```python
import numpy as np
from scipy import stats as st
from matplotlib import pyplot as plt

class SIR:
    def __init__(self, population, alpha=0.005, beta=0.01, gamma=0.1, years=20):

        self.population = population

        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma

        self.P = np.array([
            [ 1-beta,    beta,       0,      0],
            [      0, 1-gamma,   gamma,      0],
            [  alpha,       0, 1-alpha,      0],
            [      0,       0,       0,      1]
        ])

        self.years = years
        self.totalDays = int(years*365)
        self.X_n = np.zeros((self.totalDays,self.population))

        self.z_alpha = 1.697

    def updatePopulation(self, population):
        self.population = population

    def updateParams(self, alpha, beta, gamma):

        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma

        self.P = np.array([
            [ 1-beta,    beta,       0,      0],
            [      0, 1-gamma,   gamma,      0],
            [  alpha,       0, 1-alpha,      0],
            [      0,       0,       0,      1]
        ])

    def setInitialState(self,S=1000,I=0,R=0,V=0):

        if S+I+R+V != self.population:
            S = self.population - (I+R+V)

        susceptible = np.zeros(S)
        infected = np.ones(I)
        recovered = np.ones(R)*2
        vaccinated = np.ones(V)*3

        self.X_n[:,:] = 0
        self.X_n[0] = np.concatenate((susceptible,infected,recovered,vaccinated))

    def simulate(self):

        for i in range(1,self.totalDays):
            # Double for loop solution
            for j in range(self.population):

                if np.random.random() > self.P[int(self.X_n[i-1,j]),int(self.X_n[i-1,j])]:
                    self.X_n[i,j] = self.X_n[i-1,j] + 1
                else:
                    self.X_n[i,j] = self.X_n[i-1,j]

                if self.X_n[i,j] == 3:
                    self.X_n[i,j] = 0


        for i in range(self.totalDays):
            self.X_n[i].sort()

    def simulateWithDependence(self):

        for i in range(1,self.totalDays):
            # Update parameters to get chance of infection dependant on amount in state 1

            self.updateParams(self.alpha, (0.5*np.sum(self.X_n[i-1] == 1))/self.population ,self.gamma)

            for j in range(self.population):

                if self.X_n[i-1,j] == 3:
                    # Special case if state == 3 (Vaccinated)
                    self.X_n[i,j] = 3

                elif self.X_n[i-1,j] == 2:
                    # Special case if state == 2 (Recovered)
                    if np.random.random() > self.P[int(self.X_n[i-1,j]),int(self.X_n[i-1,j])]:
                        self.X_n[i,j] = 0
                    else:
                        self.X_n[i,j] = 2

                else:
                    # No special cases for states 0 and 1, (Susceptible and Infected)
                    if np.random.random() > self.P[int(self.X_n[i-1,j]),int(self.X_n[i-1,j])]:
                        self.X_n[i,j] = self.X_n[i-1,j] + 1
                    else:
                        self.X_n[i,j] = self.X_n[i-1,j]


        for i in range(self.totalDays):
            self.X_n[i].sort()


    def plot(self):

        plt.figure(0)
        plt.imshow(self.X_n.T)
        plt.show()


    def graphSIR(self, show=True, index=0):

        SIRV = np.zeros((4,len(self.X_n)))
        SIRVlabel = ["Susceptible", "Infected", "Recovered", "Vaccinated"]

        axis = np.linspace(0,len(self.X_n),len(self.X_n))

        for i in range(len(SIRV)):
            for j in range(len(self.X_n)):
                SIRV[i,j] = np.count_nonzero(self.X_n[j] == i)

        plt.figure("SIRV")#+str(index))
        plt.title("SIR-plot")
        for i in range(len(SIRV)-1):
            plt.plot(axis, SIRV[i],label=f"{SIRVlabel[i]}")
```

```
        plt.ylim([0,self.population])
        plt.legend()
        if show:
            plt.show()

    def countStateDays(self, v=True):
        stateFirst = np.sum(self.X_n[int(self.totalDays/2):,0] == 0)
        stateSecond = np.sum(self.X_n[int(self.totalDays/2):,0] == 1)
        stateThird = np.sum(self.X_n[int(self.totalDays/2):,0] == 2)

        if v:
            print(f"Absolute numbers of days in different states: ")
            print(f"S: {stateFirst:8}, I: {stateSecond:8}, R: {stateThird:8}.")

            print(f"Numbers of days in different states per year: ")
            print(f"S: {2*stateFirst/self.years:8}, I: {2*stateSecond/self.years:8}, R: {2*stateThird/self.years:8}.")

            print(f"Relative numbers of days in different states: ")
            print(f"S: {2*stateFirst/self.totalDays:8.2f}, I: {2*stateSecond/self.totalDays:8.2f}, R: {2*stateThird/self.totalDays:8.2f}.")

        return stateFirst,stateSecond,stateThird

    def numericalLimitingDistributions(self, n=30, v=False):

        results = np.zeros((n,3))

        for i in range(n):
            self.simulate()
            results[i] = self.countStateDays(v=False)

        # Calculate error estimates:
        CIs = np.zeros((3,2))

        for i in range(len(CIs)):
            CIs[i,0], CIs[i,1] = st.t.interval(0.95, n-1, loc=np.mean(results[:,i]), scale=st.sem(results[:,i]))


        if v:
            print(f"CIs: ")
            for i in range(len(CIs)):
                print(f"State: {i}, Lower/Upper: {2*CIs[i,0]/self.years:.2f}, {2*CIs[i,1]/self.years:.2f}, size: {np.abs(2*CIs[i,0]/self.years - 2*CIs[i,1]/self.years):.2f}")

        self.CI = CIs

    def findMaxInfected(self):
        maxI_n = 1
        for j in range(len(self.X_n)):
            I_n = np.count_nonzero(self.X_n[j] == 1)
            if I_n > maxI_n:
                maxI_n = I_n
                argmaxI_n = j

        return maxI_n, argmaxI_n



    def findMaxInfectedCIs(self, simulations=100, v=False, states=[50,0,0]):

        maxI = np.zeros(simulations)
        argmaxI = maxI.copy()

        for i in range(len(maxI)):
            if v and i > 0:
                print(f"Working on {i+1} of {len(maxI)}\tMean max I: {np.mean(maxI[:i]):.2f}, Mean argmax I: {np.mean(argmaxI[:i]):2f}", end="\r")
            # Restart simulation
            self.setInitialState(I=states[0], R=states[1], V=states[2])
            self.simulateWithDependence()

            maxI[i], argmaxI[i] = self.findMaxInfected()

        print() # Removes carriage return
        print(f"Mean max I: {np.mean(maxI)}, Mean argmax I: {np.mean(argmaxI)}")

        n = len(maxI)

        CI_maxI = st.t.interval(0.95, n-1, loc=np.mean(maxI), scale=st.sem(maxI))
        CI_argmaxI = st.t.interval(0.95, n-1, loc=np.mean(argmaxI), scale=st.sem(argmaxI))

        print(f"95% CI for max I: \t[{CI_maxI[0]:.3f},\t {CI_maxI[1]:.3f}], diff: {np.abs(CI_maxI[0] - CI_maxI[1]):.3f}")
        print(f"95% CI for arg max I: \t[{CI_argmaxI[0]:.3f},\t {CI_argmaxI[1]:.3f}], diff: {np.abs(CI_argmaxI[0] - CI_argmaxI[1]):.3f}")
```