoutPopulation,populationType,highwayClass)
```

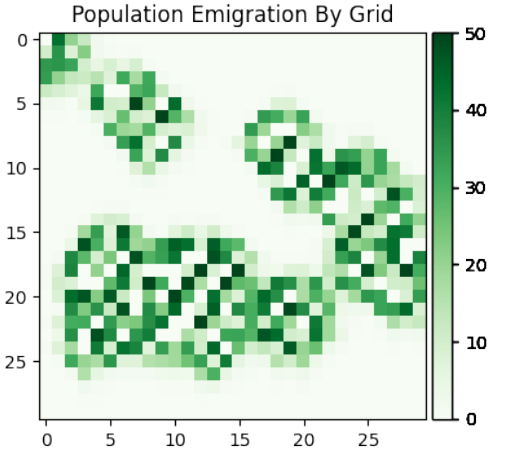


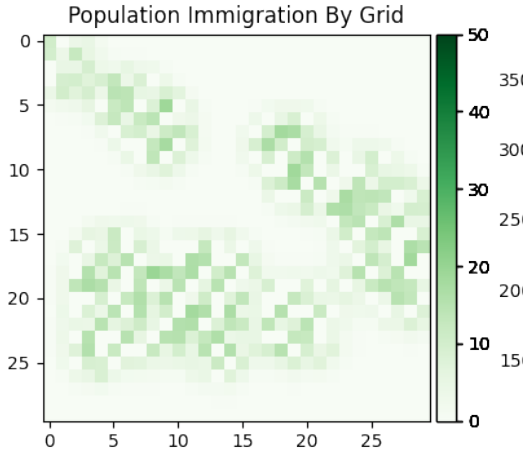
[66]:
```
simulation.runWithHighway()
```

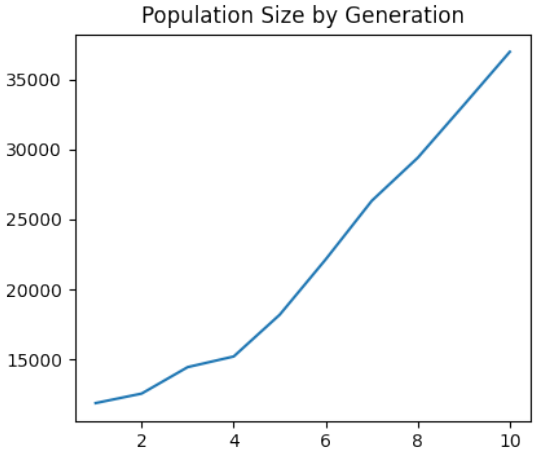

Population Density By Grid

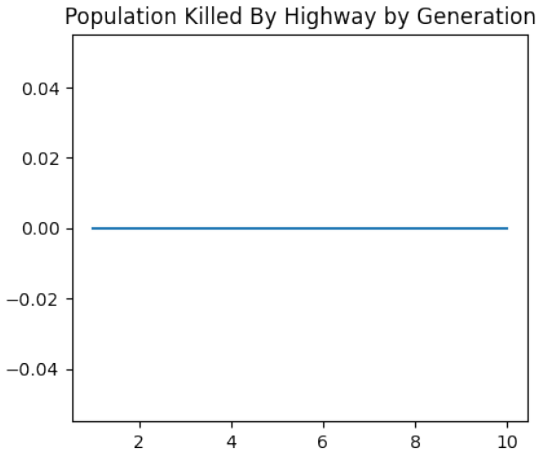

Population +/- By Grid

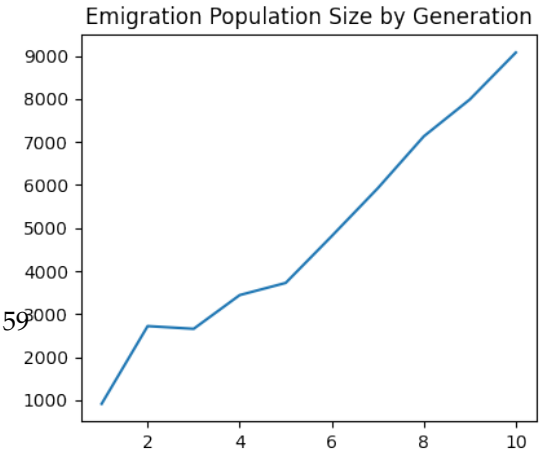

Population Growth By Grid


Population Emigration By Grid


Population Immigration By Grid


Population Size by Generation


Population Killed By Highway by Generation


Emigration Population Size by Generation

4 Predator-prey Model and Simulation

4.1 Conceptual Model

4.1.1 Classic predator-prey

Inspired by Lotka-Volterra equations,

$$f(x,y) = \begin{cases} \frac{dx}{dt} = \alpha x - \beta xy \\ \frac{dy}{dt} = \theta xy - \gamma y \end{cases} \tag{1}$$

where x is the number of preys, y is the number of predetors. $\frac{dx}{dt}$ and $\frac{dy}{dt}$ represent the instantaneous growth rates of the two populations. Predator-prey differential equation system describes the interactions between predator and prey.

$$\frac{dx}{dt} = \alpha x - \beta xy$$

The prey are assumed to have an unlimited food supply and to reproduce exponentially, unless subject to predation; this exponential growth is represented in the equation above by the term αx. The rate of predation upon the prey is assumed to be proportional to the rate at which the predators and the prey meet, this is represented above by βxy. If either x or y is zero, then there can be no predation. With these two terms the equation above can be interpreted as follows: the rate of change of the prey's population is given by its own growth rate minus the rate at which it is preyed upon.

$$\frac{dy}{dt} = \theta xy - \gamma y$$

For predator equation, δxy represents the growth of the predator population. (Note the similarity to the predation rate; however, a different constant is used, as the rate at which the predator population grows is not necessarily equal to the rate at which it consumes the prey). The term γy represents the loss rate of the predators due to either natural death or emigration, it leads to an exponential decay in the absence of prey. Hence the equation expresses that the rate of change of the predator's population depends upon the rate at which it consumes prey, minus its intrinsic death rate.

4.1.2 Modern Reaction-diffusion on Predator-prey

According to [8], a predator-prey interaction in a reaction diffusion system is as :

$$f(x) = \begin{cases} \frac{du}{dt} = \delta_1 \Delta u + ru(1 - \frac{u}{w}) - pvh(ku) \\ \frac{dv}{dt} = \delta_2 \Delta v + qvh(ku) - sv \end{cases} \tag{2}$$

where u($\hat{x}$,t) and v($\hat{x}$,t) are the population densities of prey and predator at time t and position $\hat{x}$. Δ is the usual Laplacian operator in $d \leq 3$ space dimensions and the parameters δ_1, δ_2, r, w, p, k, q and s are strictly positive.

The 'functional response' h($\cdot$) is assumed to be a C^2 function satisfying the following conditions:

$$(i) h(0) = 0$$

$$(ii)\ \lim_{x \to \text{inf}} h(x) = 1$$

$$(iii) h()\text{is strictly increasing on} [0, \text{inf})$$

The functional response represents the prey consumption rate per predator as a fraction of the maximal consumption rate p. The constant k determines how fast the consumption rate saturates as the prey density increases. q and r denote maximal per capita birth rates of predator and prey, respectively, s is the per capita predator death rate, and w the prey-carrying capacity

4.2 Implementation

4.2.1 Parameters

Global variables (Constant):

Variable	Definition
δ_1	Growth rate for prey Laplace
δ_2	Growth rate for predator Laplace
k	How fast the consumption rate saturates as the prey density increases
q	Maximal per capita birth rates of predator
r	Maximal per capita birth rates of prey
s	Per capita predator death rate
w	Prey-carrying capacity
p	Prey consumption rate per predator as a fraction of the maximal consumption rate
T	Length of time
dt	Each time step

State variables:

Variable	Definition
$U[i, j]$	U: prey density at $mesh[i, j]$
$V[i, j]$	V: predator density at $mesh[i, j]$

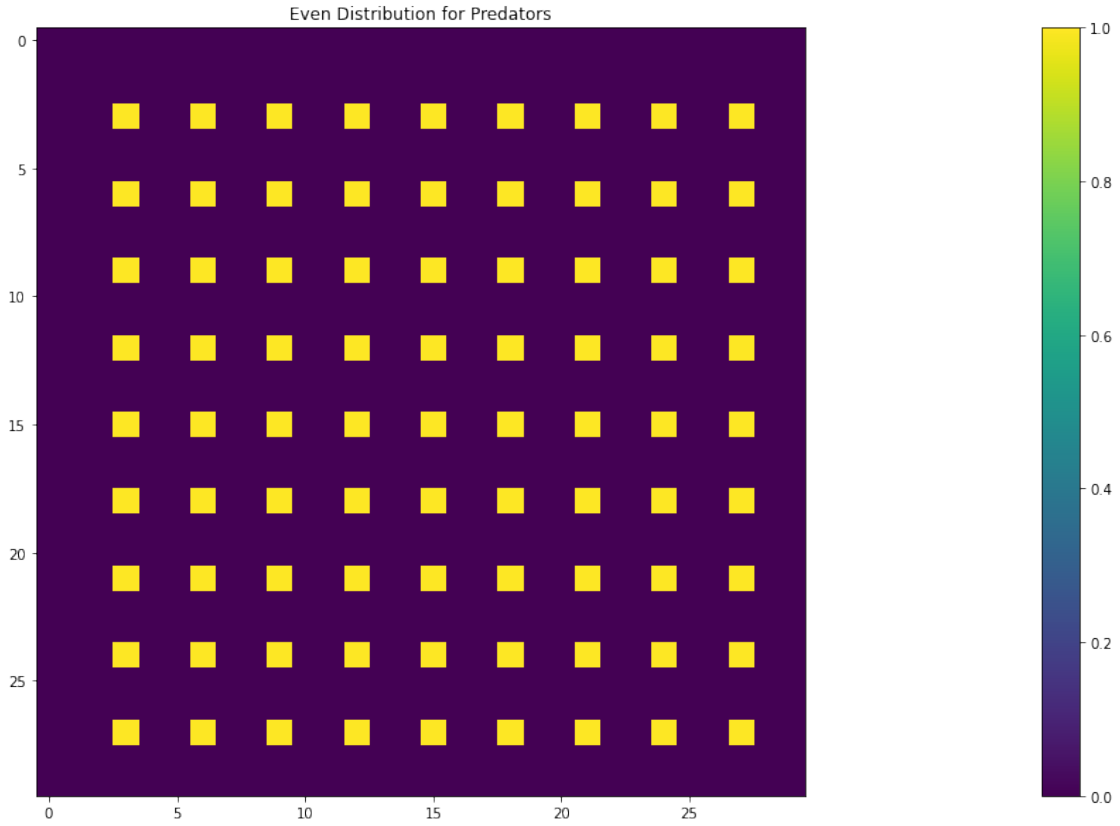
4.2.2 Data generation

In the nature, the predators are more often EVEN distributed due to the territorial awareness and competitions, while the preys are often CLUMPED distributed due to the gregarious lifestyle and resources

Initialize EVEN distribution for predators due to the territorial awareness and competitions

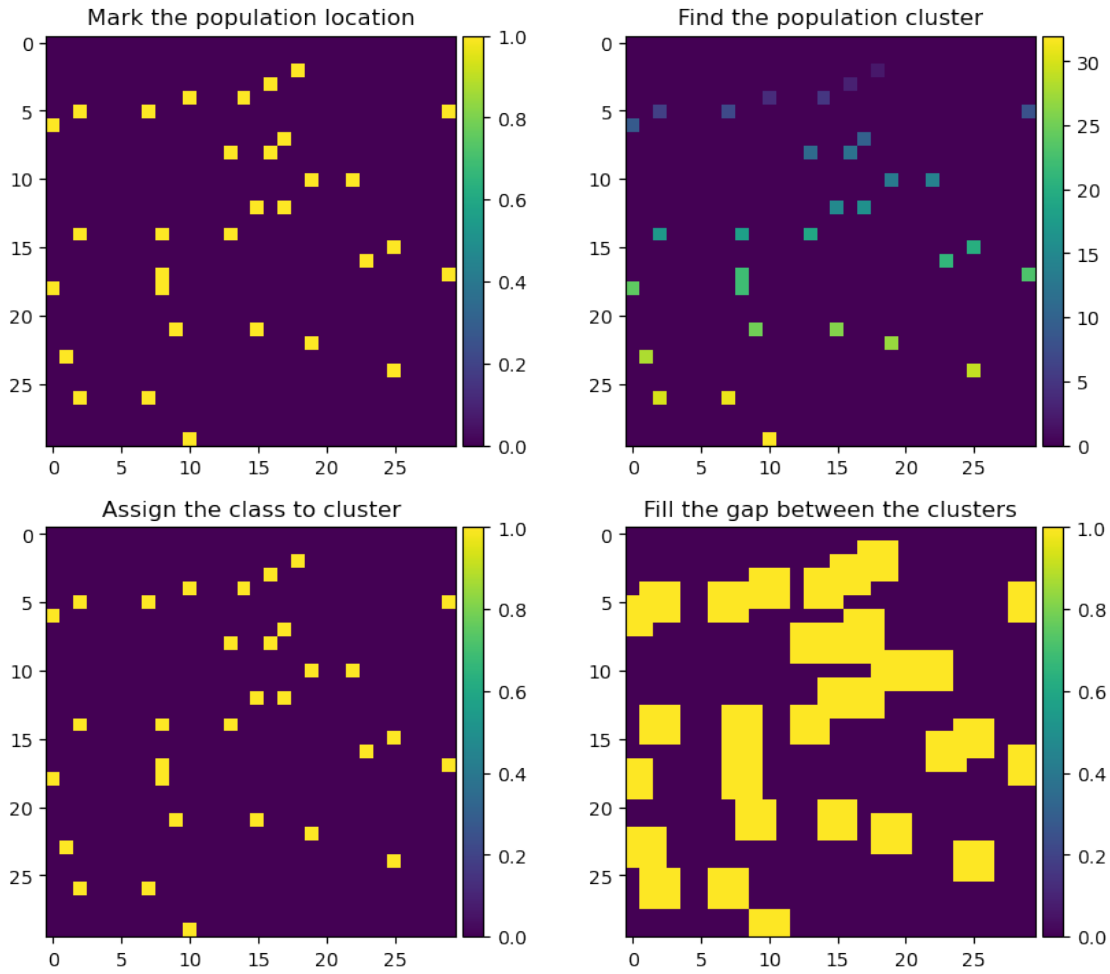
[53]:
```
import os
import random
import numpy as np
import dataForInput
import visualization
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

```
import pandas as pd
# set parameters
numberOfClasses = 2
areasOfClasses = [1]
widthLengthOfWorld = 30
initialProb = 0.03
randomSeedForPopulation = 7
randomSeedForHighway = 20
gapSpace = 5
probAsEndOfRoad = 0.5
probForDirection = [0.2,0.2,0.6]
highwayClass = 10

I = dataForInput.initialWorld(widthLengthOfWorld)
E = dataForInput.genrateEvenDistribution(I, 1, gapSpace = 3,widthLengthOfWorld =␣
 ↪widthLengthOfWorld)
visualization.displayPattern(E,"Even Distribution for Predators")
```



Initialize CLUMPED distribution for preys due to the the gregarious lifestyle and clumped resources

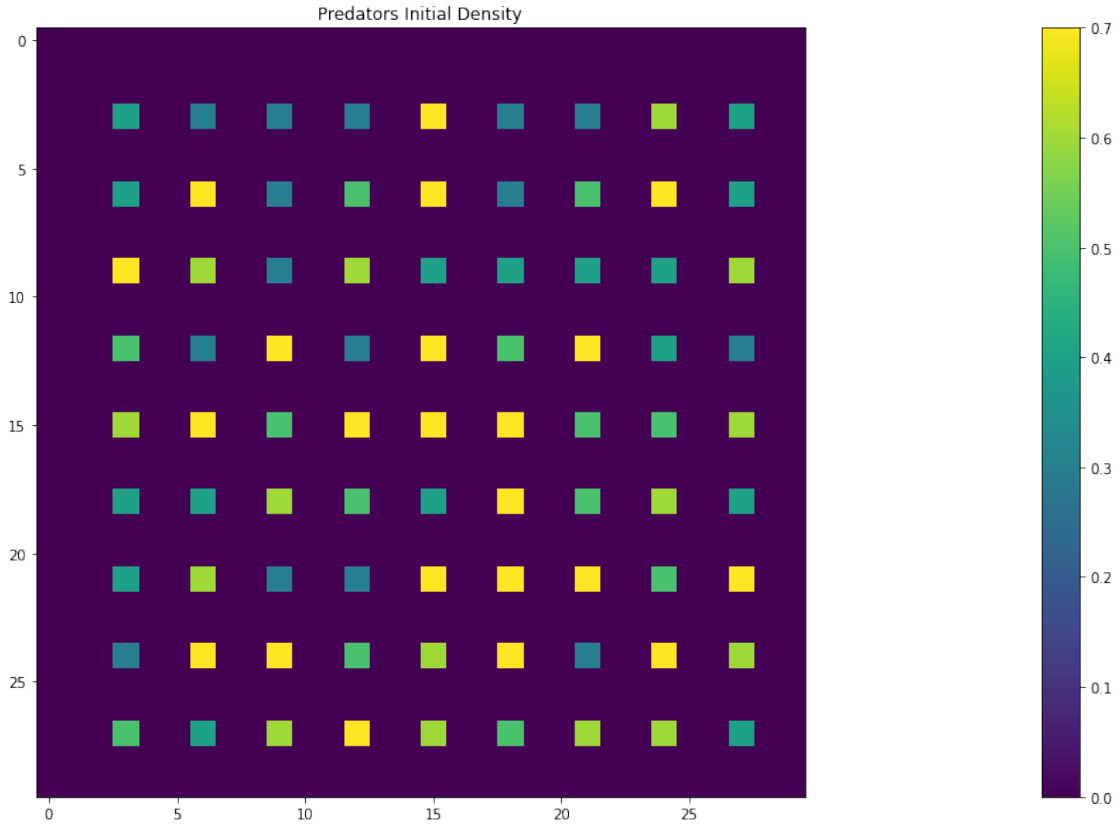
[54]:
```
C = dataForInput.locatePopulationOnProb(I,␣
↪randomSeedForPopulation,widthLengthOfWorld,initialProb)
D = dataForInput.findClusterByNeighbor(C,widthLengthOfWorld)
clusterWorld = dataForInput.assignClassToCluster(areasOfClasses, D, widthLengthOfWorld)
F = dataForInput.fillGapOnUnmarked(clusterWorld,widthLengthOfWorld)

fig,ax1,ax2,ax3,ax4 = visualization.initial2DArrayForGeneratedPattern()
visualization.update2DArrayForGeneratedPattern(fig,ax1,C)
visualization.update2DArrayForGeneratedPattern(fig,ax2,D)
visualization.update2DArrayForGeneratedPattern(fig,ax3,clusterWorld)
visualization.update2DArrayForGeneratedPattern(fig,ax4,F)
visualization.updateCharts(fig)
```

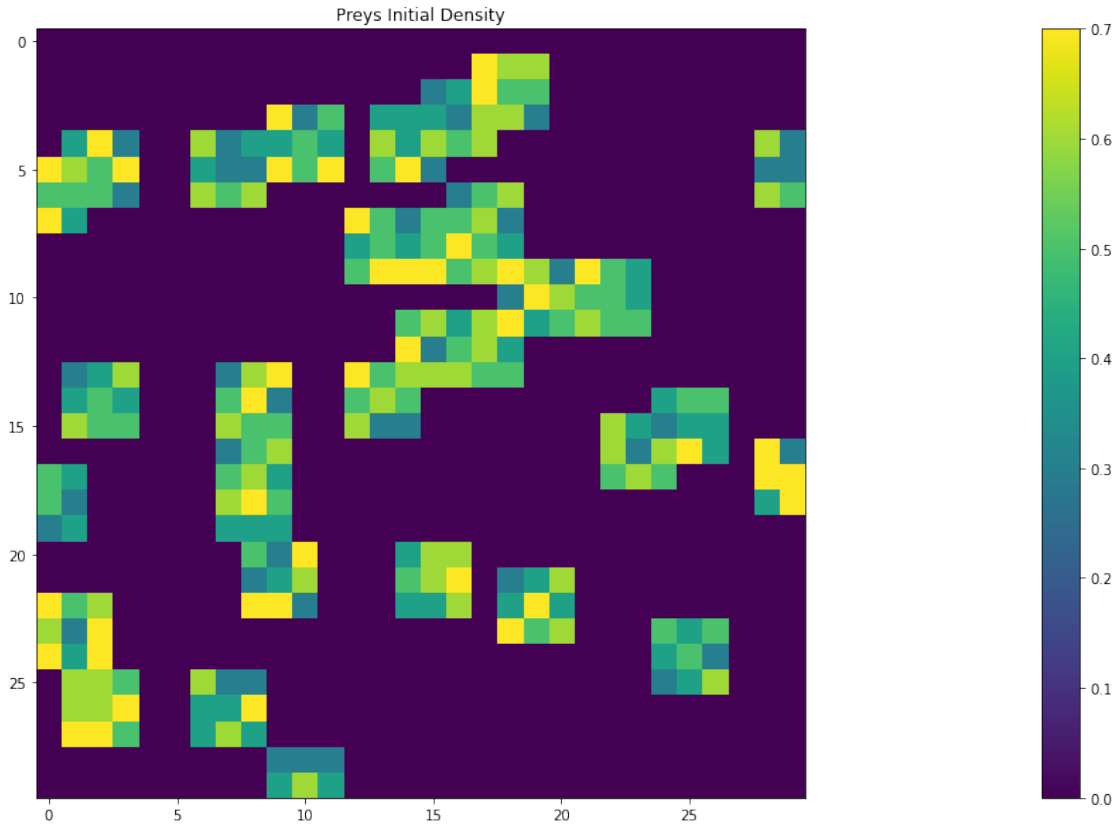


Generate population density for predators and preys by randomly assigning a density between 0.3 - 0.8

[120]:
```
U = dataForInput.initialWorld(widthLengthOfWorld)
V = dataForInput.initialWorld(widthLengthOfWorld)
```

```
for i in range(widthLengthOfWorld):
    for j in range(widthLengthOfWorld):
        if E[i,j] == 1:
            # generate an initial population density between 0.3 - 0.8
            V[i,j] = np.random.randint(low = 3, high = 8) / 10
        if F[i,j] == 1:
            U[i,j] = np.random.randint(low = 3, high = 8) / 10
visualization.displayPattern(V,"Predators Initial Density")
visualization.displayPattern(U,"Preys Initial Density")
```





Define the utility functions for Diffusion-Reaction system

[13]:
```
def laplacian(Z):
    Ztop = Z[0:-2, 1:-1]
    Zleft = Z[1:-1, 0:-2]
    Zbottom = Z[2:, 1:-1]
    Zright = Z[1:-1, 2:]
    Zcenter = Z[1:-1, 1:-1]
    return (Ztop + Zleft + Zbottom + Zright -
            4 * Zcenter) / dx**2
def h(x):
    return 1 - np.exp(-x)
```

set the parameters for Difussion-Reaction models

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
delta1 = 5e-6 # growth rate for prey Laplace
delta2 = 6e-6 # growth rate for predator Laplace
k = 1.05  # k determines how fast the consumption rate saturates as the prey density␣
 ↪increases
q = 2 #  maximal per capita birth rates of predator
```

```
r = 1.5 #  maximal per capita birth rates of prey
s = 0.3 # per capita predator death rate
w = 0.85 # prey-carrying capacity
p = 1.1 # represents the prey consumption rate per predator as a fraction of the␣
 ↪maximal consumption rate p.

T = 9
dt = 1/64
n = int(T/dt)
size = 30
dx = 2. / size
```

4.3 Visualization and Verification

The dimension of the world as 30, belows visualization shows the emmigration pattern of predators and preys in 9 generations, and the modeling is varified.

Below is the utility functions to calculate the population for a generation and to implement the visualizations

[102]:
```
def calc_population(world, population_per_cell = 100):
    res = world.copy()
    return [population_per_cell * world[i,j] for i in range(widthLengthOfWorld) for j␣
 ↪in range(widthLengthOfWorld)]

def draw_population(world_list):
    total_population =[]
    for i in range(len(world_list)):
        total_population.append(sum(calc_population(world_list[i],100)))
    temp = pd.DataFrame({'Generations':list(range(len(world_list))), 'Total␣
 ↪Population': total_population}).plot(x = 'Generations', y ='Total␣
 ↪Population',figsize = (10,5))
```

Below is the main function to implement the Diffusion-Reaction system

[114]:
```
def Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, r, s, w, p):
    U = U.copy()
    V = V.copy()
    U_list = []
    V_list = []
    step_plot = n // T
    # We simulate the PDE with the finite difference method.
    for i in range(n):
        # We compute the Laplacian of u and v.
        deltaU = laplacian(U)
        deltaV = laplacian(V)
        # We take the values of u and v inside the grid.
        Uc = U[1:-1, 1:-1]
        Vc = V[1:-1, 1:-1]
        # We update the variables.
```

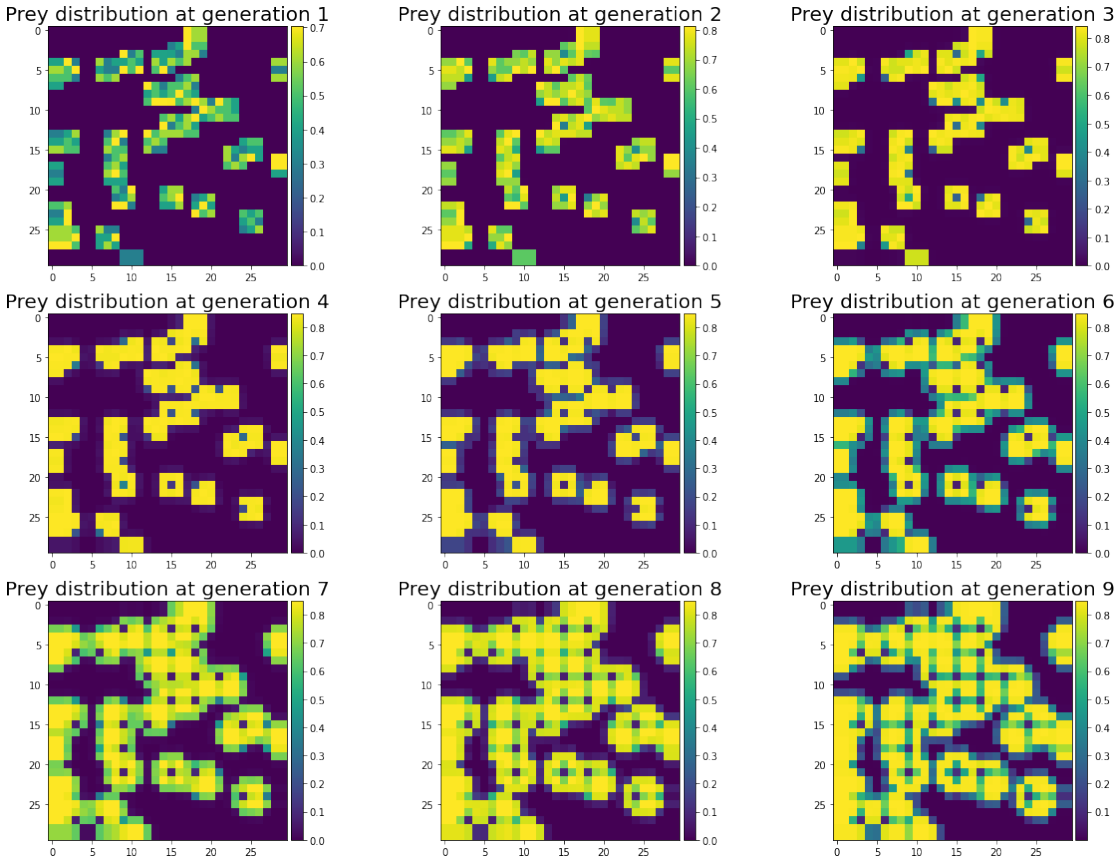
```
        oldU = U[1:-1, 1:-1].copy()
        U[1:-1, 1:-1] = Uc + dt * (delta1 * deltaU + r*Uc*(1-Uc/w) - p*Vc*h(k*Uc)) #␣
↪ignore h
        newU = Uc + dt * (delta1 * deltaU + r*Uc*(1-Uc/w) - p*Vc*h(k*Uc))
        V[1:-1, 1:-1] = Vc + dt * (delta2 * deltaV + q*Vc*h(k*Uc) - s*Vc)

        for Z in (U, V):
            Z[0, :] = Z[1, :]
            Z[-1, :] = Z[-2, :]
            Z[:, 0] = Z[:, 1]
            Z[:, -1] = Z[:, -2]

        # We plot the state of the system at
        # 9 different times.
        if i % step_plot == 0 and i < 9 * step_plot:
            U_list.append(U.copy())
            V_list.append(V.copy())

    return U_list, V_list
```

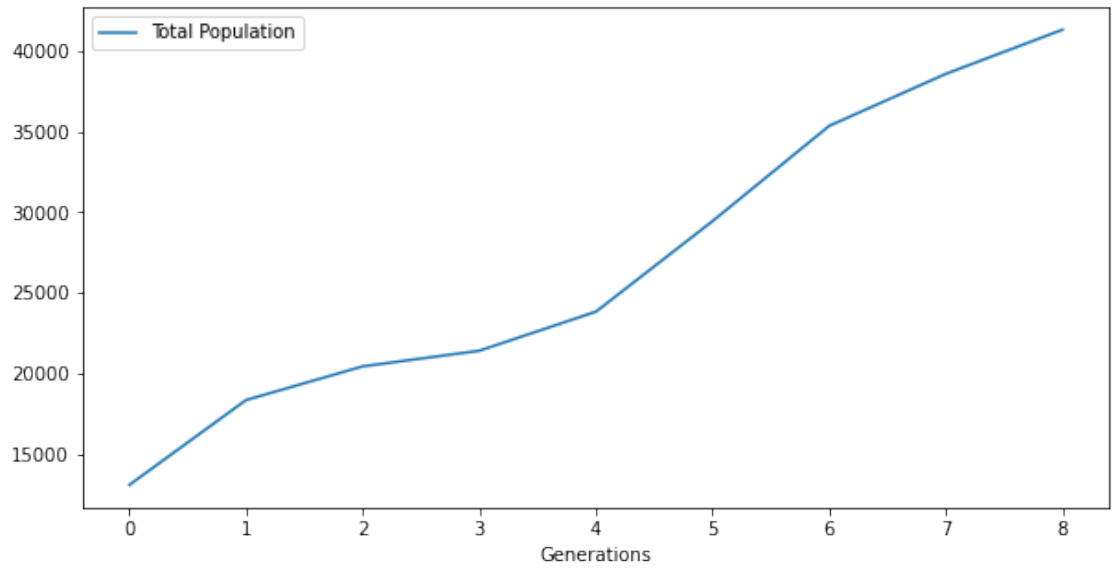
[122]:
```
U_list, V_list = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, r, s, w, p)
```

Below shows the visualization for preys.

The prey emigration pattern is emerged from its initial clumped distribution, and some of the preys are eaten by the predators, which can be observed from those dots in the below plots.

[123]:
```
fig, axes = plt.subplots(3,3, figsize = (20,15))

for i in range(9):
    ax = axes.flat[i]
    ax.set_title('Prey distribution at generation {}'.format(i + 1), size = 20)
    show_obj = ax.imshow(U_list[i], interpolation = 'none')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size = "5%", pad = 0.05)
    fig.colorbar(show_obj, cax = cax, orientation = 'vertical')
draw_population(U_list)
```

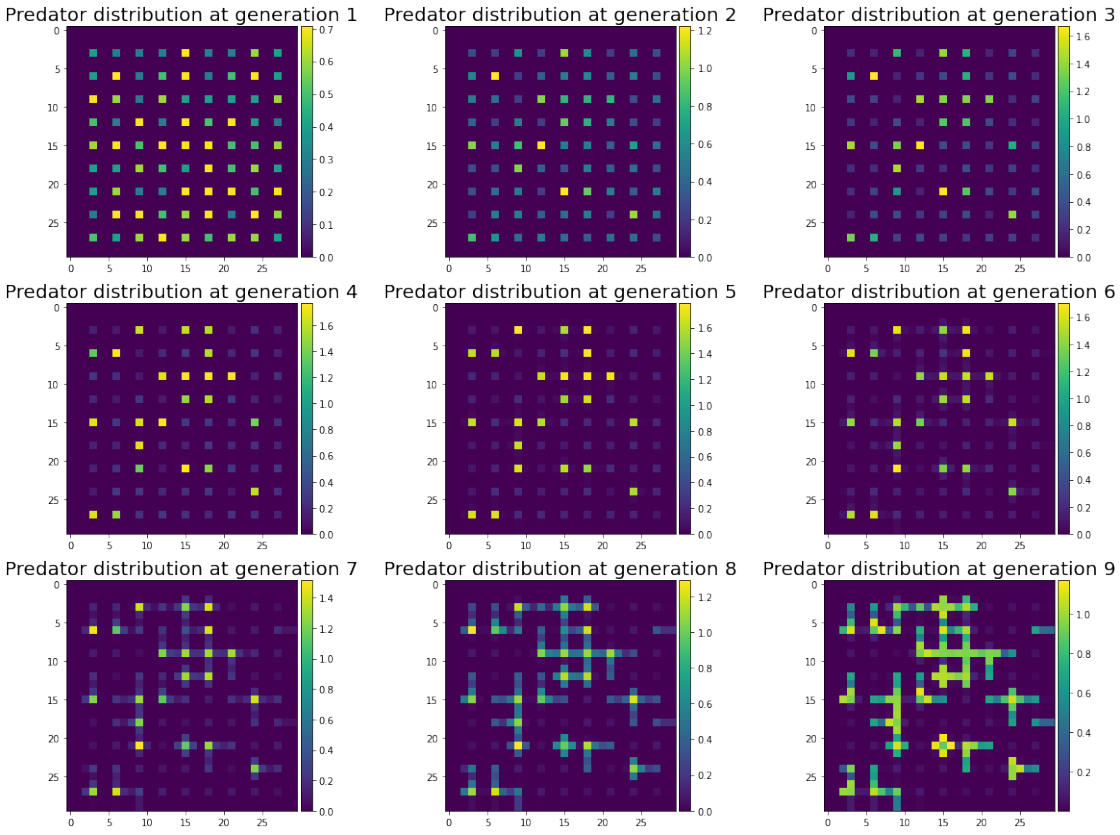

Prey distribution at generation 1
Prey distribution at generation 2
Prey distribution at generation 3
Prey distribution at generation 4
Prey distribution at generation 5
Prey distribution at generation 6
Prey distribution at generation 7
Prey distribution at generation 8
Prey distribution at generation 9

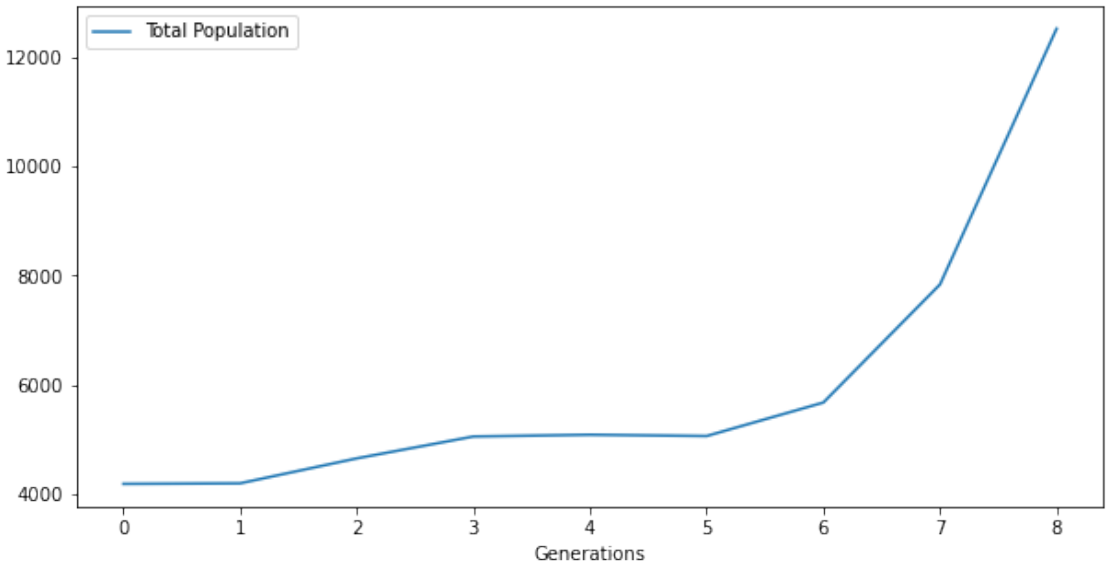

Total Population
40000
35000
30000
25000
20000
15000
0
1
2
3
4
5
6
7
8
Generations

Below is the visualization for predators

After 9 generations' immigration, the pattern of the predators are heavily relied on the prey distribution. There are some predator groups dying out due to the lack of preys in their locations. There are also many predator groups keeping growing based on the plentiful amount of the preys.

[124]:
```
fig, axes = plt.subplots(3,3, figsize = (20,15))

for i in range(9):
    ax = axes.flat[i]
    ax.set_title('Predator distribution at generation {}'.format(i + 1), size = 20)
    show_obj = ax.imshow(V_list[i], interpolation = 'none')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size = "5%", pad = 0.05)
    fig.colorbar(show_obj, cax = cax, orientation = 'vertical')
draw_population(V_list)
```





4.4 Sensitivity Analysis and Validation

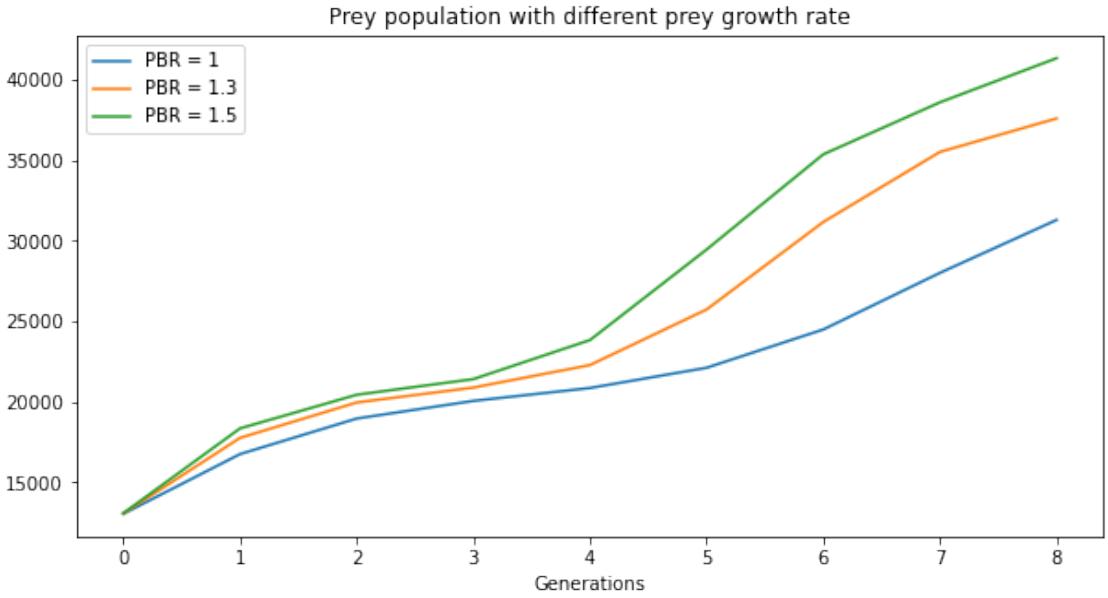
A visualization component was developed to show the results of the sensitivity analysis by multiple-line chart with capable of showing and comparing the results of three experiments in one chart.

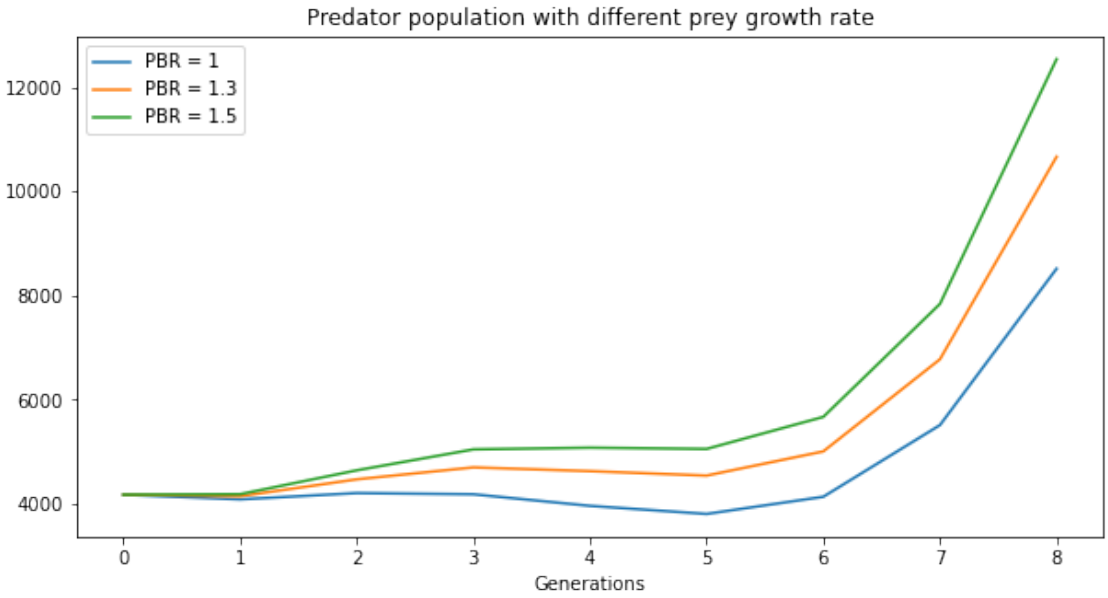
Prey birth rate (1, 1.3, 1.5) The birth rate was set as 1, 1.3, 1.5, while other parameters were the same for the experiments. The results show that the experiment with a birth rate of 1.5 has the highest prey population and the highest prey population. In contrast, the experiment with a birth rate of 1 has the lowest ones.This makes sense because predator population needs to be feuled by prey population.

[125]:
```
U_list_1, V_list_1 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, 1, s,␣
 ↪w, p)
U_list_2, V_list_2 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, 1.3, s,␣
 ↪w, p)
U_list_3, V_list_3 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, 1.5, s,␣
 ↪w, p)
temp = pd.DataFrame({'Generations':list(range(len(U_list))),
                     'PBR = 1': [sum(calc_population(U_list_1[i],100)) for i in␣
 ↪range(len(U_list))],
                     'PBR = 1.3': [sum(calc_population(U_list_2[i],100)) for i in␣
 ↪range(len(U_list))],
                     'PBR = 1.5': [sum(calc_population(U_list_3[i],100)) for i in␣
 ↪range(len(U_list))]})

temp2 = pd.DataFrame({'Generations':list(range(len(U_list))),
                     'PBR = 1': [sum(calc_population(V_list_1[i],100)) for i in␣
 ↪range(len(U_list))],
```

```
                    'PBR = 1.3': [sum(calc_population(V_list_2[i],100)) for i in␣
 ↪range(len(U_list))],
                    'PBR = 1.5': [sum(calc_population(V_list_3[i],100)) for i in␣
 ↪range(len(U_list))]})
temp.plot(x = 'Generations',figsize = [10,5],title = 'Prey population with different␣
 ↪prey growth rate')
temp2.plot(x = 'Generations',figsize = [10,5],title = 'Predator population with␣
 ↪different prey growth rate')
```

[125]: <AxesSubplot:title={'center':'Predator population with different prey growth rate'}, xlabel='Generations'>



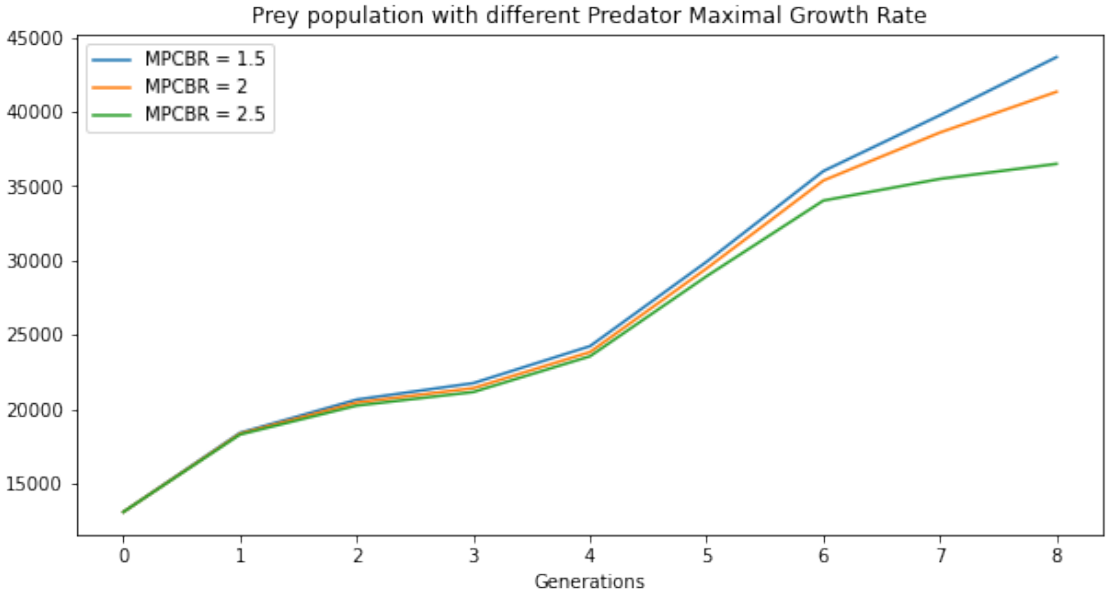


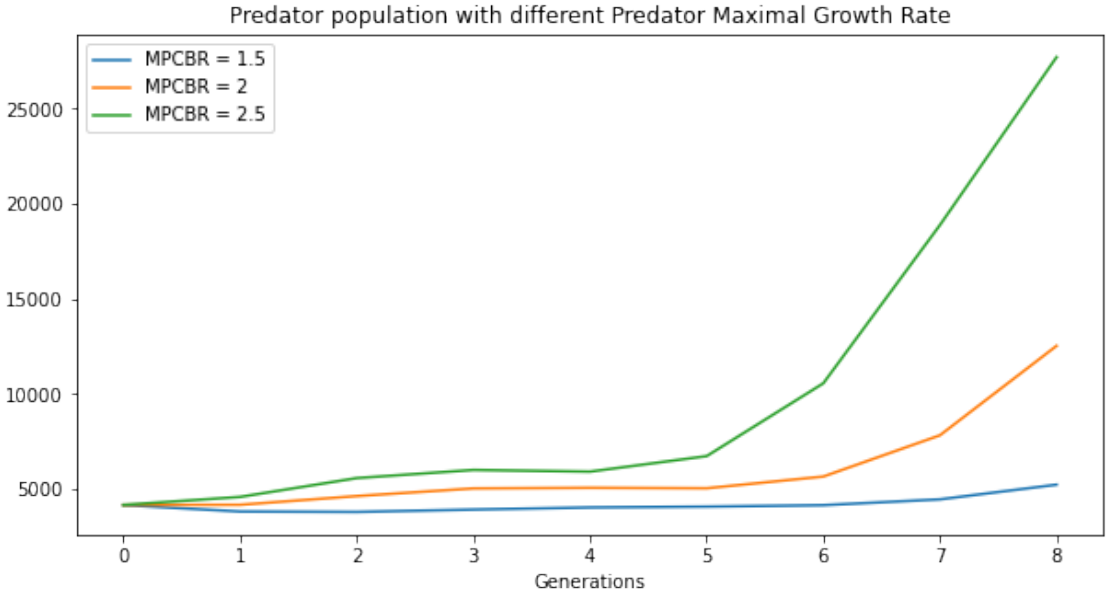
q :maximal per capita birth rates of predator (1.5, 2, 2.5) The maximal per capita birth rates of predator was set as 1.5, 2, 2.5, while other parameters were the same for the experiments. The results show that the experiment with a maximal predator birth rate of 2.5 has the highest predator population and the lowest prey population.This is reasonable because the more predator will eat more preys, thus the prey population will keep deminishing.

[127]:
```
U_list_4, V_list_4 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, 1.5, r, s,␣
 ↪w, p)
U_list_5, V_list_5 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, 2, r, s,␣
 ↪w, p)
U_list_6, V_list_6 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, 2.5, r, s,␣
 ↪w, p)
temp3 = pd.DataFrame({'Generations':list(range(len(U_list))),
                      'MPCBR = 1.5': [sum(calc_population(U_list_4[i],100)) for i in␣
 ↪range(len(U_list))],
                      'MPCBR = 2': [sum(calc_population(U_list_5[i],100)) for i in␣
 ↪range(len(U_list))],
                      'MPCBR = 2.5': [sum(calc_population(U_list_6[i],100)) for i in␣
 ↪range(len(U_list))]})

temp4 = pd.DataFrame({'Generations':list(range(len(U_list))),
                      'MPCBR = 1.5': [sum(calc_population(V_list_4[i],100)) for i in␣
 ↪range(len(U_list))],
                      'MPCBR = 2': [sum(calc_population(V_list_5[i],100)) for i in␣
 ↪range(len(U_list))],
                      'MPCBR = 2.5': [sum(calc_population(V_list_6[i],100)) for i in␣
 ↪range(len(U_list))]})
```

```
temp3.plot(x = 'Generations',figsize = [10,5],title = 'Prey population with different␣
 ↪Predator Maximal Growth Rate')
temp4.plot(x = 'Generations',figsize = [10,5],title = 'Predator population with␣
 ↪different Predator Maximal Growth Rate')
```

[127]: <AxesSubplot:title={'center':'Predator population with different Predator Maximal Growth Rate'}, xlabel='Generations'>

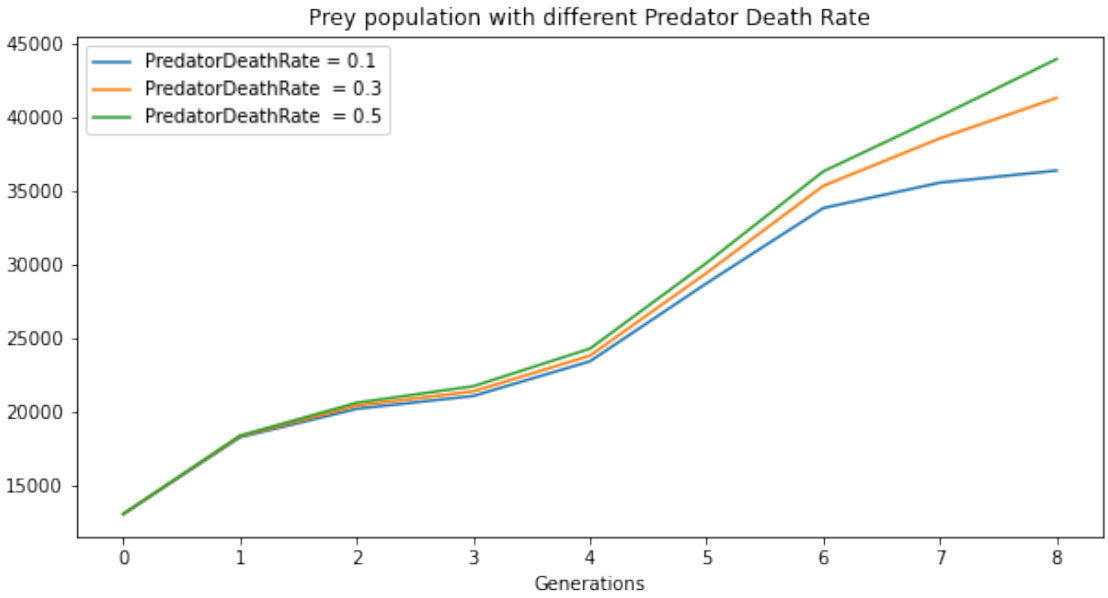


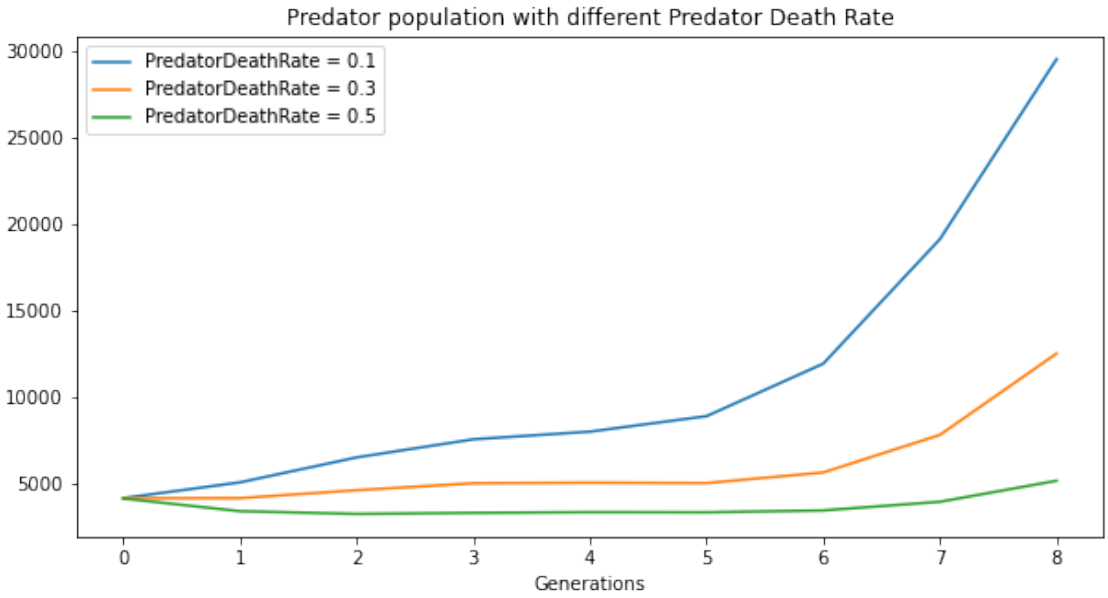


s: Death Rate for predator during migration, (0.1, 0.3, 0.5) The death rate for predators during immigration was set as 0.1, 0.3, 0.5, while other parameters were the same for the experiments. The results show that the experiment with a death rate of 0.5 has the highest prey population and the lowest predator population.This is reasonable because the more predators died during immigration, the more chance the preys can live.

[128]:
```
U_list_7, V_list_7 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, r, 0.1,␣
 ↪w, p)
U_list_8, V_list_8 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, r, 0.3,␣
 ↪w, p)
U_list_9, V_list_9 = Reaction_Diffusion_Simulation(U, V, delta1, delta2, k, q, r, 0.5,␣
 ↪w, p)
temp5 = pd.DataFrame({'Generations':list(range(len(U_list))),
                      'PredatorDeathRate = 0.1': [sum(calc_population(U_list_7[i],100))␣
 ↪for i in range(len(U_list))],
                      'PredatorDeathRate  = 0.3':␣
 ↪[sum(calc_population(U_list_8[i],100)) for i in range(len(U_list))],
                      'PredatorDeathRate  = 0.5':␣
 ↪[sum(calc_population(U_list_9[i],100)) for i in range(len(U_list))]})

temp6 = pd.DataFrame({'Generations':list(range(len(U_list))),
                      'PredatorDeathRate = 0.1': [sum(calc_population(V_list_7[i],100))␣
 ↪for i in range(len(U_list))],
                      'PredatorDeathRate = 0.3': [sum(calc_population(V_list_8[i],100))␣
 ↪for i in range(len(U_list))],
                      'PredatorDeathRate = 0.5': [sum(calc_population(V_list_9[i],100))␣
 ↪for i in range(len(U_list))]})
temp5.plot(x = 'Generations',figsize = [10,5],title = 'Prey population with different␣
 ↪Predator Death Rate')
temp6.plot(x = 'Generations',figsize = [10,5],title = 'Predator population with␣
 ↪different Predator Death Rate')
```

[128]:
```
<AxesSubplot:title={'center':'Predator population with different Predator Death
Rate'}, xlabel='Generations'>
```





Define the below function that can calculate the number of animals(predator and prey) that got killed due to the highway

[129]:
```
def calc_killed(TargetWorld, WorldWithHighway, highwayClass ,population_per_cell =␣
↪100):
    cnt_death = 0
```

```
    for i in range(widthLengthOfWorld):
        for j in range(widthLengthOfWorld):
            if WorldWithHighway[i,j] == highwayClass:
                cnt_death += TargetWorld[i,j] * population_per_cell
    return cnt_death
```

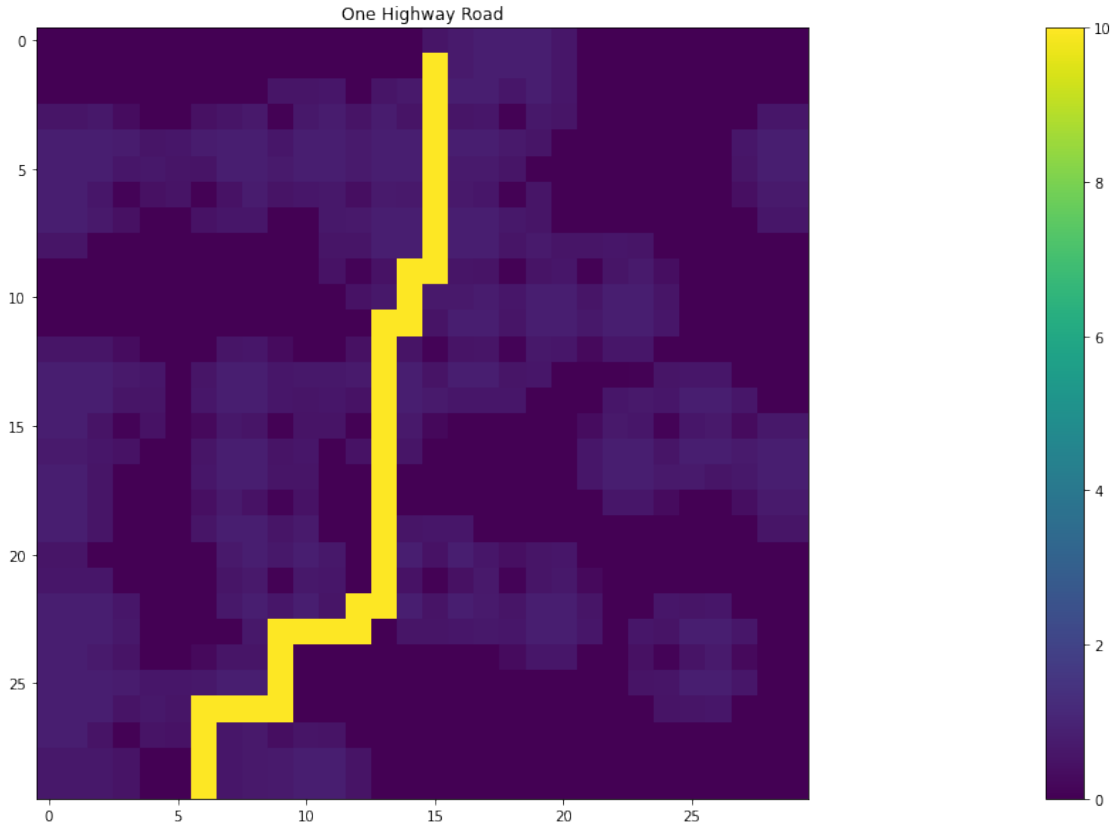
4.5 Scenario Analysis with Road Network

A visualization component was developed to show the results of the scenario analysis with the road network. Line plots were designed to monitor the progress of the simulation, as well as the number of animals (predator and preys) killed by the highway

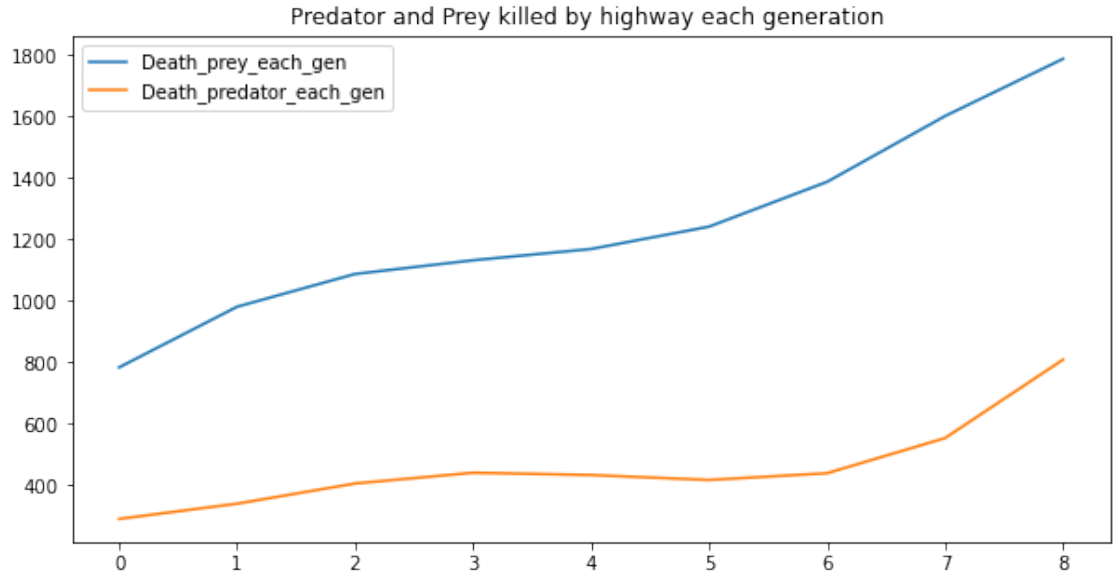
4.5.1 One highway scenario

One road was added to the predator-prey world and overlapping with a few population clusters. It was used as the world for the first scenario analysis. We simulated the system for eight generations and the results show that around 1800 preys and 800 predators were killed by road traffic over 8 generations.

[135]:
```
G = dataForInput.addHighway2World(U_list_1[8],widthLengthOfWorld, probAsEndOfRoad,␣
 ↪probForDirection,highwayClass,randomSeedForHighway)
visualization.displayPattern(G,"One Highway Road")
pd.DataFrame({'Death_prey_each_gen':[calc_killed(i, G, 10, 100) for i in U_list_1],
 'Death_predator_each_gen':[calc_killed(i, G, 10, 100) for i in V_list_1]}).
 ↪plot(figsize = [10,5],
    title= 'Predator and Prey killed by highway each generation ')
```



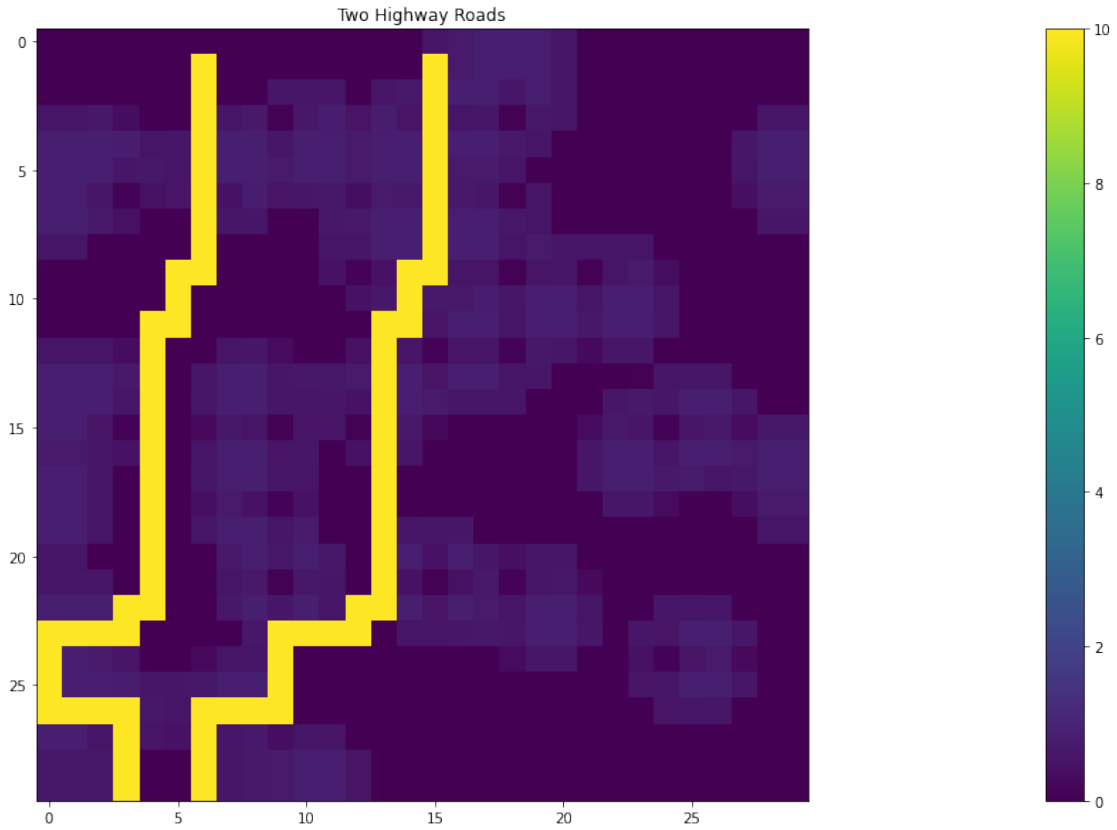
[135]: <AxesSubplot:title={'center':'Predator and Prey killed by highway each generation '}>



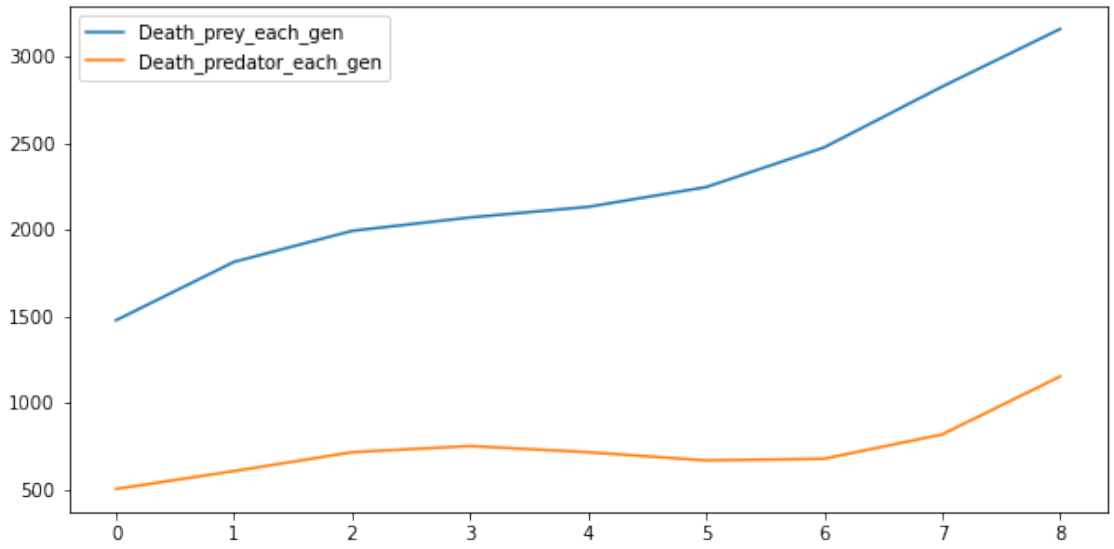
4.5.2 Two highways scenario

Two roads were added to the predator-prey population world and overlapped with more population clusters than the previous scenario. It was used as the world for the second scenario analysis. We simulated the system for eight generations and used the same global setting as the previous simulations. The results show that around 3200 preys and 1200 predators were killed by road traffic, much more than the previous scenario.

[136]:
```
probAsEndOfRoad = 0.2
probForDirection = [0.2,0.2,0.6]
randomSeedHighway = 30

G2 = dataForInput.addHighway2World(G,widthLengthOfWorld, probAsEndOfRoad,␣
 ↪probForDirection,highwayClass,randomSeedForHighway)
visualization.displayPattern(G2,"Two Highway Roads")

pd.DataFrame({'Death_prey_each_gen':[calc_killed(i, G2, 10, 100) for i in U_list_1],
 'Death_predator_each_gen':[calc_killed(i, G2, 10, 100) for i in V_list_1]}).
 ↪plot(figsize = [10,5])
```



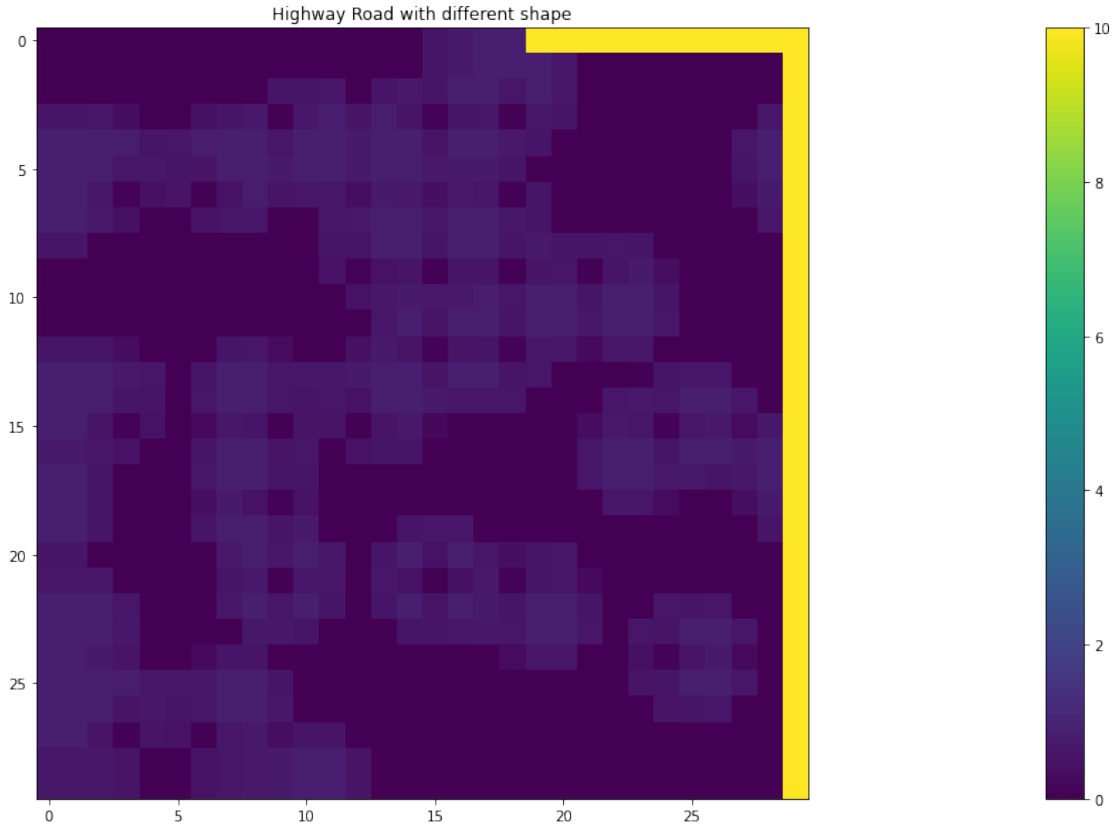
[136]: <AxesSubplot:>



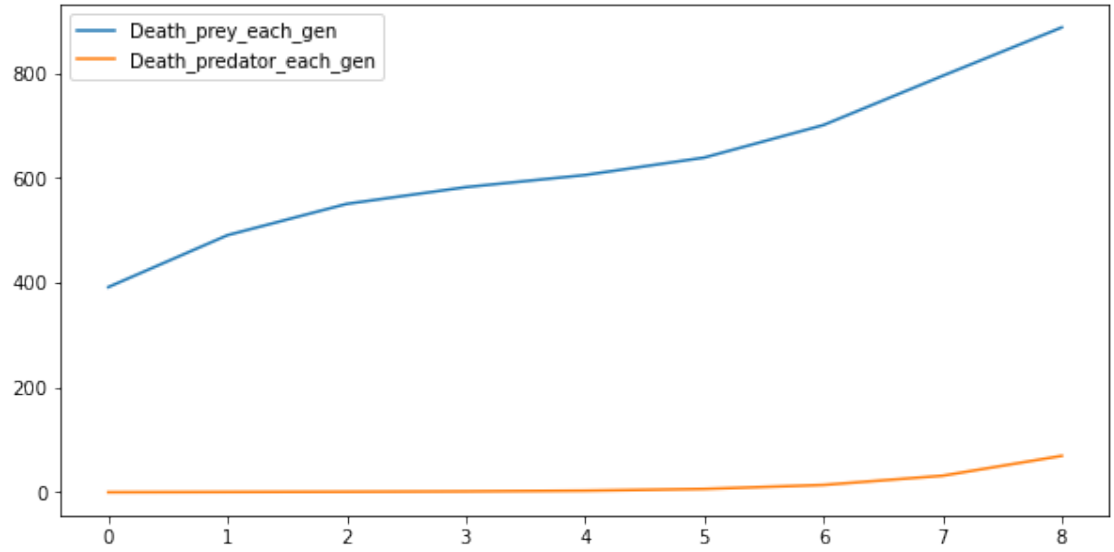
4.5.3 Different shapes of highway to circumvent the corridors

One road was added to predator-prey population world and located at the corner of the world. It was used as the world for the third scenario analysis. We simulated the system for eight generations and used the same global setting as the previous simulations. The results show that much lesser predators and preys were killed by road traffic. Larger population size and emigrant population were observed at the end of the simulation compared to the previous scenarios. Thus it suggests that if the developed road is not on the path of the dispersal and the population habitat, it will not affect the population dynamics that much.

[137]:
```
probAsEndOfRoad = 0.6
probForDirection = [0.00,0.95,0.05]
randomSeedHighway = 20

G_horizon = dataForInput.addHighway2World(U_list_1[8],widthLengthOfWorld,␣
 ↪probAsEndOfRoad, probForDirection,highwayClass,randomSeedForHighway)
visualization.displayPattern(G_horizon,"Highway Road with different shape")

pd.DataFrame({'Death_prey_each_gen':[calc_killed(i, G_horizon, 10, 100) for i in␣
 ↪U_list_1],
 'Death_predator_each_gen':[calc_killed(i, G_horizon, 10, 100) for i in V_list_1]}).
 ↪plot(figsize = [10,5])
```



[137]: <AxesSubplot:>



5 Multi-Species Model and Simulation

5.1 Conceptual Model

In a natural environment, populations of different species are not isolated from each other but are involved in various inter-species interactions. For a multispecies model, given we have n species, we can describe a community of interacting species through a system of equations [9]:

$$\frac{dU_i}{dt} = F_i(U_1, \ldots, U_n),\ i = 1, \ldots n$$

where $U_i(t)$ is the population size of the ith species and n is the total number of species taken into account by the model.

Since we are modeling the interaction among species through reaction and diffusion, we can turn the above system of ODEs into a system of reaction-diffusion equations:

$$\frac{\partial u_i(r,t)}{\partial t} = D_i \triangle u_i + F_i(u_1, \ldots, u_n),\ i = 1, \ldots, n$$

where D_i is the diffusivity of the ith species.

For this project, we are investigating the spatio-temporal dynamices of a community consisting of three competitive species. They are described by the following equations:

$$\begin{aligned}
\frac{\partial u_1}{\partial t} &= D_1 \frac{\partial^2 u_1}{\partial x^2} + \epsilon_1(1 - r_{11}u_1 - r_{12}u_2 - r_{13}u_3)u_1 \\
\frac{\partial u_2}{\partial t} &= D_2 \frac{\partial^2 u_2}{\partial x^2} + \epsilon_2(1 - r_{21}u_2 - r_{22}u_2 - r_{23}u_3)u_2 \\
\frac{\partial u_3}{\partial t} &= D_3 \frac{\partial^2 u_3}{\partial x^2} + \epsilon_3(1 - r_{31}u_3 - r_{32}u_2 - r_{33}u_3)u_3
\end{aligned}$$

where u_1, u_2, u_3 show the concentration of species 1, 2, 3 correspondingly at time t for position x in space, coefficients $\epsilon_1, \epsilon_2, \epsilon_3$ give the intrinsic growth rates of the species, coefficients r_{ij} describe intra-species (when $i = j$) and inter-species (when $i \neq j$) competition respectively. Coefficients D_1, D_2, D_3 describe the intensity of spatial mixing due to self-motion of species. All parameters are nonnegative given their biological implications.

Below are the packages needed to import for this section to run:

[32]:
```
import sys
import os
import random
import math
from argparse import Namespace
from copy import deepcopy
```

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

sys.path.append('/Users/q7/Desktop/Codes/cse6730Project/Utils')

import dataForInput
import visualization
```

5.2 Parameters for the Multi-species Model

Variable	Definition
world_width	The width of the simulating world
world_length	The length of the simulating world
gap_space	The width of gap among cells in the simulating world
num_species	The total number of species in the simulation process
num_predator	The number of predator species in the simulation process
num_preys	The number of preys species in the simulation process
prey_1_population_prob	The initial probability for generating the clumped distribution of prey 1 population
prey_2_population_prob	The initial probability for generating the clumped distribution of prey 2 population
population_density_init_low	The lowerbound of the initial prey 1 and prey 2 population density
population_density_init_high	The upperbound of the initial prey 1 and prey 2 population density
highway_class	The specific label that distinguishes highway cells
prob_as_end_of_road	The probability that specifies whether the road ends
prob_for_highway_directions	An array of probabilities that specify the probability of a road going left, down, and right
D_i	Rate of diffusion for the species, $i \in \{1, 2, 3\}$
ϵ_i	Strongness of interaction effects, $i \in \{1, 2, 3\}$
r_{ij}	Coefficients of interactions between species $i, j \in \{1, 2, 3\}$
dt	Time interval for the population update throughout the simulation

[3]:
```
args = Namespace(
    world_length=30,
    world_width=30,
    gap_space=3,
    # the following two parameters are used to generate the clustered distribution␣
↪(needed to figure out what they mean)
    num_classes=1,
    areas_of_classes=[1],

    num_species=3,
```

```
    num_predator=1,
    num_preys=2,
    prey_1_population_prob=0.04,
    prey_2_population_prob=0.05,
    population_density_init_low=3,
    population_density_init_high=8,
    random_seed_for_population=7,

    random_seed_for_highway=20,
    highway_class=10,
    prob_as_end_of_road=0.5,
    prob_for_highway_direction=[0.2, 0.2, 0.6],

    plot_record_steps=30,
    simulation_steps=30 * 9,

    D1=3e-2,
    D2=5e-2,
    D3=7e-2,

    epsilon_1=0.15,
    epsilon_2=0.20,
    epsilon_3=0.25,

    r_11=1e-4, # the innerspecies competition is mild
    r_12=0,
    r_13=0,
    r_21=2e-4,
    r_22=2e-4, # the innerspecies competition is moderate
    r_23=2e-4,
    r_31=4e-4,
    r_32=2e-4,
    r_33=5e-4, # the innerspecies competition is strong

    dt=0.1,
)
```

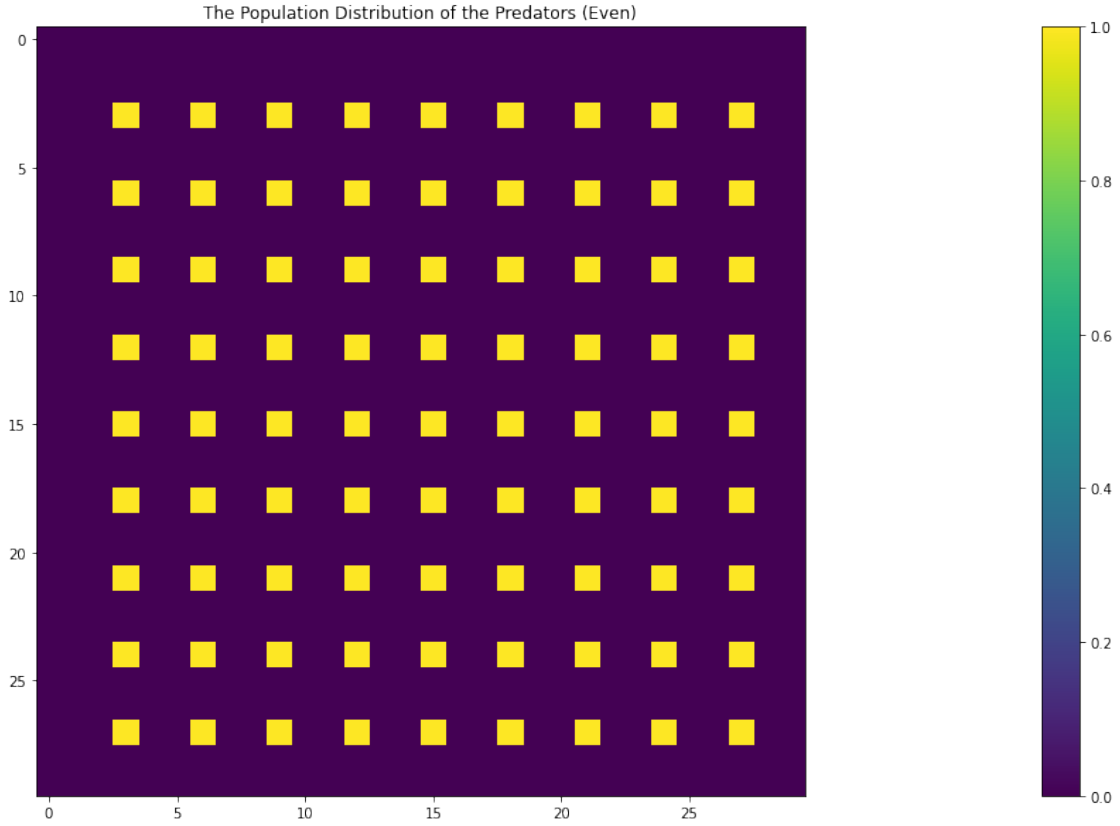
5.3 Data Generation

The three-species interaction is an extension to the predator-prey interaction discussed previously. In this case, we have two preys competiting for the same resources (e.g., red squirrel and grey squirrel) and a predator that hunts those two preys (e.g., red fox). Using the convention of data generation in predator-prey, we assume the populations of preys are clumply distributed, and the population of predators is evenly distributed

5.3.1 Even Distribution for Predators

Below is the code used to generate the even population distribution of the predator:

[33]:
```
world = dataForInput.initialWorld(args.world_length)
predator_distribution = dataForInput.genrateEvenDistribution(world, args.num_predator,␣
 ↪args.gap_space,
                                                              args.world_length)
visualization.displayPattern(predator_distribution, "The Population Distribution of␣
 ↪the Predators (Even)")
```



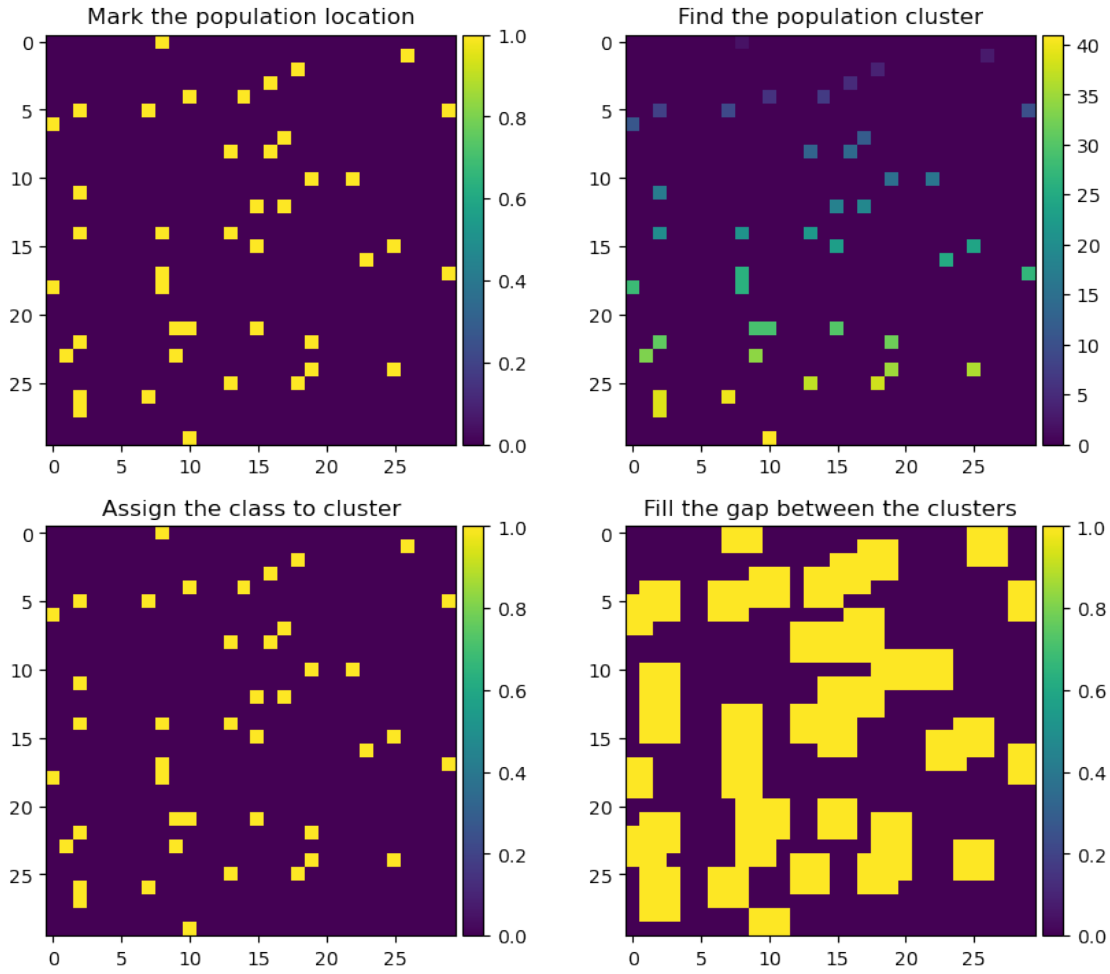
5.3.2 Clumped Distribution for Two Preys

Below is the code used to generate the clumped population distributions of prey 1 and prey 2:

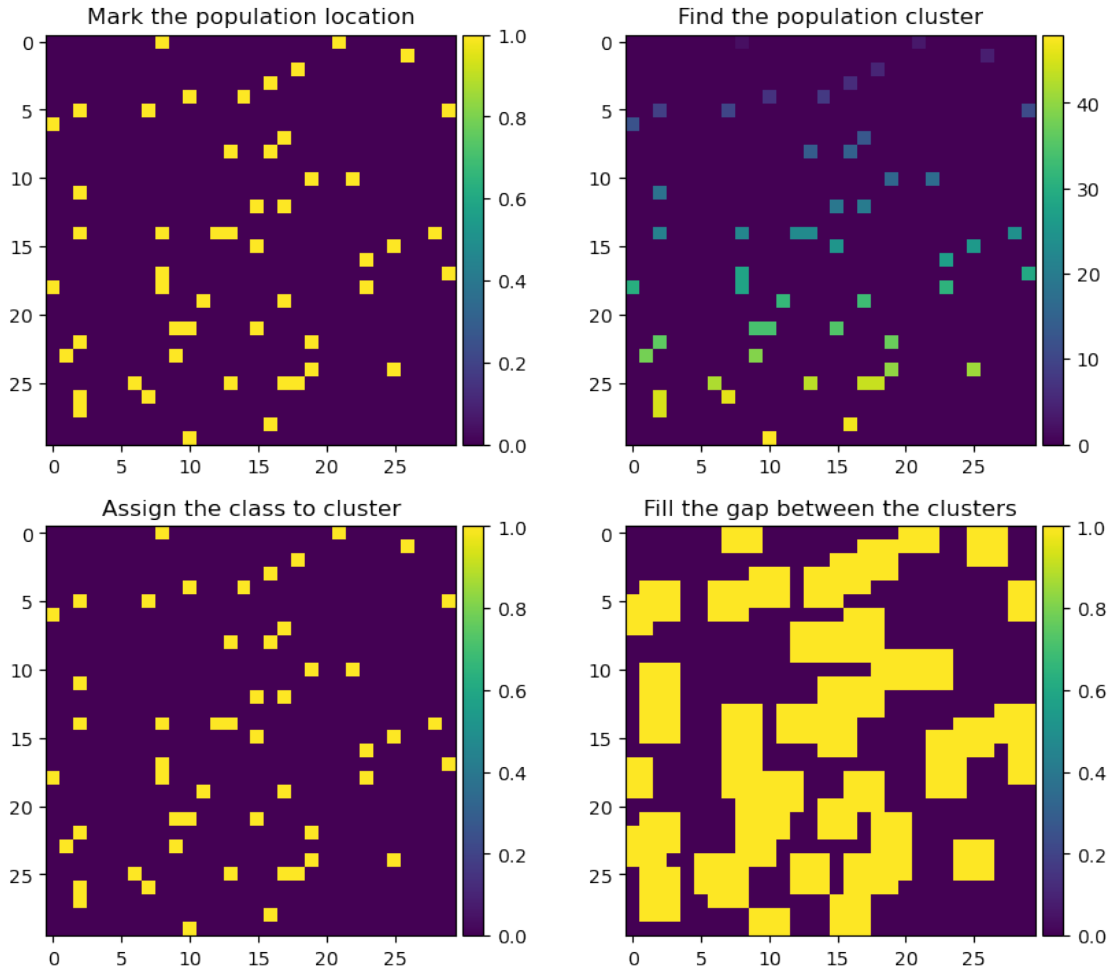
[5]:
```
def generate_and_visualize_clumped_distribution(is_prey_1):
    prey_population_prob = 0
    if is_prey_1 == True:
        prey_population_prob = args.prey_1_population_prob
    else:
        prey_population_prob = args.prey_2_population_prob

    prey_population_location = dataForInput.locatePopulationOnProb(world, args.
 ↪random_seed_for_population,
```

```
                                                                    args.world_width,␣
↪prey_population_prob)
    prey_population_cluster = dataForInput.
↪findClusterByNeighbor(prey_population_location, args.world_width)
    cluster_world = dataForInput.assignClassToCluster(args.areas_of_classes,␣
↪prey_population_cluster, args.world_width)
    cluster_world_with_filled_gaps = dataForInput.fillGapOnUnmarked(cluster_world,␣
↪args.world_width)

    fig, ax1, ax2, ax3, ax4 = visualization.initial2DArrayForGeneratedPattern()
    visualization.update2DArrayForGeneratedPattern(fig, ax1, prey_population_location)
    visualization.update2DArrayForGeneratedPattern(fig, ax2, prey_population_cluster)
    visualization.update2DArrayForGeneratedPattern(fig, ax3, cluster_world)
    visualization.update2DArrayForGeneratedPattern(fig, ax4,␣
↪cluster_world_with_filled_gaps)
    visualization.updateCharts(fig)

    # print('here')
    return cluster_world_with_filled_gaps
```

[6]:
```
# generate and visualize the clumped population of prey 1
prey_1_distribution = generate_and_visualize_clumped_distribution(is_prey_1=True)
```



[7]:
```
# generate and visualize the clumped population of prey 2
prey_2_distribution = generate_and_visualize_clumped_distribution(is_prey_1=False)
```

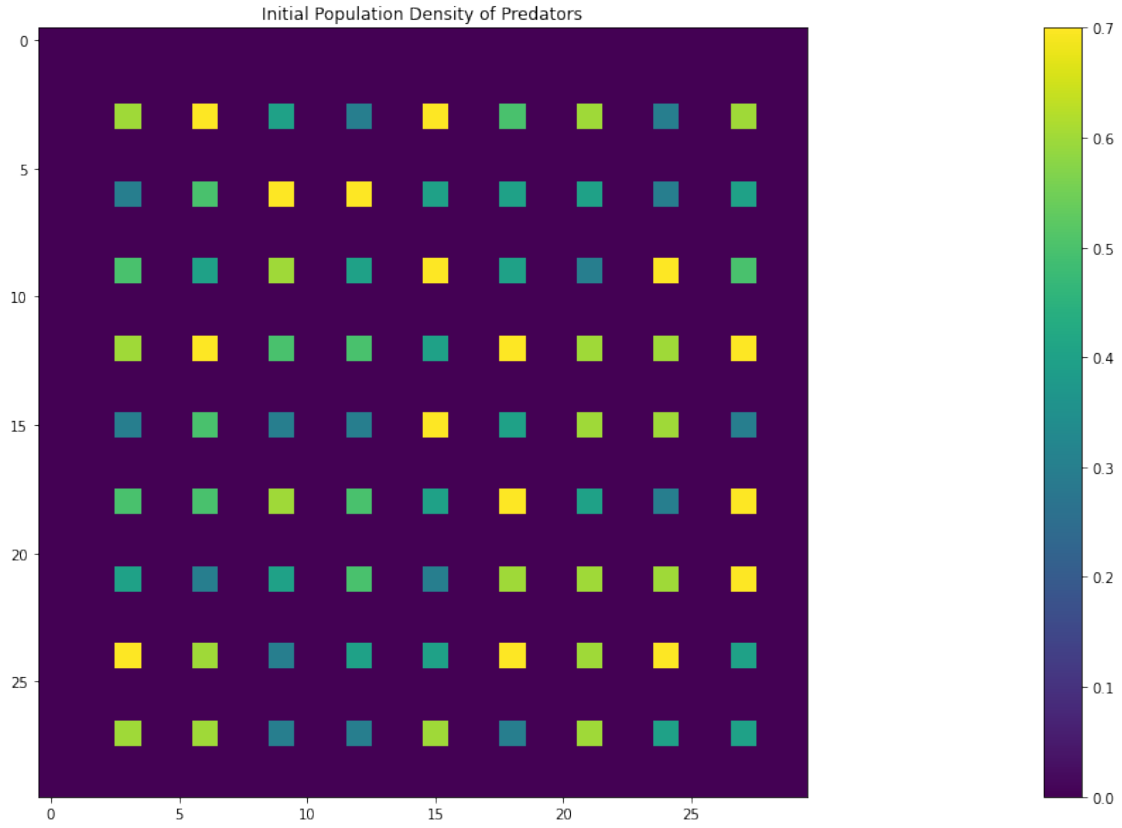


5.3.3 Generate the Actual Population Density for Predators and Preys

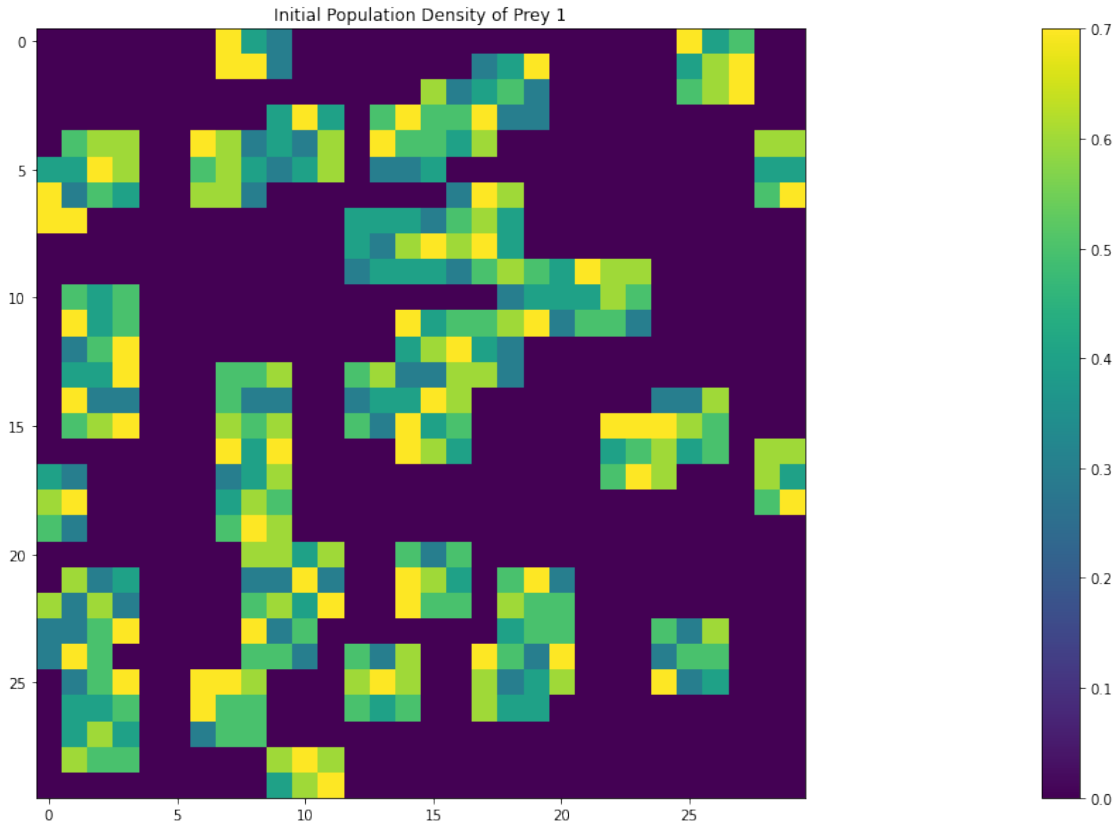
With the distribution of different species, we generate the population densities of these species. The initial population densities are set to range from 0.3 to 0.8, avoiding the situation in which the assigned initial population density is 0.

[8]:
```
def fill_out_population_density(distribution):
    world_with_population_density = dataForInput.initialWorld(args.world_width)
    for i in range(args.world_width):
        for j in range(args.world_width):
            if distribution[i][j] == 1:
                world_with_population_density[i][j] = np.random.randint(args.
↪population_density_init_low,
                                                                        args.
↪population_density_init_high) / 10.0
    return world_with_population_density
```

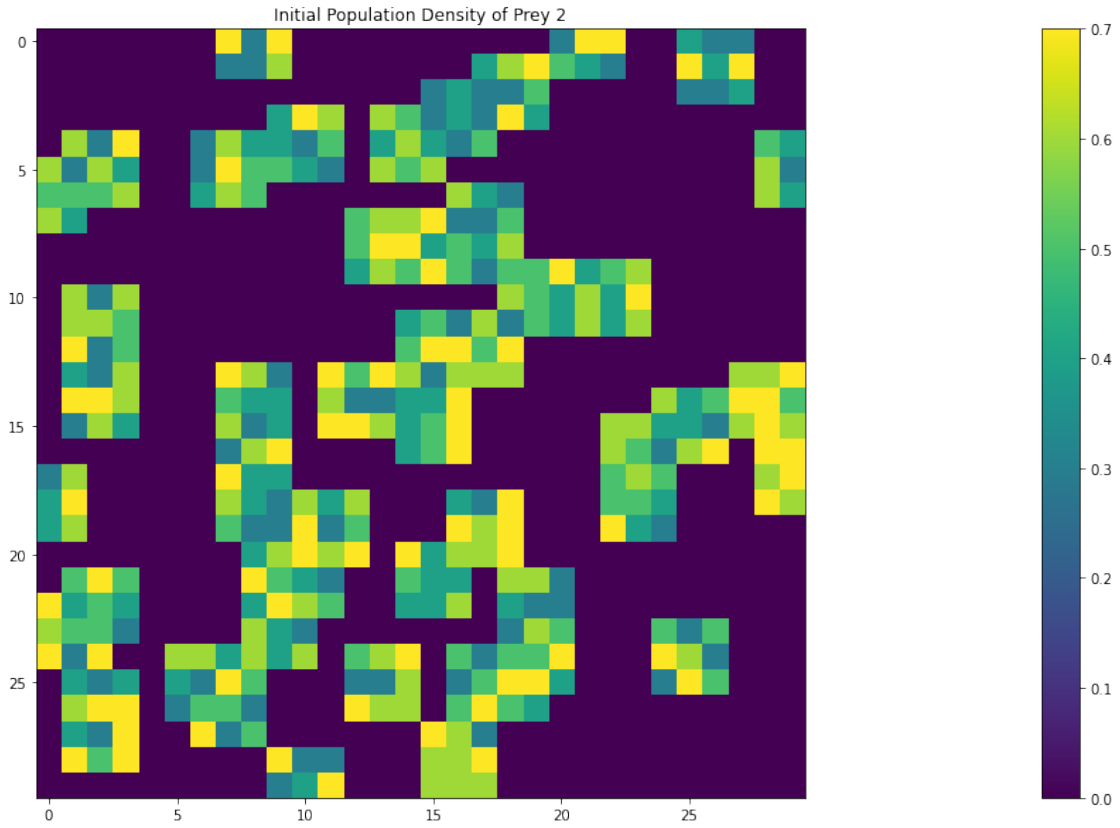
[9]:
```
predator_population_density = fill_out_population_density(predator_distribution)
visualization.displayPattern(predator_population_density, "Initial Population Density␣
↪of Predators")
```



[10]:
```
prey_1_population_density = fill_out_population_density(prey_1_distribution)
visualization.displayPattern(prey_1_population_density ,"Initial Population Density of␣
↪Prey 1")
```



[11]:
```
prey_2_population_density = fill_out_population_density(prey_2_distribution)
visualization.displayPattern(prey_2_population_density, "Initial Population Density of␣
↪Prey 2")
```



5.4 Simulation, Visualization and Verification

5.4.1 Utils

Below are the utility functions that faciliate the simlation process:

[13]:
```
# TODO: rename the variables and extend args to include values for variables like␣
↪population_per_cell
def calculate_population(world, population_per_cell = 100):
    res = world.copy()
    return [population_per_cell * world[i,j] \
            for i in range(args.world_width) \
            for j in range(args.world_width)]


def draw_population(world_list):
    total_population = []

    for i in range(len(world_list)):
        total_population.append(sum(calculate_population(world_list[i], 100)))

    temp = pd.DataFrame({'Generations': list(range(len(world_list))), \
```

```
                        'Total Population': total_population}).\
                        plot(x ='Generations', y='Total Population', figsize=(10,5))


def plot_density_over_generations(density_track, title):
    n = len(density_track)
    row_len = round(math.sqrt(n))
    col_len = int(n * 1.0 / row_len)
    fig, axes = plt.subplots(row_len, col_len, figsize = (20, 15))

    for i in range(n):
        ax = axes.flat[i]
        ax.set_title((title + ' population at generation {}').format(i + 1), size=10)
        show_obj = ax.imshow(density_track[i], interpolation = 'none')
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size = "5%", pad = 0.05)
        fig.colorbar(show_obj, cax = cax, orientation = 'vertical')

    draw_population(density_track)


def intialize_population_density():
    predator_population_density = fill_out_population_density(predator_distribution)
    prey_1_population_density = fill_out_population_density(prey_1_distribution)
    prey_2_population_density = fill_out_population_density(prey_2_distribution)
    return predator_population_density, prey_1_population_density,␣
 ↪prey_2_population_density
```

5.4.2 Update Function

Since the simulation belongs to the reaction-diffusion family, the implementation of the simulation is also consisted of two subparts – diffusion and reaction. First, we calculate the discrete laplacian of the current population density to simulate the diffusion process. Next, we derive the corresponding reaction terms for each species according to the population densities of different species.

[14]:
```
def get_discrete_laplacian(M):
    """Get the discrete Laplacian of matrix M"""
    L = deepcopy(M)
    L = -4*M
    L += np.roll(M, (0,-1), axis=1) # right neighbor
    L += np.roll(M, (0,+1), axis=1) # left neighbor
    L += np.roll(M, (-1,0), axis=0) # top neighbor
    L += np.roll(M, (+1,0), axis=0) # bottom neighbor

    return L
```

[15]:
```
def update_three_species_population_density(u1, u2, u3):
    """
    Updates a density configuration according to the conceptual model
```

```
    u1 denotes the population density of prey 1, u2 denotes the populaion density of␣
↪prey 2,
    and u3 denotes the popualtion density of the predator
    """

    # obtain the discrete Laplacians first
    L1 = get_discrete_laplacian(u1)
    L2 = get_discrete_laplacian(u2)
    L3 = get_discrete_laplacian(u3)

    # print('args.D1', args.D1)

    # apply the update formula
    diff_u1 = (args.D1*L1 + args.epsilon_1*(1-args.r_11*u1-args.r_12*u2-args.
↪r_13*u3)*u1) * args.dt
    diff_u2 = (args.D2*L2 + args.epsilon_2*(1-args.r_21*u1-args.r_22*u2-args.
↪r_23*u3)*u2) * args.dt
    diff_u3 = (args.D3*L3 + args.epsilon_3*(1-args.r_31*u1-args.r_32*u2-args.
↪r_33*u3)*u3) * args.dt

    u1 += diff_u1
    u2 += diff_u2
    u3 += diff_u3

    return u1, u2, u3
```

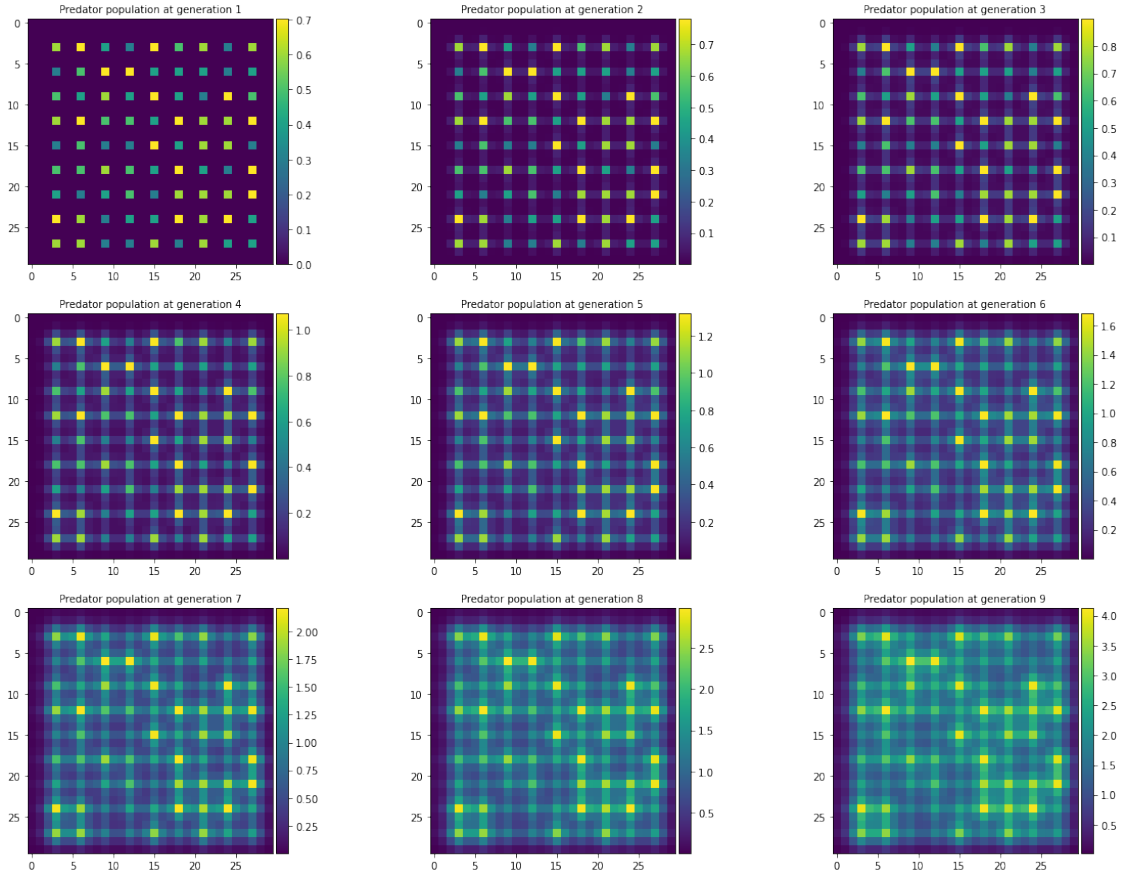
[17]:
```
def simulate(predator_population_density, prey_1_population_density,␣
↪prey_2_population_density):
    predator_density_generation_track = []
    prey_1_density_generation_track = []
    prey_2_density_generation_track = []

    for i in range(args.simulation_steps):
        predator_population_density, prey_1_population_density,\
        prey_2_population_density =␣
↪update_three_species_population_density(predator_population_density,
                                                prey_1_population_density,
                                                prey_2_population_density)
        if i % args.plot_record_steps == 0:
            predator_density_generation_track.
↪append(deepcopy(predator_population_density))
            prey_1_density_generation_track.append(deepcopy(prey_1_population_density))
            prey_2_density_generation_track.append(deepcopy(prey_2_population_density))

    return predator_density_generation_track, prey_1_density_generation_track,␣
↪prey_2_density_generation_track
```

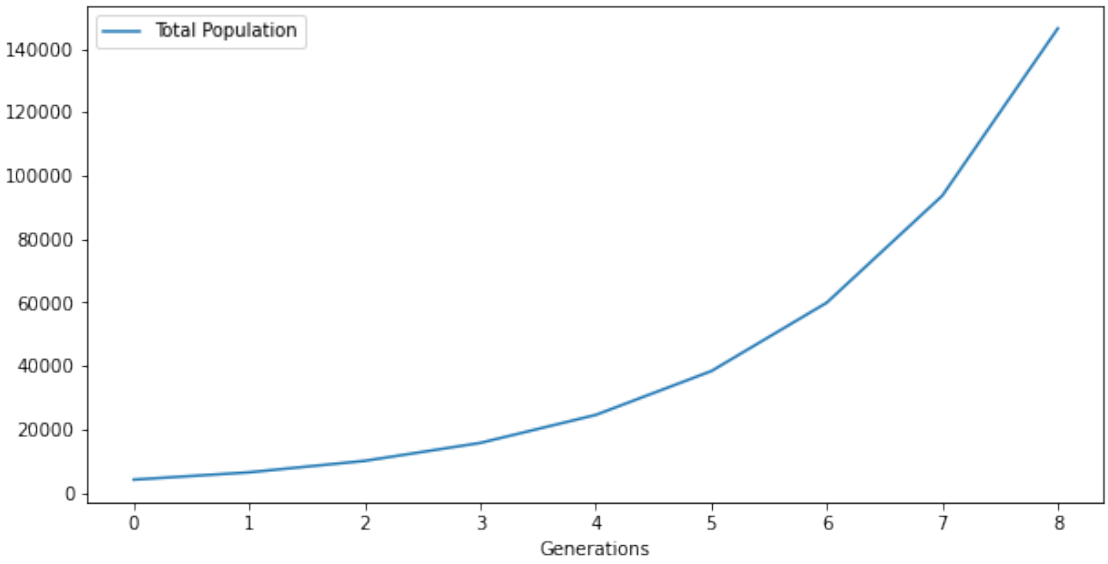
[18]:
```

```

```
predator_density_generation_track, prey_1_density_generation_track,␣
↪prey_2_density_generation_track =\
 simulate(predator_population_density, prey_1_population_density,␣
↪prey_2_population_density)
```

Below are the population densities of predator over 9 generations. We can see that as the simulation goes on, because of the effect of diffusion, the population density becomes more spread-out. The population density of spots with high population density remains high.

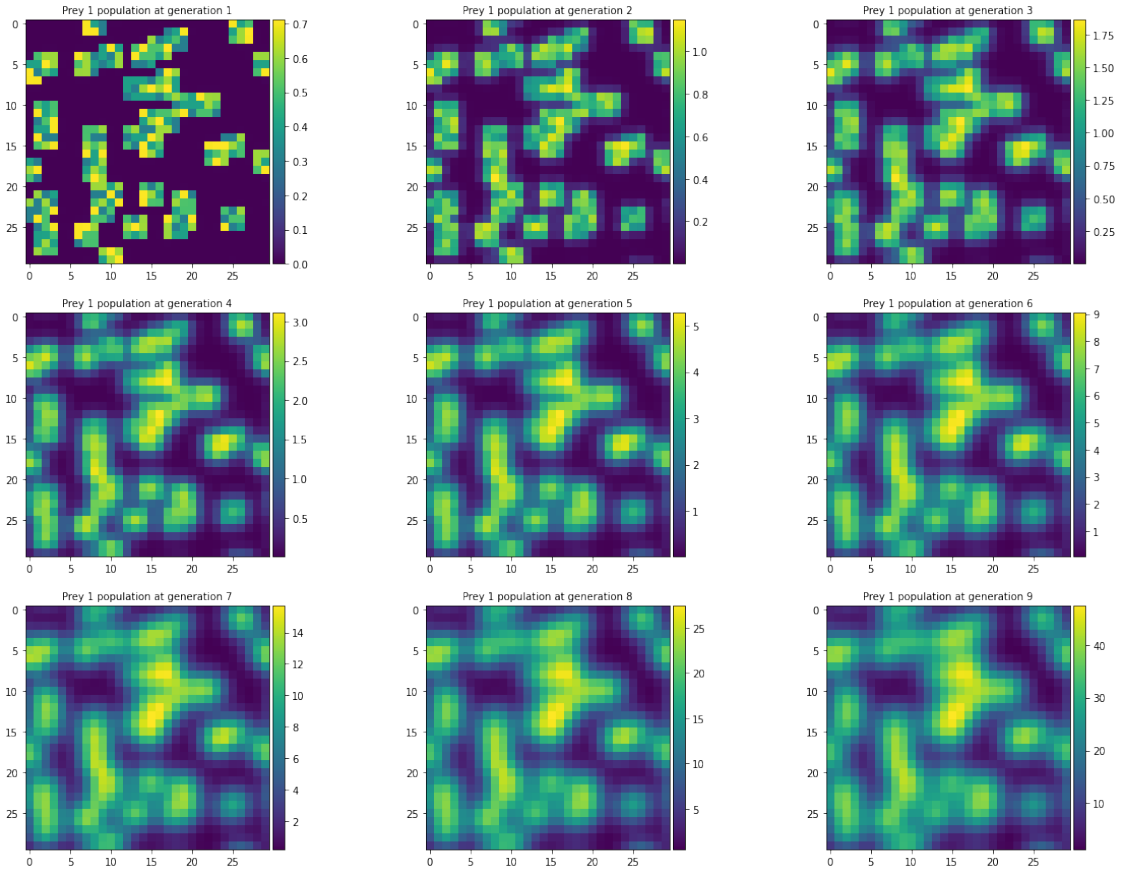
[19]:
```
plot_density_over_generations(predator_density_generation_track, 'Predator')
```

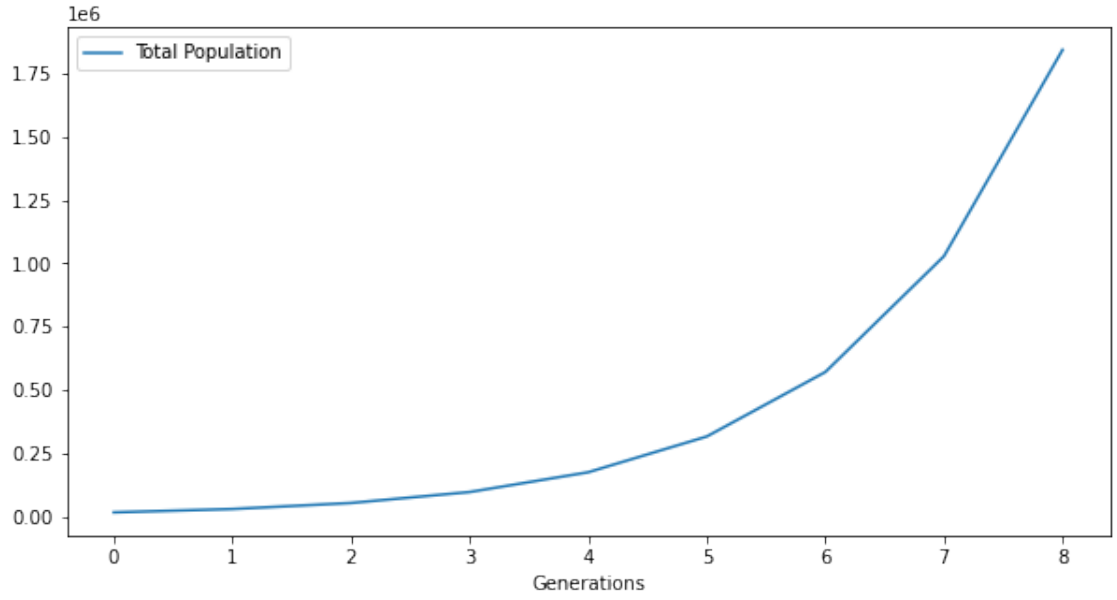




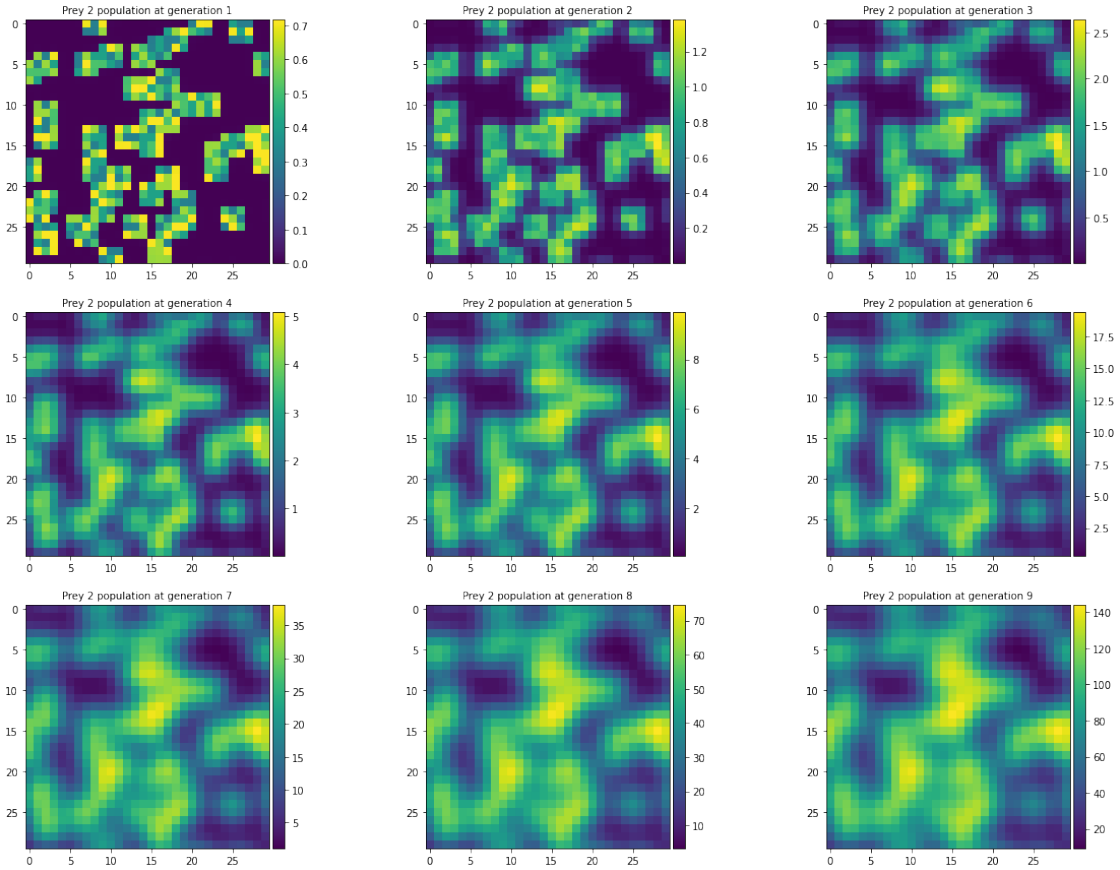
Below are the population densities of prey 1 and prey 2 over 9 generations. We can see that as the simulation goes on, because of the effect of diffusion, the population density becomes more spread-out. Also, even though the inital distributions of the two species are clumped, over time, it seems that these clumps move toward each other get merged eventually.

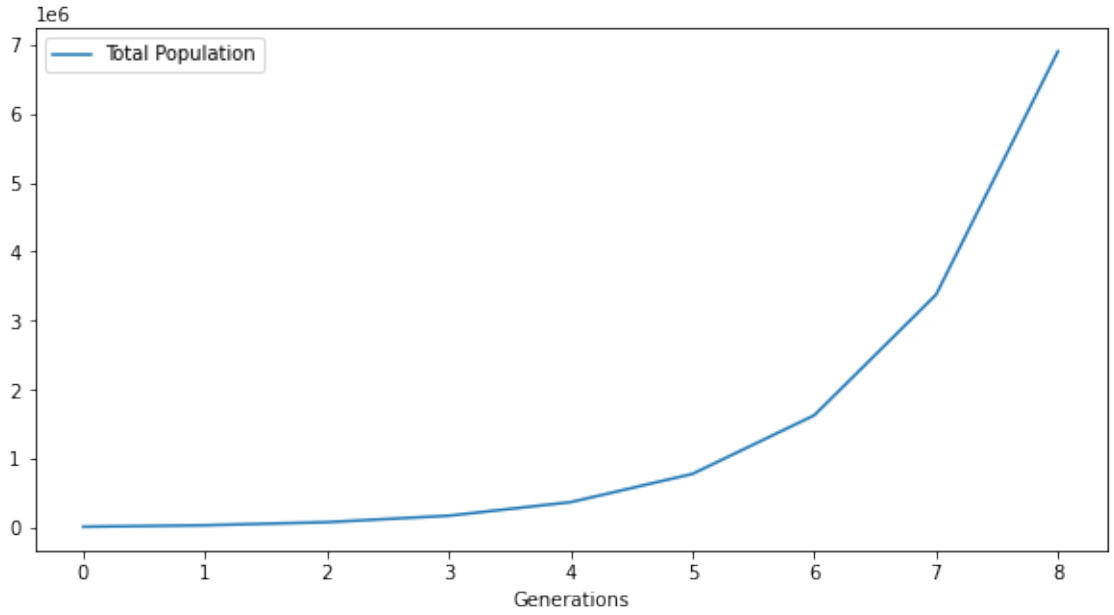
[20]:
```
plot_density_over_generations(prey_1_density_generation_track, 'Prey 1')
```


Prey 1 population at generation 1
Prey 1 population at generation 2
Prey 1 population at generation 3
Prey 1 population at generation 4
Prey 1 population at generation 5
Prey 1 population at generation 6
Prey 1 population at generation 7
Prey 1 population at generation 8
Prey 1 population at generation 9


1e6
Total Population
Generations

[21]:
```
plot_density_over_generations(prey_2_density_generation_track, 'Prey 2')
```





5.5 Sensitivity Analysis and Validation

In the sensitivity analysis, we mainly explore how the rate of diffusion and the strongness of interaction terms affect the population densities of the three species.

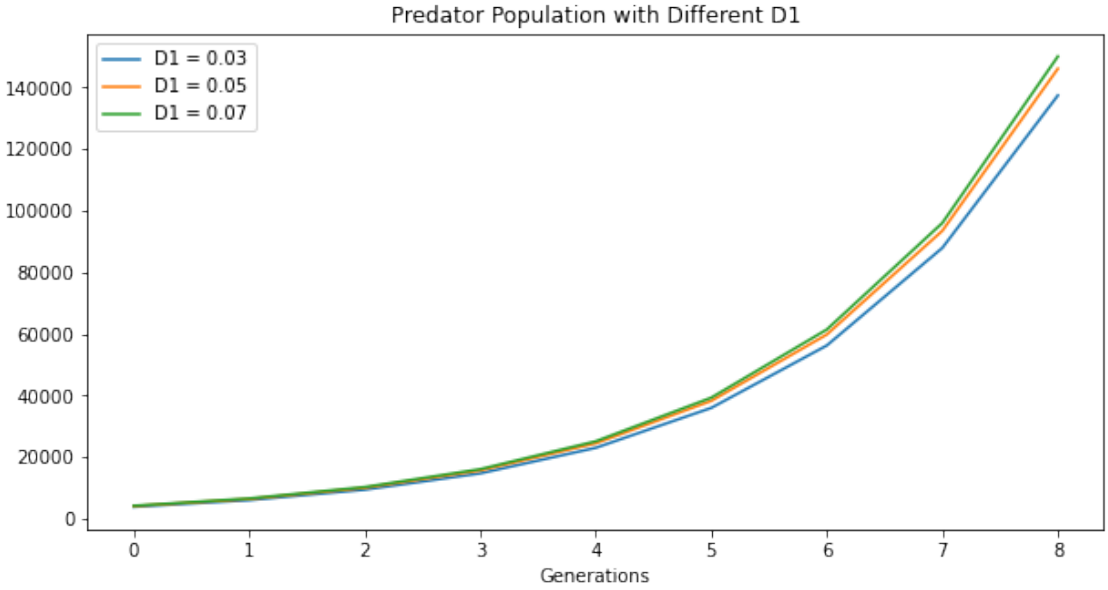
5.5.1 The Rate of Diffusion

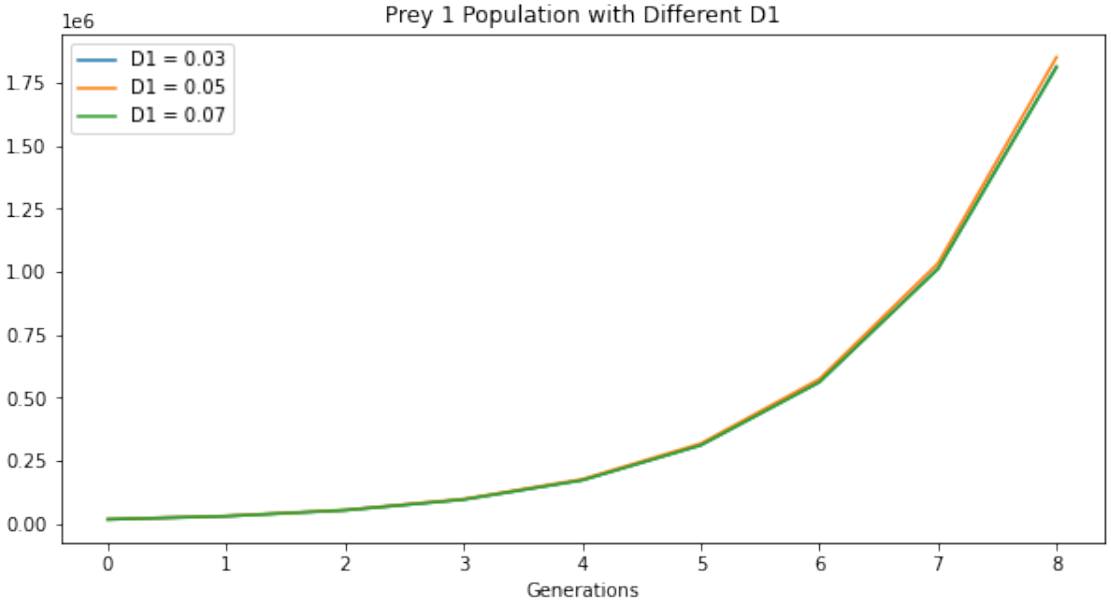
In this subsection we try to investigate how the rate of diffusion among creatrues in a same species affects the population density. We vary D_i in $\{0.03, 0.05, 0.07\}$ for $i \in \{1, 2, 3\}$.

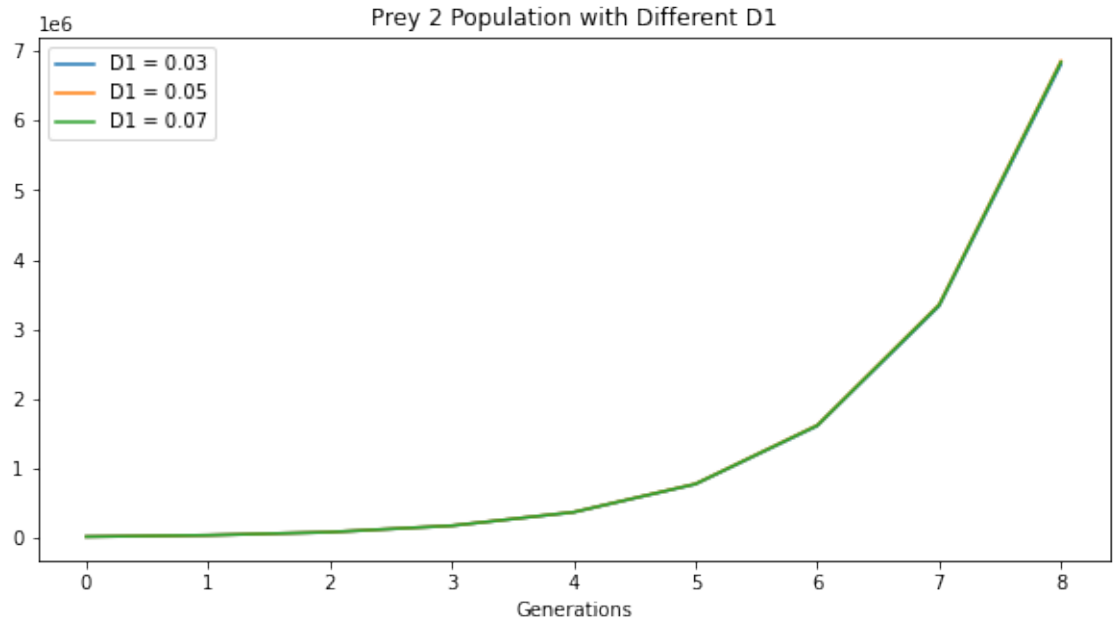
[22]:
```
def plot_population_density_with_different_parameters(tracks, population_name,␣
↪parameter_name, parameter_values):
    df = pd.DataFrame({'Generations': list(range(len(tracks[0]))),
                       '{} = {}'.format(parameter_name, parameter_values[0]):␣
↪[sum(calculate_population(tracks[0][i], 100)) for i in range(len(tracks[0]))],
                       '{} = {}'.format(parameter_name, parameter_values[1]):␣
↪[sum(calculate_population(tracks[1][i], 100)) for i in range(len(tracks[1]))],
                       '{} = {}'.format(parameter_name, parameter_values[2]):␣
↪[sum(calculate_population(tracks[2][i], 100)) for i in range(len(tracks[2]))]})
    df.plot(x = 'Generations', figsize = [10,5], title = '{} Population with Different␣
↪{}'.format(population_name, parameter_name))
```

First, we vary D_1 in values of $\{0.03, 0.05, 0.07\}$. We can see that when $D_1 = 0.03$, the population densities of prey 1 and prey 2 are the largetst. This conforms with the commonsense because if the distribution of predators is less spread-out and the population of predators is less, the preys have higher chance of survival.

[23]:
```
candidate_D1 = [3e-2, 5e-2, 7e-2]
original_D1 = args.D1

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.D1 = candidate_D1[0]
predator_density_generation_track_1, prey_1_density_generation_track_1,\
          prey_2_density_generation_track_1 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.D1 = candidate_D1[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_2,\
          prey_1_density_generation_track_2, prey_2_density_generation_track_2 =␣
 ↪simulate(predator_population_density, prey_1_population_density,␣
 ↪prey_2_population_density)


args.D1 = candidate_D1[2]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.D1 = original_D1

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'D1', candidate_D1)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'D1', candidate_D1)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'D1', candidate_D1)
```


Predator Population with Different D1
D1 = 0.03
D1 = 0.05
D1 = 0.07
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
8
Generations

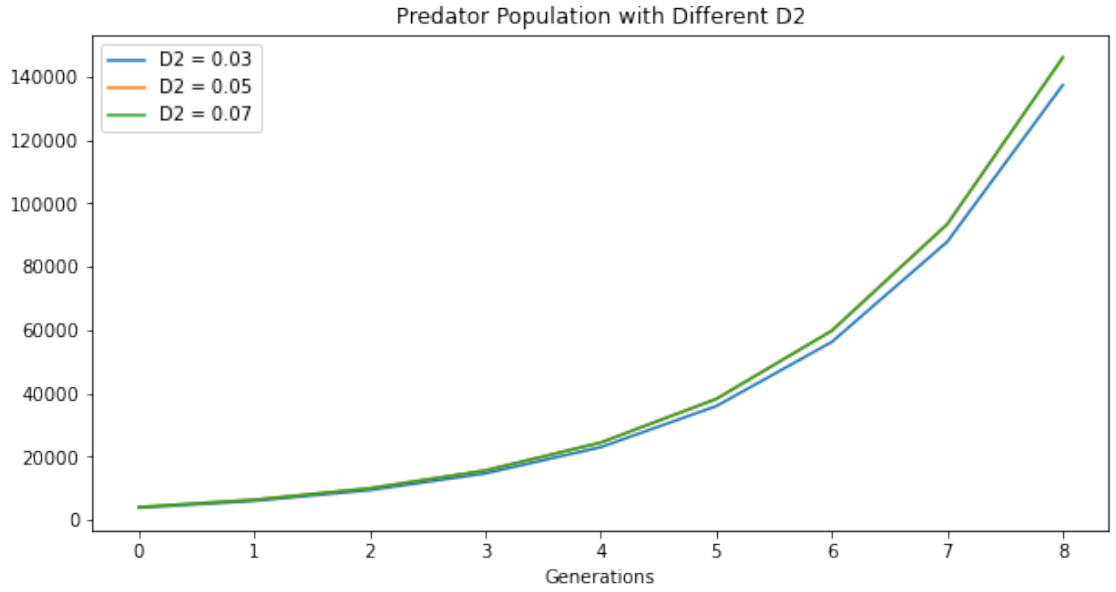

Prey 1 Population with Different D1
1e6
D1 = 0.03
D1 = 0.05
D1 = 0.07
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
6
7
8
Generations

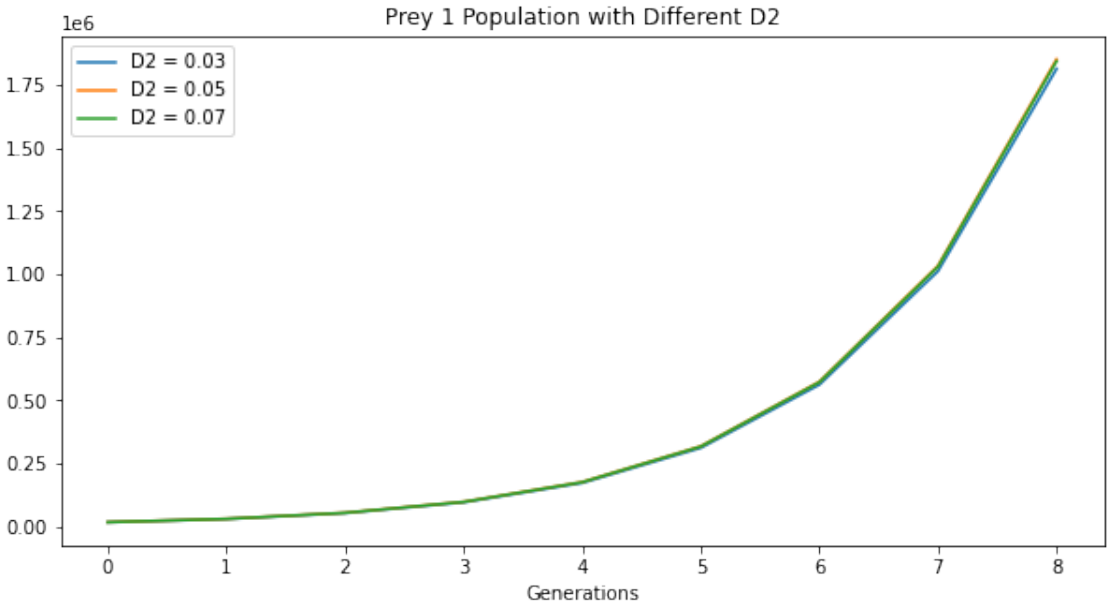


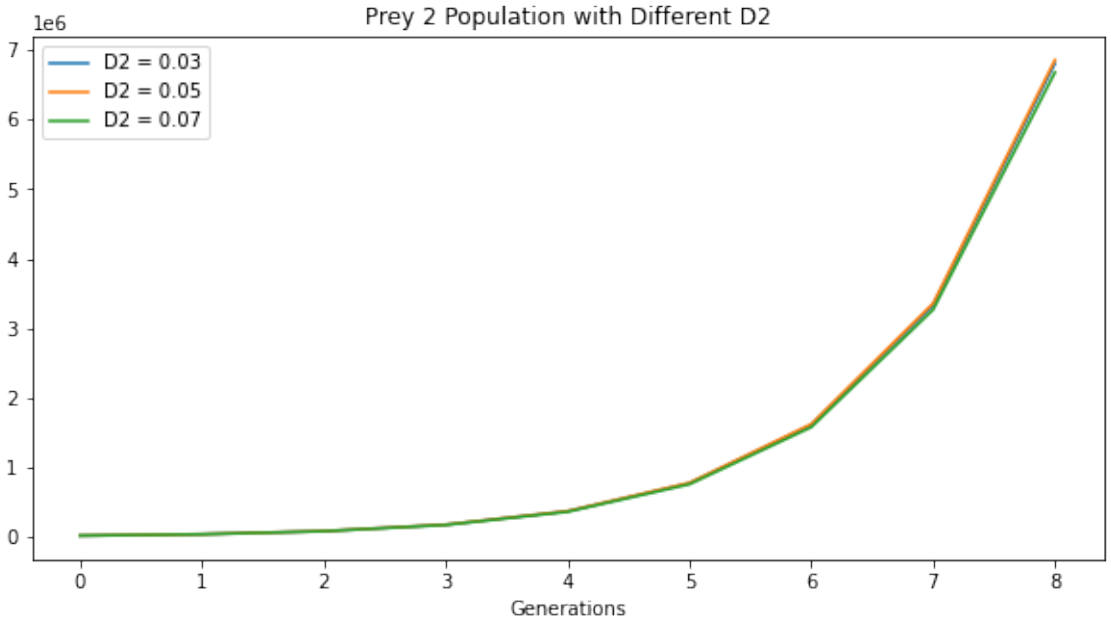
Next, we vary D_2 in values of $\{0.03, 0.05, 0.07\}$. We can see that when $D_2 = 0.03$, the population size of prey 1 is the largetst. This is explainable because if prey 1 and prey 2 compete each other for resources, if the distribution of prey 2 is less spread-out and the population density of prey 2 is lower, the population size of prey 1 should be higher.

[24]:
```
candidate_D2 = [3e-2, 5e-2, 7e-2]
original_D2 = args.D2

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.D2 = candidate_D2[0]
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.D2 = candidate_D2[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_5,\
          prey_1_density_generation_track_5, prey_2_density_generation_track_5 =␣
 ↪simulate(predator_population_density, prey_1_population_density,␣
 ↪prey_2_population_density)


args.D2 = candidate_D2[2]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
```

```
predator_density_generation_track_7, prey_1_density_generation_track_7,\
          prey_2_density_generation_track_7 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.D2 = original_D2

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'D2', candidate_D2)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'D2', candidate_D2)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'D2', candidate_D2)
```

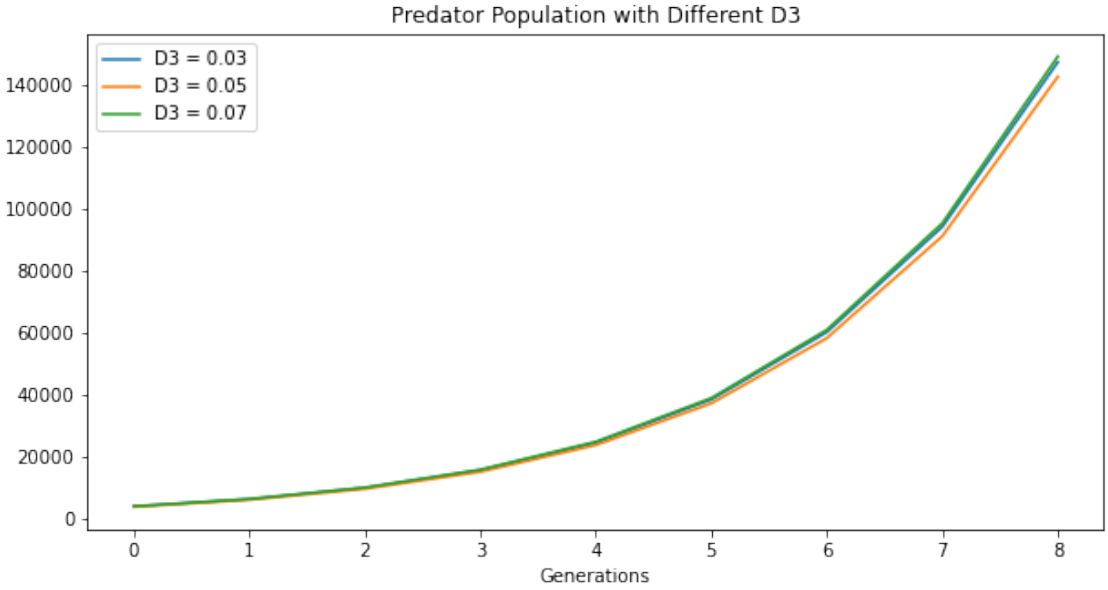


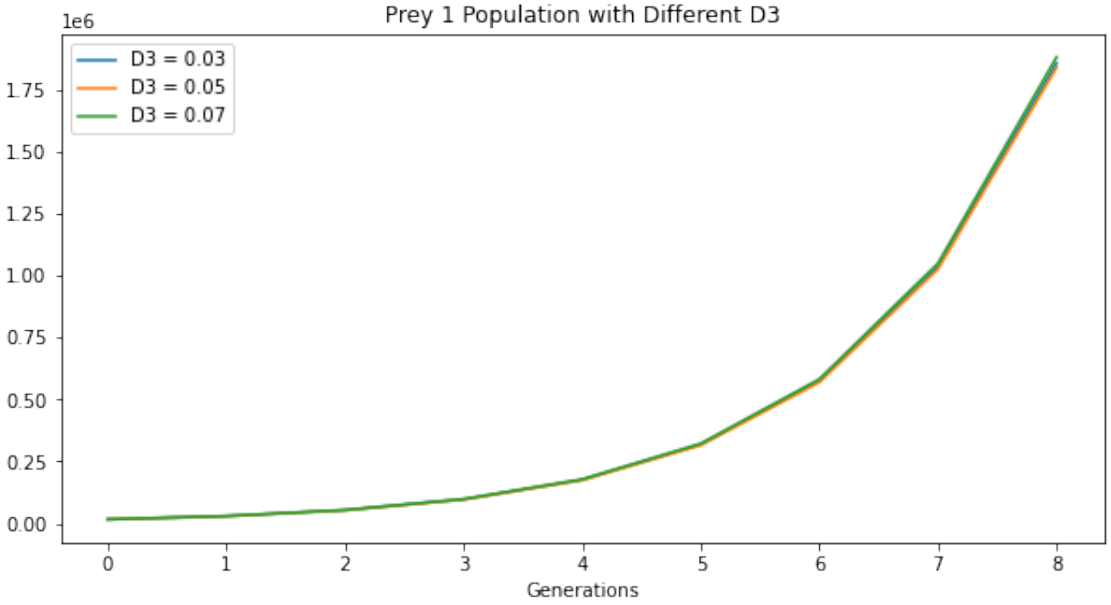


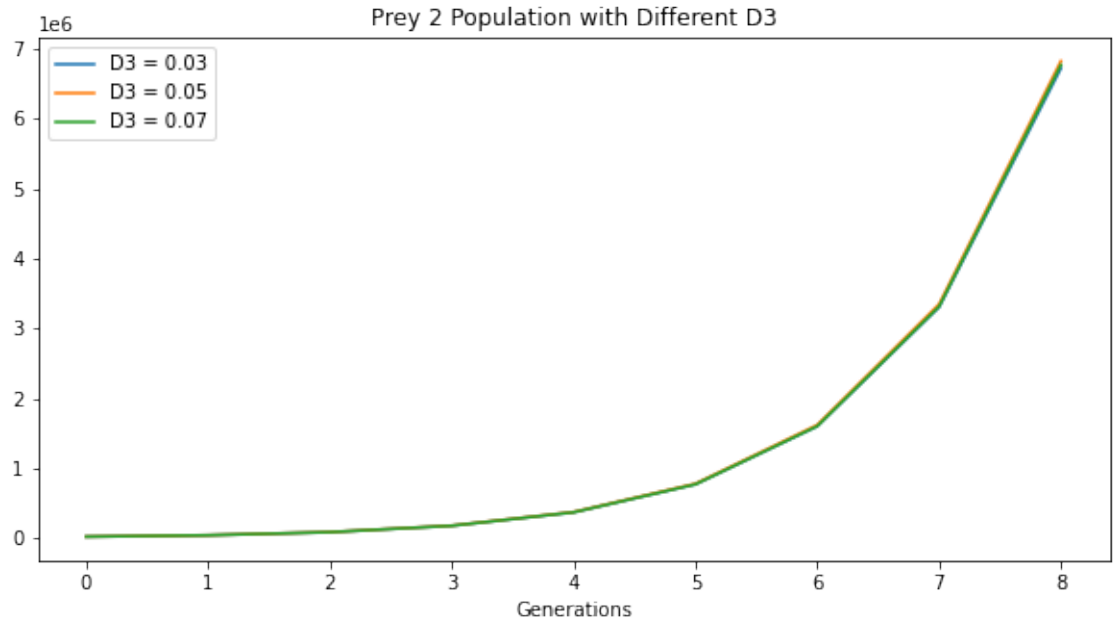


Finally, we vary D_3 in values of $\{0.03, 0.05, 0.07\}$. We can see that when $D_3 = 0.03$, the population density of prey 2 is the largetst. This makes sense because if prey 1 and prey 2 compete each other for the same resources, if the distribution of prey 2 is less spread-out and the population size of prey 2 is lower, the population size of prey 1 should be higher.

[25]:
```
candidate_D3 = [3e-2, 5e-2, 7e-2]
original_D3 = args.D3

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.D3 = candidate_D3[0]
predator_density_generation_track_1, prey_1_density_generation_track_1,\
          prey_2_density_generation_track_1 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.D3 = candidate_D3[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_2, prey_1_density_generation_track_2, \
          prey_2_density_generation_track_2 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)


args.D3 = candidate_D3[2]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.D2 = original_D3

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'D3', candidate_D3)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'D3', candidate_D3)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'D3', candidate_D3)
```


Predator Population with Different D3
D3 = 0.03
D3 = 0.05
D3 = 0.07
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
8
Generations


Prey 1 Population with Different D3
1e6
D3 = 0.03
D3 = 0.05
D3 = 0.07
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
6
7
8
Generations



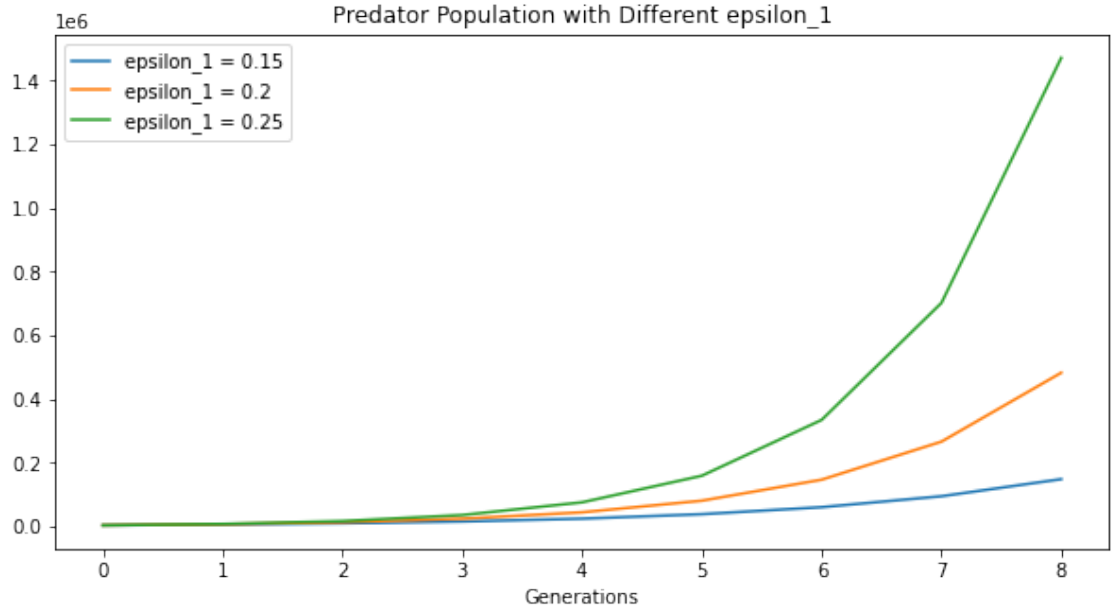
5.5.2 The Effect of Interaction Term

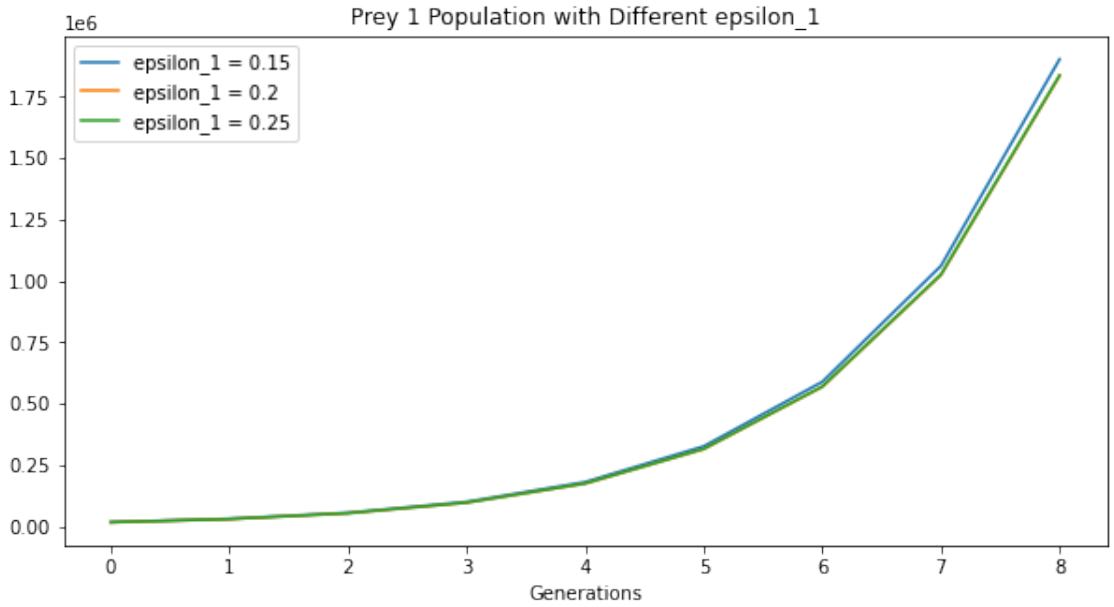
In this subsection we try to investigate how the strongness of interactions among creatrues in different species affects the population density. We vary ϵ_i in $\{0.15, 0.20, 0.25\}$ for $i \in \{1, 2, 3\}$.

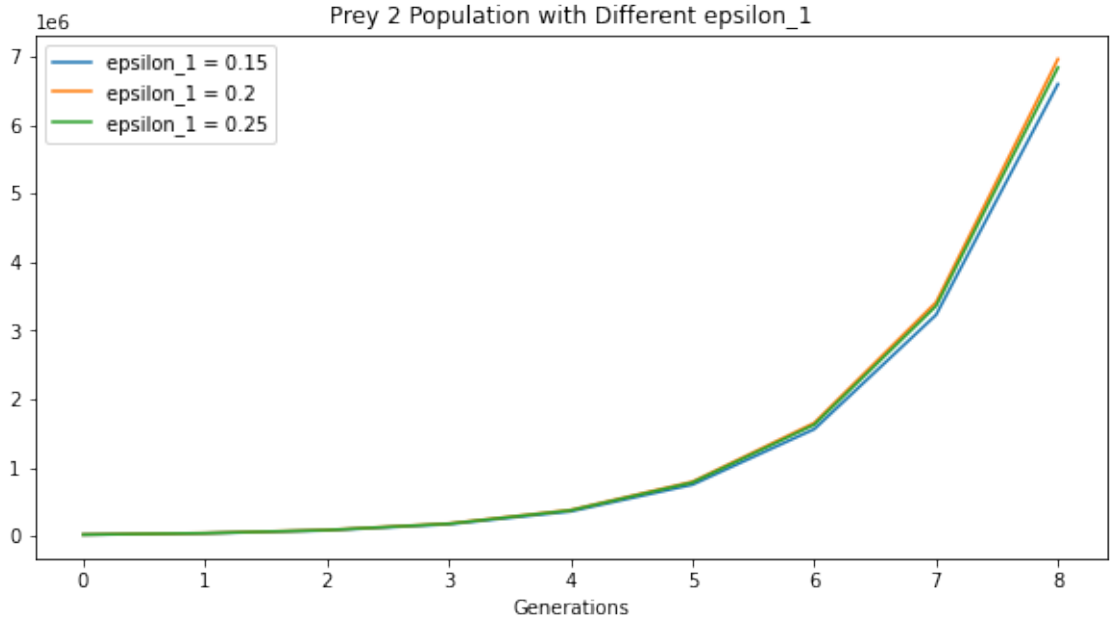
First, we vary ϵ_1 in values of $\{0.15, 0.20, 0.25\}$. From the below figures, we see that when the effect of interaction gets stronger, the total population of the predator gets larger.

[26]:
```
candidate_epsilon_1 = [0.15, 0.20, 0.25]

original_epsilon_1 = args.epsilon_1

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.epsilon_1 = candidate_epsilon_1[0]
predator_density_generation_track_1, prey_1_density_generation_track_1,\
          prey_2_density_generation_track_1 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_1 = candidate_epsilon_1[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_2, prey_1_density_generation_track_2,\
          prey_2_density_generation_track_2 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_1 = candidate_epsilon_1[2]
```

```
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.epsilon_1 = original_epsilon_1

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'epsilon_1', candidate_epsilon_1)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'epsilon_1', candidate_epsilon_1)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'epsilon_1', candidate_epsilon_1)
```

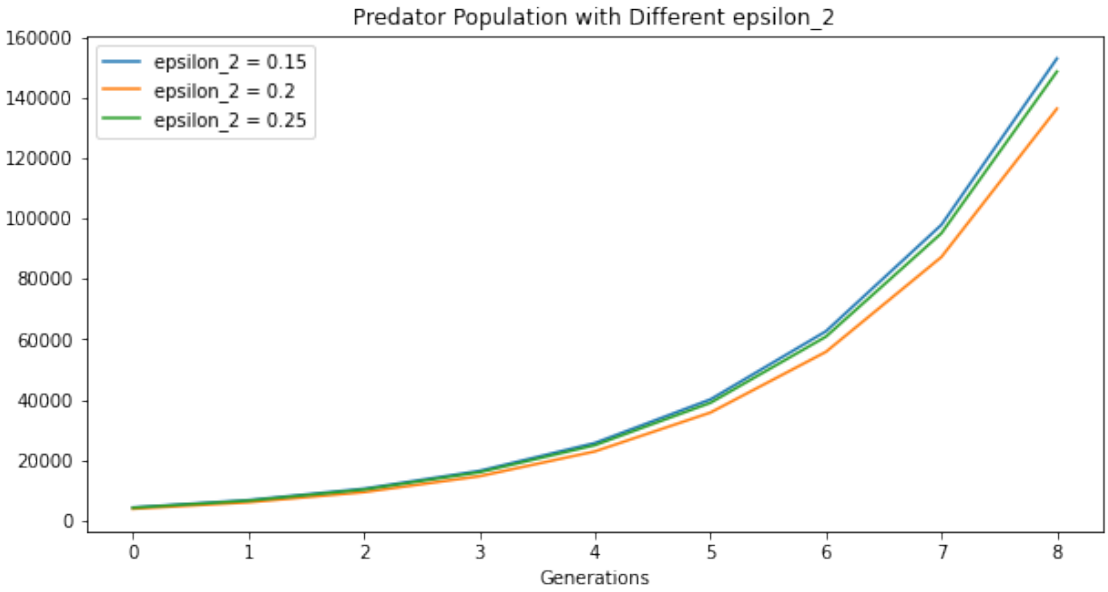


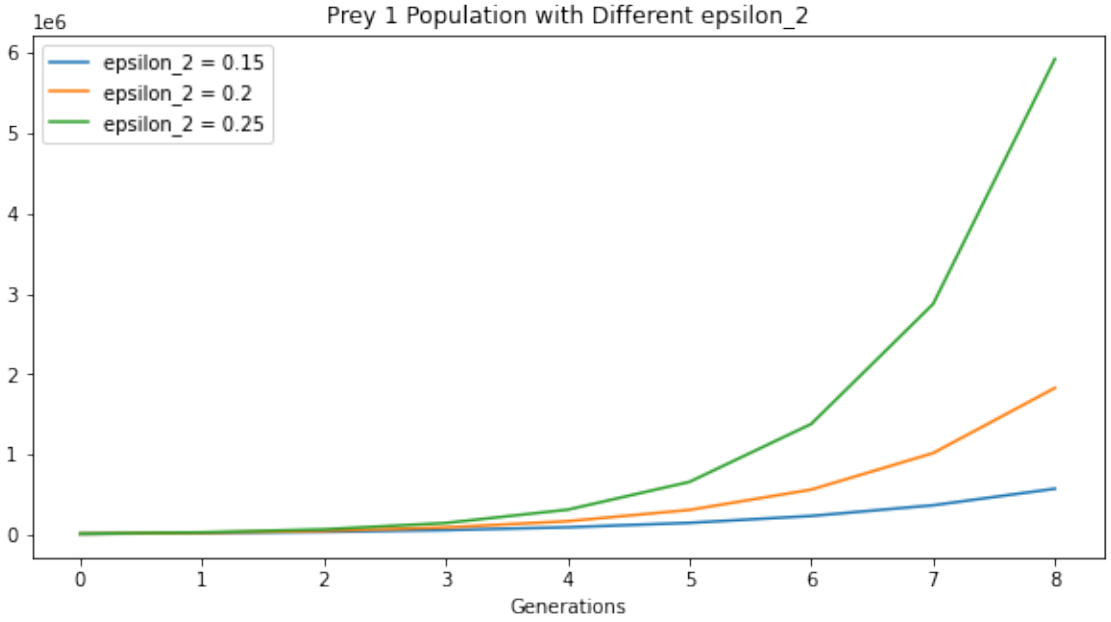


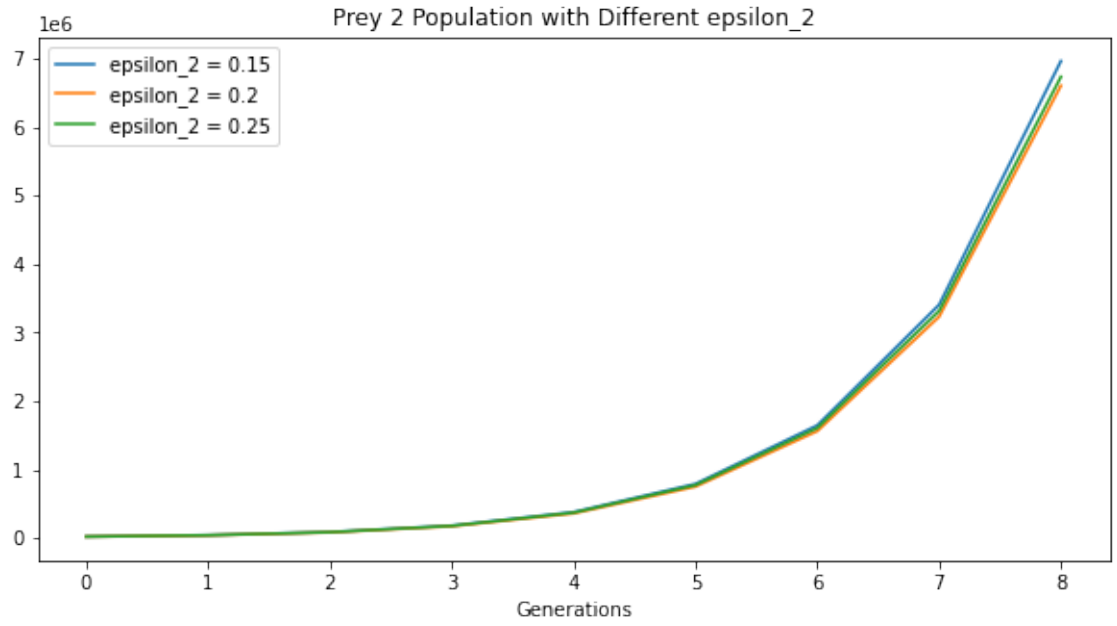


Next, we vary ϵ_2 in values of $\{0.15, 0.20, 0.25\}$. From the below graphs, we see that when the effect of interaction gets stronger, the total population of the prey 1 gets larger. Also, the population of prey 2 is the largest when the ϵ_2 is the lowest in $\{0.15, 0.20, 0.25\}$. This is in accordance with the commonsense because prey 1 and prey 2 competes for the same resources and negatively impact each other. When $\epsilon_2 = 0.15$, the population of prey 1 is also the lowest. Thus, the population of prey 2 should be the largest when $\epsilon_2 = 0.15$.

```
[ ]: candidate_epsilon_2 = [0.15, 0.20, 0.25]

original_epsilon_2 = args.epsilon_2

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.epsilon_2 = candidate_epsilon_2[0]
predator_density_generation_track_1, prey_1_density_generation_track_1,\
          prey_2_density_generation_track_1 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_2 = candidate_epsilon_2[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_2, prey_1_density_generation_track_2,\
          prey_2_density_generation_track_2 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_2 = candidate_epsilon_2[2]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.epsilon_2 = original_epsilon_2

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'epsilon_2', candidate_epsilon_2)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'epsilon_2', candidate_epsilon_2)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'epsilon_2', candidate_epsilon_2)
```


Predator Population with Different epsilon_2
epsilon_2 = 0.15
epsilon_2 = 0.2
epsilon_2 = 0.25
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
8
Generations

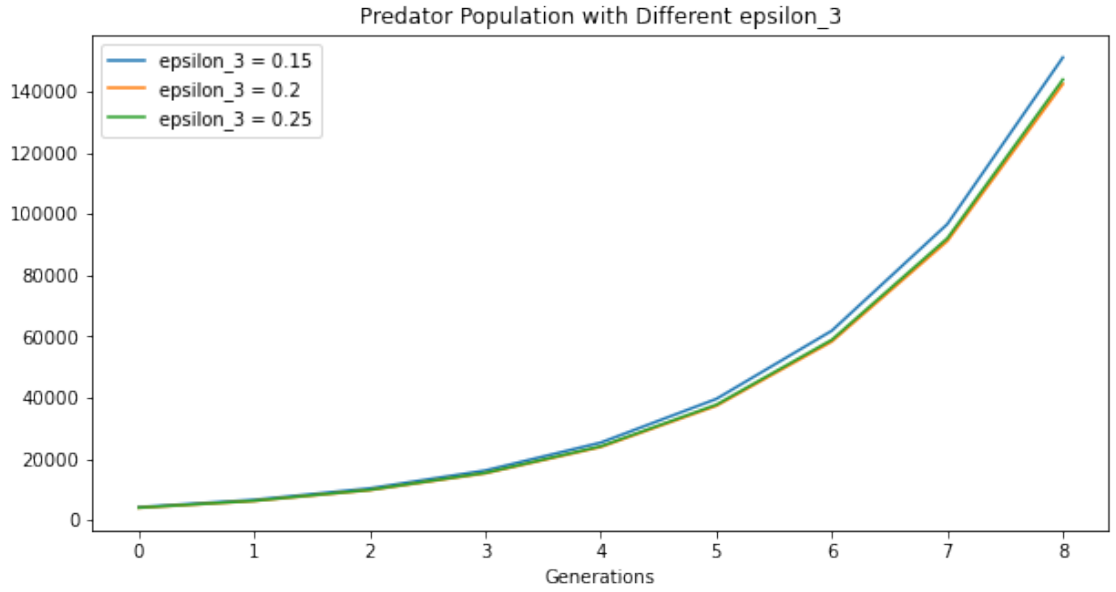

Prey 1 Population with Different epsilon_2
1e6
epsilon_2 = 0.15
epsilon_2 = 0.2
epsilon_2 = 0.25
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
Generations

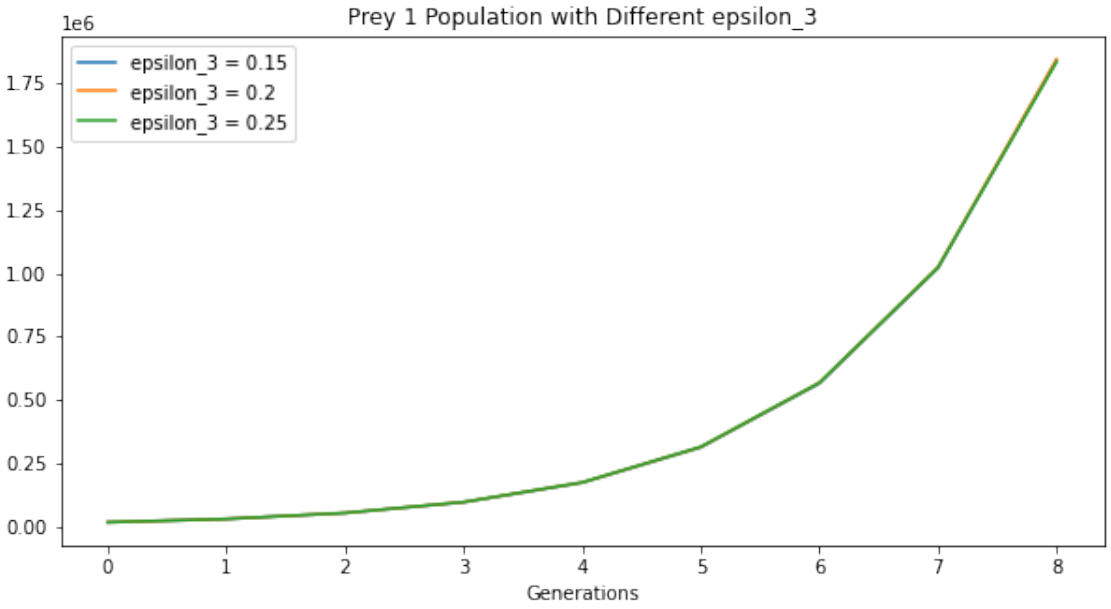


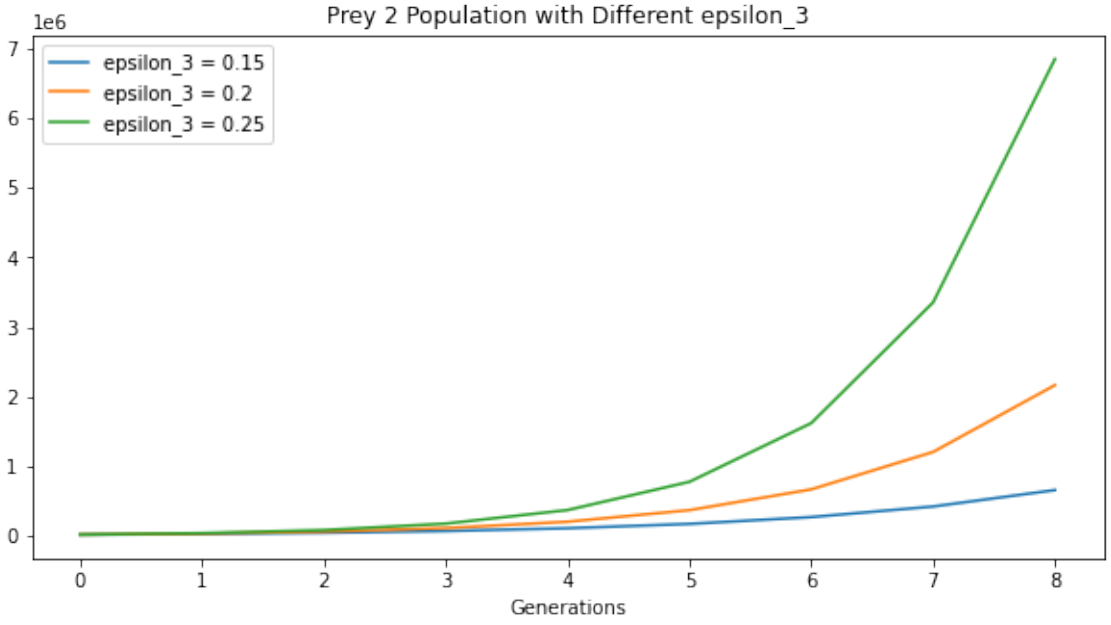
Finally, we vary ϵ_3 in values of $\{0.15, 0.20, 0.25\}$. From the below figures, we see that when the effect of interaction gets stronger, the total population of the prey 3 gets larger. Also, the population of the predator is the largest when the ϵ_3 is the 0.25. When ϵ_3 is 0.25, the population of prey 2 is the largest, which means that the predator has the most food source, since the predator primarily feeds on prey 2.

[27]:
```
candidate_epsilon_3 = [0.15, 0.20, 0.25]

original_epsilon_3 = args.epsilon_3

predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()

args.epsilon_3 = candidate_epsilon_3[0]
predator_density_generation_track_1, prey_1_density_generation_track_1,\
          prey_2_density_generation_track_1 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_3 = candidate_epsilon_3[1]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
predator_density_generation_track_2, prey_1_density_generation_track_2,\
          prey_2_density_generation_track_2 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

args.epsilon_3 = candidate_epsilon_3[2]
predator_population_density, prey_1_population_density, prey_2_population_density =␣
 ↪intialize_population_density()
```

```
predator_density_generation_track_3, prey_1_density_generation_track_3,\
          prey_2_density_generation_track_3 = simulate(predator_population_density,␣
 ↪prey_1_population_density, prey_2_population_density)

# reset D1 to the original value
args.epsilon_3 = original_epsilon_3

plot_population_density_with_different_parameters([predator_density_generation_track_1,␣
 ↪predator_density_generation_track_2,
                                                    ␣
 ↪predator_density_generation_track_3], 'Predator', 'epsilon_3', candidate_epsilon_3)
plot_population_density_with_different_parameters([prey_1_density_generation_track_1,␣
 ↪prey_1_density_generation_track_2,
                                                    prey_1_density_generation_track_3],␣
 ↪'Prey 1', 'epsilon_3', candidate_epsilon_3)
plot_population_density_with_different_parameters([prey_2_density_generation_track_1,␣
 ↪prey_2_density_generation_track_2,
                                                    prey_2_density_generation_track_3],␣
 ↪'Prey 2', 'epsilon_3', candidate_epsilon_3)
```







5.6 Scenario Analysis with Road Network

One of the main goals of this project is to analyze the impact of highways on the population of species under different scenarios (single-species, predator-prey, and multi-species). In the previous sections, we have already discussed the impact of highways in the single-species and the predator-prey scenarios. For

this section, we discuss the influence of highways on the population of species in the presence of multi-species. Specifically, we will investigate the effects of three different kinds of highways on the population size of predator, prey 1 and prey 2.

5.6.1 Utils

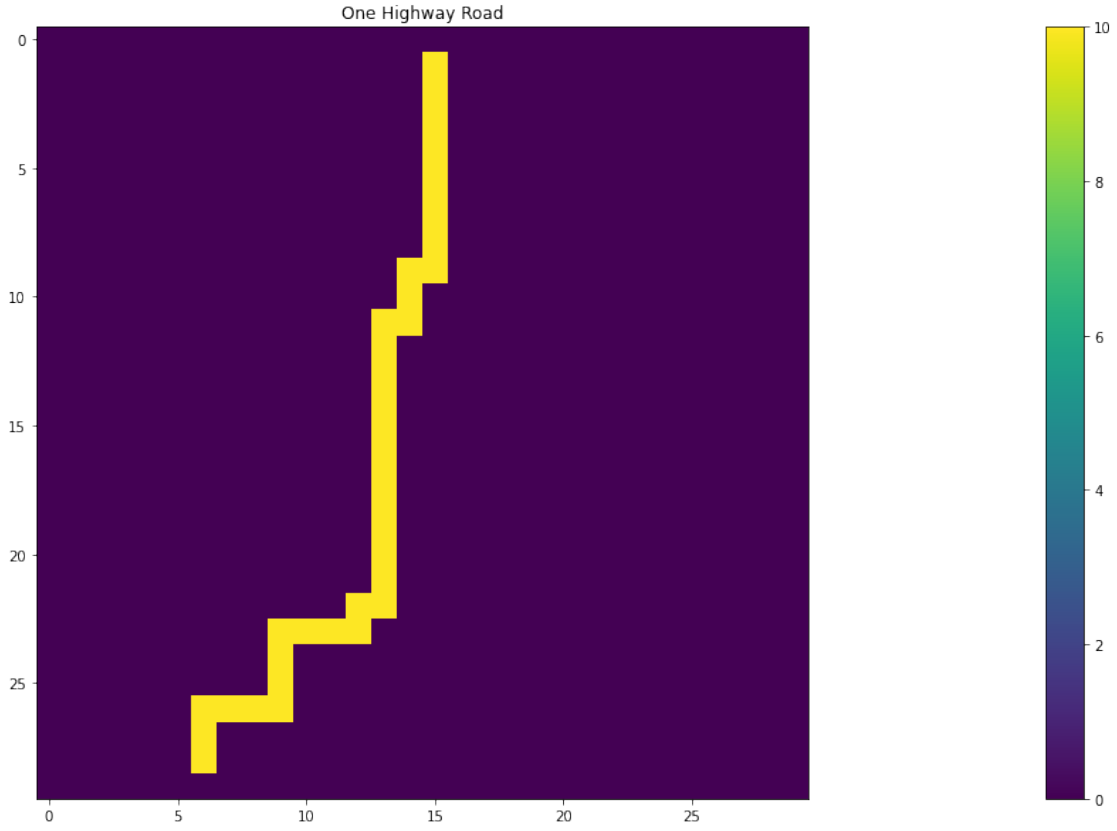
Below are the utility functions that faciliate the highway impact analysis:

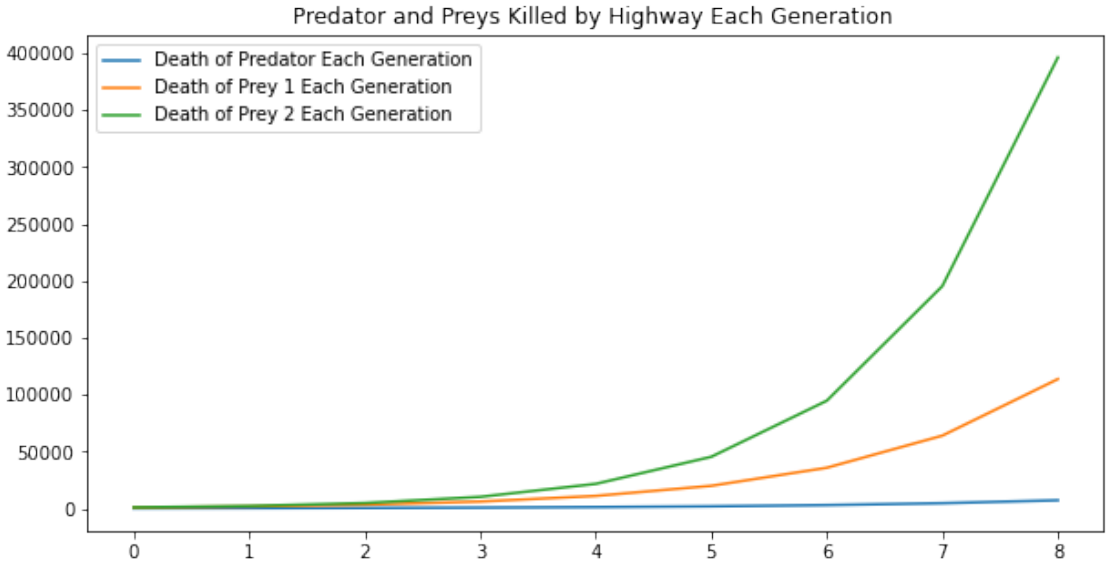
[28]:
```
def calculate_population_killed_by_highway(target_world, world_with_high_way,␣
↪population_per_cell=100):
    death_cnt = 0
    for i in range(args.world_width):
        for j in range(args.world_width):
            if world_with_high_way[i,j] == args.highway_class:
                death_cnt += target_world[i,j] * population_per_cell
    return death_cnt


def get_world_with_highway(num_highways):
    world_with_highway = np.zeros((args.world_width, args.world_width))
    for i in range(num_highways):
        world_with_highway = dataForInput.addHighway2World(world_with_highway,
                                                           args.world_width, args.
↪prob_as_end_of_road,
                                                           args.
↪prob_for_highway_direction, args.highway_class,
                                                           args.random_seed_for_highway)
    return world_with_highway


def plot_death_by_highway(population_generation_tracks, world_with_highway):
    pd.DataFrame({'Death of Predator Each Generation':␣
↪[calculate_population_killed_by_highway(world, world_with_highway) for world in␣
↪population_generation_tracks[0]],
                  'Death of Prey 1 Each Generation':␣
↪[calculate_population_killed_by_highway(world, world_with_highway) for world in␣
↪population_generation_tracks[1]],
                  'Death of Prey 2 Each Generation':␣
↪[calculate_population_killed_by_highway(world, world_with_highway) for world in␣
↪population_generation_tracks[2]]})\
                  .plot(figsize = [10,5], title= 'Predator and Preys Killed by Highway␣
↪Each Generation ')
```

5.6.2 One Highway Scenario

In this case, we have a highway that goes from north to south. Also, it goes staight through the center of the simulating world, where the populations of species are the most concentrated. As can be seen in the figure below, the construction of this highway results the death of 400, 000 individuals belonging to prey 2 species, 1000, 000 individuals belonging to prey 1 species, and roughly 10, 000 belonging to the predator species.

[29]:
```
world_with_highway = get_world_with_highway(num_highways=1)
visualization.displayPattern(world_with_highway, "One Highway Road")
plot_death_by_highway([predator_density_generation_track_3,␣
↪prey_1_density_generation_track_3, prey_2_density_generation_track_3],␣
↪world_with_highway)
```

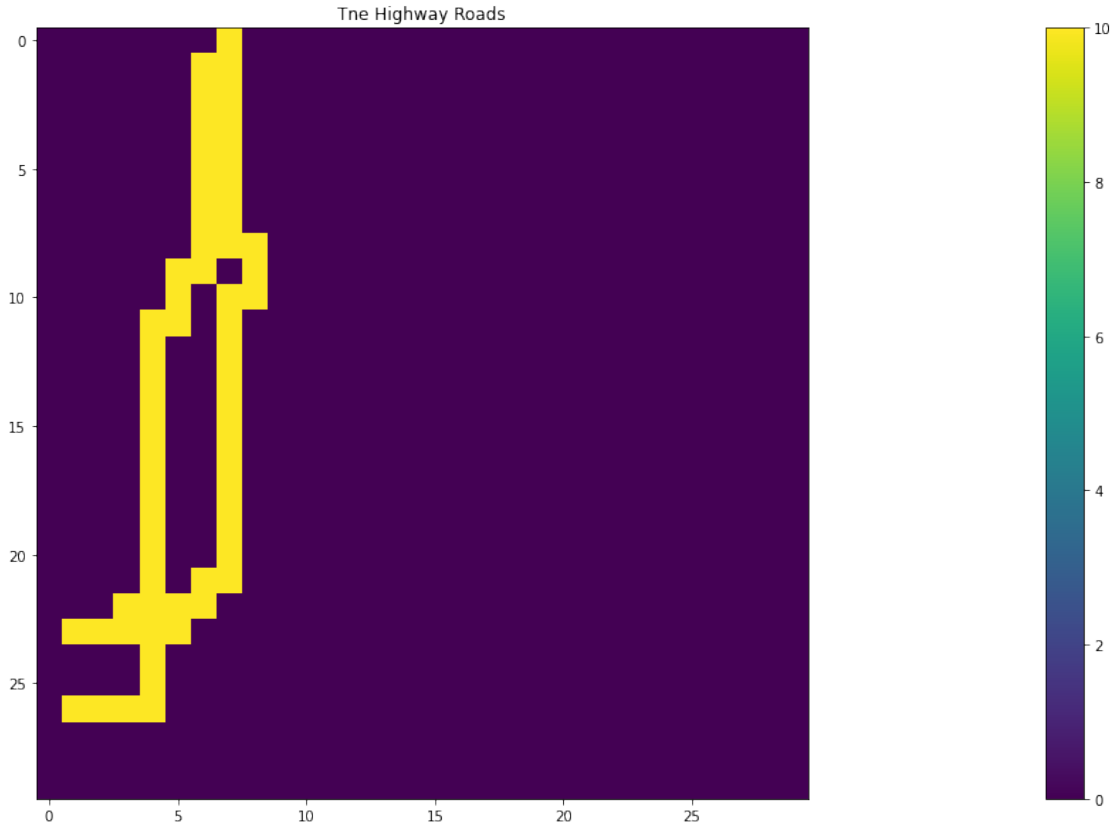


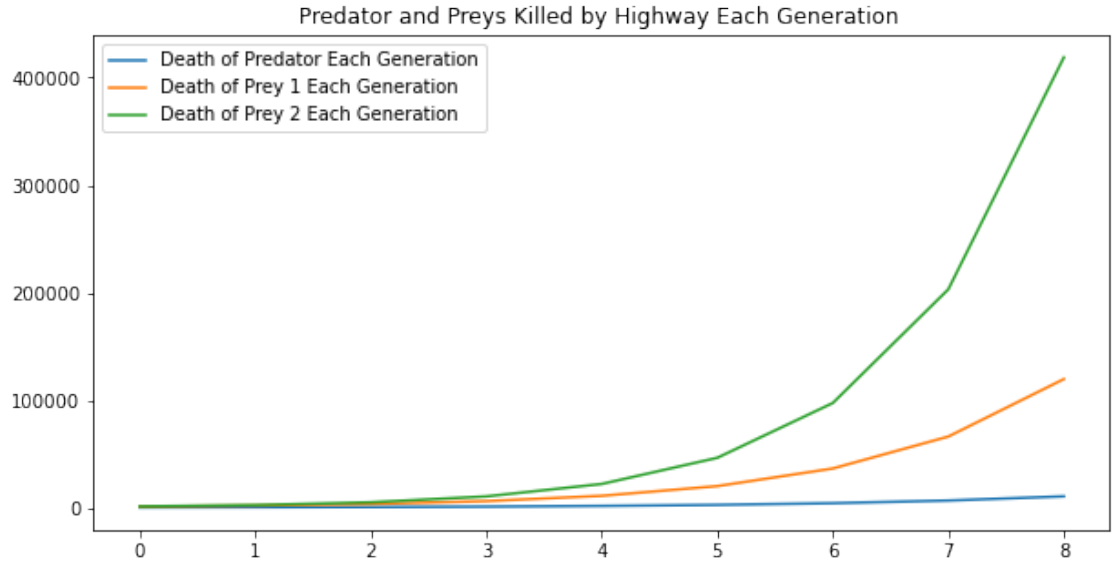


5.6.3 Two Highways Scenario

In this case, we have two highways that go from north to south. Also, they lie adjacent to each other. The dead populations of species result by this configuration of highways are almost the same as that in the previous case. This is likely due to the location of the highway roads. In this case, the highway roads are constructed at the peripheral locations, so it will have negative impact on less populations of species than the highway does in the previous case, where the highway is constructed at the center of the simulating world.

[30]:
```
original_prob_as_end_of_road = args.prob_as_end_of_road

args.prob_as_end_of_road = 0.2

world_with_highway = get_world_with_highway(num_highways=2)
visualization.displayPattern(world_with_highway, "Tne Highway Roads")
plot_death_by_highway([predator_density_generation_track_3,␣
 ↪prey_1_density_generation_track_3, prey_2_density_generation_track_3],␣
 ↪world_with_highway)

args.prob_as_end_of_road = original_prob_as_end_of_road
```

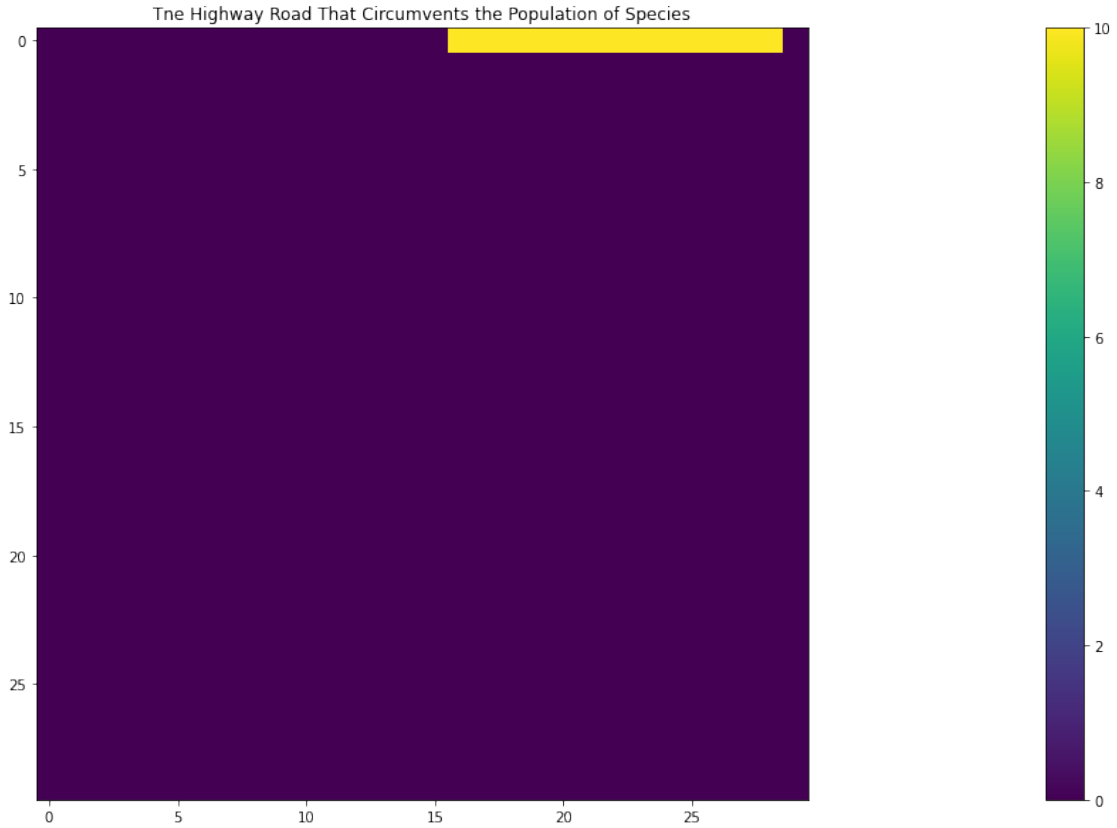

Tne Highway Roads
0
5
10
15
20
25
0
5
10
15
20
25
10
8
6
4
2
0

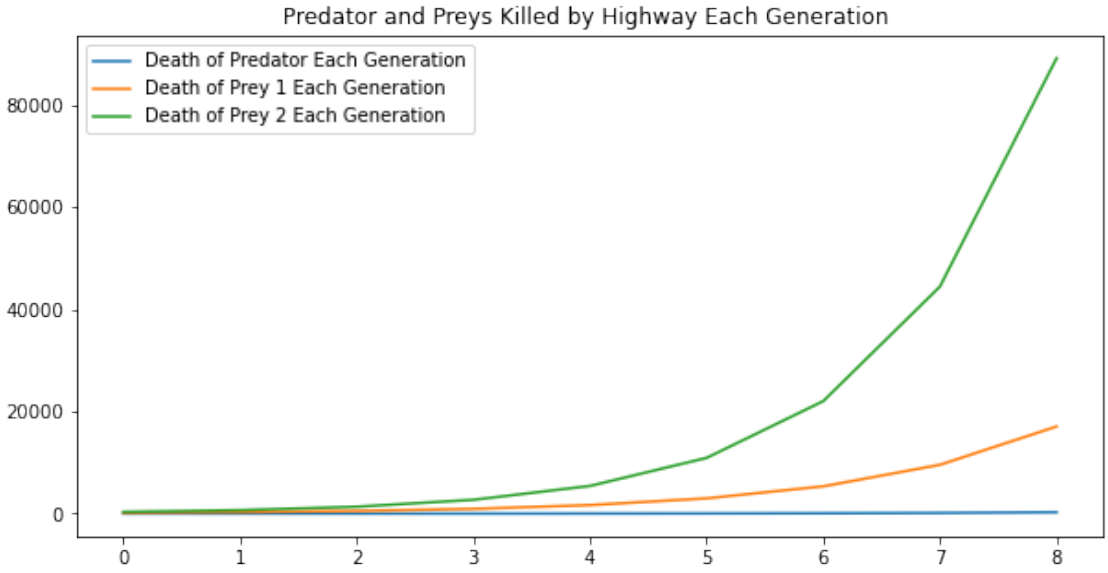

Predator and Preys Killed by Highway Each Generation
Death of Predator Each Generation
Death of Prey 1 Each Generation
Death of Prey 2 Each Generation
400000
300000
200000
100000
0
0
1
2
3
4
5
6
7
8

5.6.4 One Highway Circumventing the Population of Species

In this case, we have one highway that circumvents the main population of the three species. The impact of this highways is less devastating than those in the previous cases, as we can see that the death of Prey 2 population is 5 time less than the death of Prey 2 population when the roads are constructed in the center of the simualting world.

[31]:
```
original_prob_for_highway_direction = args.prob_for_highway_direction
original_random_seed_for_highway = args.random_seed_for_highway

args.prob_for_highway_direction = [0.00, 0.95, 0.05]
args.random_seed_for_highway = 20

world_with_highway = get_world_with_highway(num_highways=1)
visualization.displayPattern(world_with_highway, "Tne Highway Road That Circumvents␣
 ↪the Population of Species")
plot_death_by_highway([predator_density_generation_track_3,␣
 ↪prey_1_density_generation_track_3, prey_2_density_generation_track_3],␣
 ↪world_with_highway)

args.prob_for_highway_direction = original_prob_for_highway_direction
args.random_seed_for_highway = original_random_seed_for_highway
```





Overall, from the above visualization and analysis, we can see that the wise choice of highway is essential for the survival of animal species. The bad choice (e.g., location) of highway construction will have the devastating effects on the natural habitat and result the significant loss of the population of animal species.

6 Discussion

The project aims to find the impacts of road network design on population dynamics. To better simulate the population and environment from the real world, we used three models, including the Single-species model, Predator Vs. Prey model and Species Vs. Species model to simulate three typical scenarios for the population. They are one world with one dominant species without the predator, one world with one predator and one prey, and one world with two dominant species that share the same resource and space. The simulation results of the Single-species model show that the location and development intensity of the road network can significantly affect the population density. It shows that the closer the road to the population habitat and the migration path between the population habitats, the stronger the negative impacts on the population density. It also shows that more road segments can cause more damage to the population. It suggests that road network development should move away from the population habitat and the migration path between the population habitat to reduce the negative impacts on population dynamics.

Compared to the results of the Single-species model simulation, predator-prey simulation shows a very similar results. Basically, the highway will do damage to the predator-prey dynamics. The more highways that are crossing the populations, the more damage it will do. However, if the government builds the highway in a way to circumvent the animal habitat, then it will cause much lesser damage to predators and preys. Additionally, it's noteworthy that the impact of highway road on preys is much severer than that on predators, because the preys usually will have a higher growth rate and can emigrate faster and broader than predators in the same period of time. Thus the highway road will more likely kill more preys than predators over time.

In the multi-species model, we can observe that highway roads kill more preys than predators, consistent with the result in the predator-prey model. Furthermore, we can see that the dead population of predators is rather uniform across generations, compared to the nonlinear, faster growth of prey. This is probably due

to fact that predators are assumed to be evenly distributed. Even though they migrate from time to time, their populations do not differ much from each other site to site. Comparing the sizes of dead populations of prey 1 and prey 2, we can see that the dead population of prey 2 is significantly larger than that of prey 1 due to its higher growth rate and migration rate. It can be speculated that the lower death rate of prey 1 is not only due to the construction of highway roads, but also because of its competitive relationship with prey 2, whose large population size limits the resources for the growth of prey 1, thus reducing its population size in the first place. For the future study, we would like to further investigate how much of the competitive relationship between these two species contributes to the difference in their corresponding dead populations by highway roads.

7 Division of Labor

Kai Qu is in charge of Species Vs. Species modeling, simulation, and visualization; Ge Zhang is in charge of initial population pattern generation, Single-species modeling, simulation, and visualization; Zhe Zheng is in charge of Predator Vs. Prey modeling, simulation, and visualization.

References

[1] Species distribution. https://en.wikipedia.org/wiki/Species_distribution. Accessed: 2021-04-12.

[2] Ali Beykzadeh and James Watmough. An explicit formula for a dispersal kernel in a patchy landscape. *bioRxiv*, page 680256, 2019.

[3] Philip J Clark and Francis C Evans. Distance to nearest neighbor as a measure of spatial relationships in populations. *Ecology*, 35(4):445–453, 1954.

[4] Ross Cressman and Vlastimil Křivan. Two-patch population models with adaptive dispersal: the effects of varying dispersal speeds. *Journal of mathematical biology*, 67(2):329–358, 2013.

[5] David S Fay and Ken Gerow. A biologist's guide to statistical thinking and analysis. *WormBook: the online review of C. elegans biology*, pages 1–54, 2013.

[6] Brian R Hudgens and Nick M Haddad. Predicting which species will benefit from corridors in fragmented landscapes from population growth models. *The American Naturalist*, 161(5):808–820, 2003.

[7] Anne Kandler and Roman Unger. Population dispersal via diffusion-reaction equations, 2010.

[8] R. May. Stability and complexity in model ecosystems. *Princeton University Press*, 1974.

[9] Sergei Petrovskii, Kohkichi Kawasaki, Fugo Takasu, and Nanako Shigesada. Diffusive waves, dynamical stabilization and spatio-temporal chaos in a community of three competitive species. *Japan Journal of Industrial and Applied Mathematics*, 18:459–481, 06 2001.

[10] Santiago Saura and Javier Martínez-Millán. Landscape patterns simulation with a modified random clusters method. *Landscape ecology*, 15(7):661–678, 2000.

[11] Vitaly Volpert and Sergei Petrovskii. Reaction–diffusion waves in biology. *Physics of life reviews*, 6(4):267–310, 2009.