FYTN03: Computational Physics
Module C Project

Author: Joseph Alexander Binns.

Due: Sunday 17th October 2021.

1 Introduction

The main purpose of this project was to explore and analyse various scenarios surrounding random walkers. The number of dimensions, initial conditions and a variety of other factors were used to facilitate these experiments. The results were compared to theoretical expectations.

2 Theory

2.1 Random walkers

The discussion has so far only considered the closed boundary conditions, such that walkers cannot leave or enter the area being inspected. But what if open boundary conditions were taken. For example, walkers disappearing at one boundary, to reappear on the opposing boundary. This intuitively feels like a better representation of certain situations. Such as when modelling a small region of air particles in a room with a much greater volume.

2.2 Vicious walkers

When random walkers have the added property of mutual annihilation on collision, they are labelled "vicious walkers". Other interactions will also be considered. For example, a population of walkers comprised of infected and susceptible types. With the susceptible walkers becoming infected upon collision with infected walkers. Many such further situations and properties can be added. Your imagination is the limit! I have limited these properties to a probability that collisions do not result in interaction, and a probability that an infected walker dies instead of taking a step.

2.2.1 Theoretical limits

See below the theoretical thermodynamic limits, applicable for $N, V \to \infty$ with closed boundary conditions.

$$\textbf{1D:} \quad \rho(t) = t^{-\frac{1}{2}}. \tag{1}$$

$$\textbf{2D:} \quad \rho(t) = t^{-1}\log(t). \tag{2}$$

$$\textbf{3D:} \quad \rho(t) = t^{-1}. \tag{3}$$

The 1D and 2D cases are said to be fluctuation dominated, such that the experimental findings are not guaranteed to tend towards the theoretical limits, even as $N, V \to \infty$. However, the 3D case is still considered a good fit for the limit.

3 Method

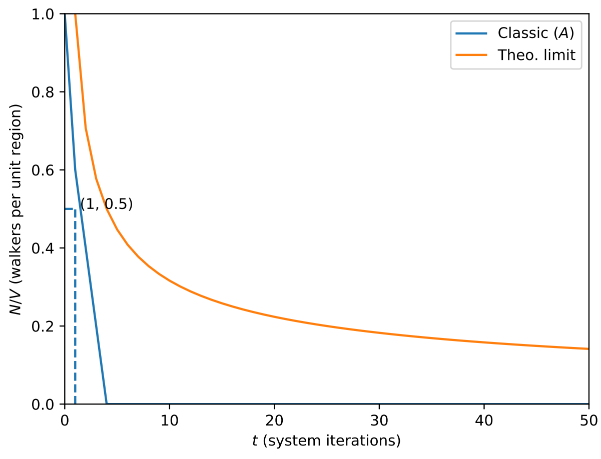
The general procedure for the code is as follows:

1. Generate a list of N vicious walker objects. These are objects belonging to a generic vicious walker class, which can be of any chosen type (i.e. susceptible "S", infected "I" or just the classic vicious walker "A").
2. Create a d dimensional array attributed to a grid object.
3. Position each vicious walker to an empty space in the grid.
4. Begin simulating. Repeatedly iterate through the existing vicious walkers. On each iteration of a given walker, choose a random dimension and direction of the grid to move in. If there exists a boundary in that direction, then check the boundary conditions and either do not move (closed boundary), or move to the opposing boundary on the grid (open boundary). If the position to be moved to is already occupied by a vicious walker, then interact appropriately with consideration to the types of the walkers involved.
5. Upon each iteration across the entirety of the remaining walkers, record the current grid data (for image/animation creation) and any data to be plotted.
6. Stop the iterations depending on the desired stopping condition. For example, one could stop if the simulation grows dormant, such that the number of walkers is unchanging for a certain number of iterations. Or one could stop if a certain total number of iterations is reached.
7. Create any desired animations and plots.

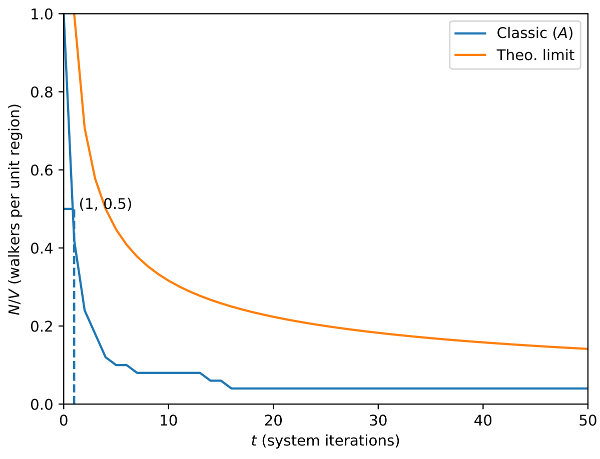
4 Analysis

The GIFs have been upscaled and converted into videos, a compilation of which can be viewed at the following URL, `https://youtu.be/bYLJGYMJn3o`.

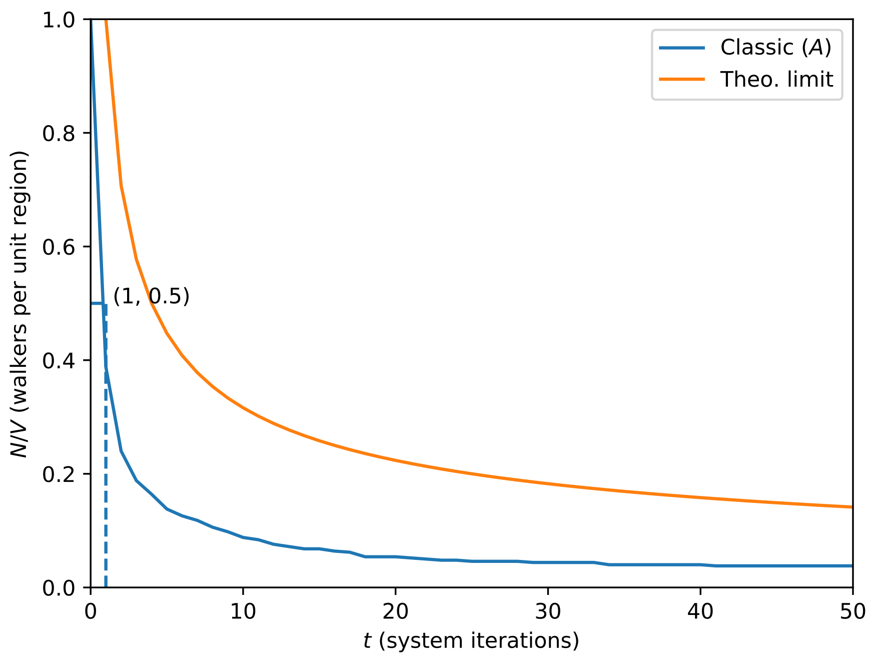
4.1 1D



(a) $L = 10$.



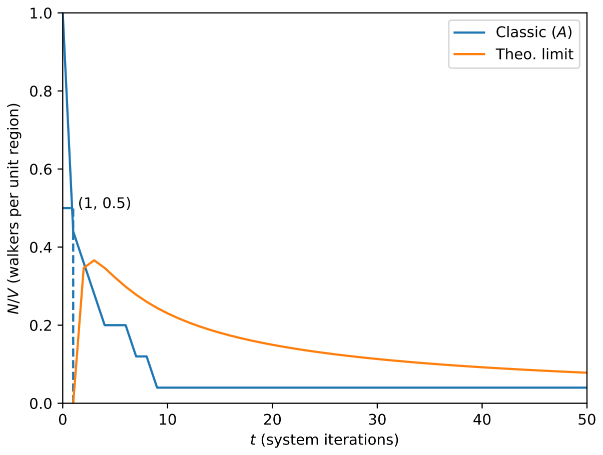
(b) $L = 100$.



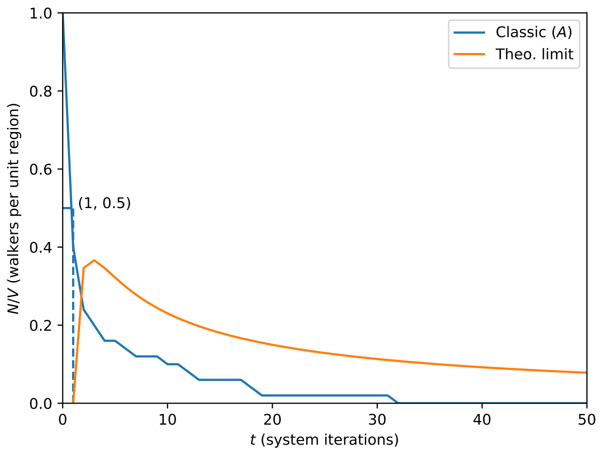
(c) $L = 1000$.

Figure 1: Plots of density against simulation time in the 1D case.

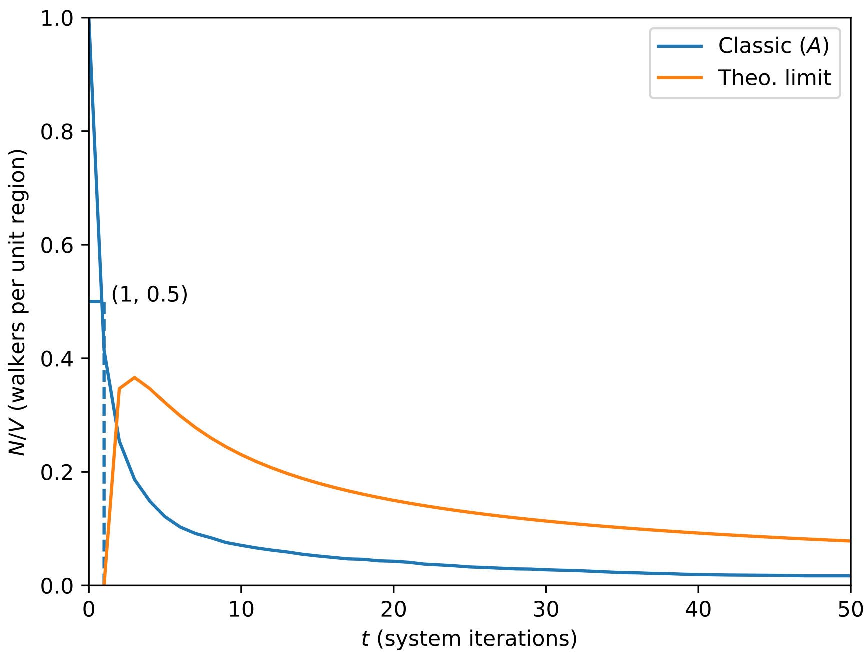
4.2 2D



(a) $L = 5$.



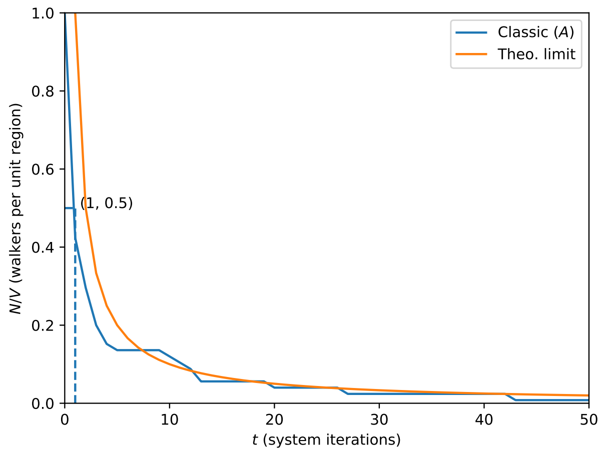
(b) $L = 10$.



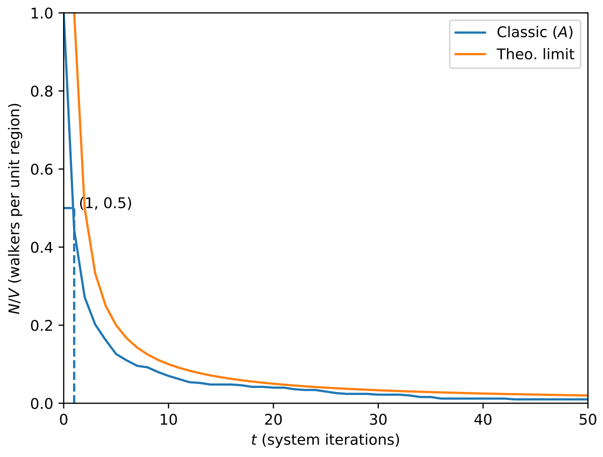
(c) $L = 100$.

Figure 2: Plots of density against simulation time in the 2D case.

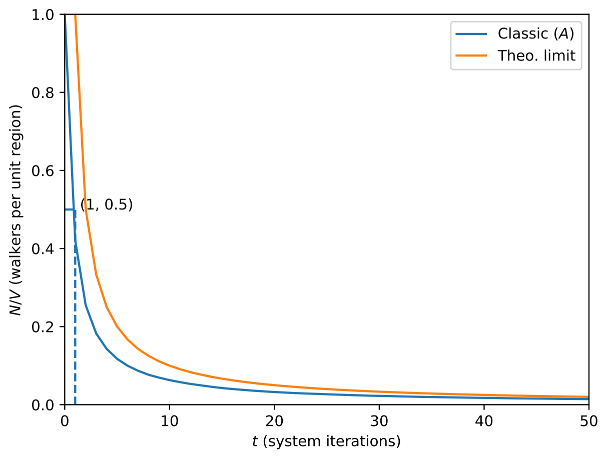
4.3 3D



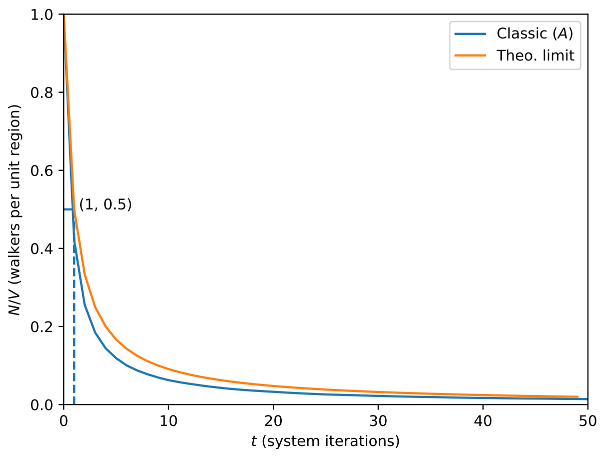
(a) $L = 5$.



(b) $L = 10$.



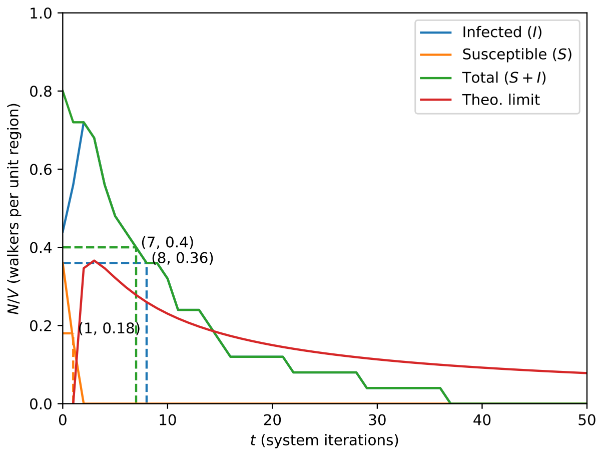
(c) $L = 50$.



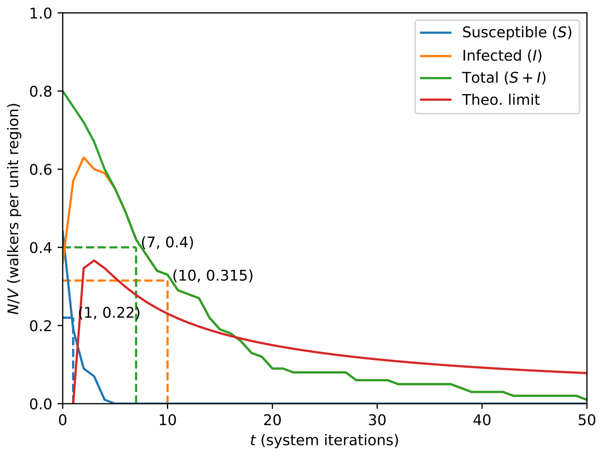
(d) $L = 50$, theoretical y intercepted shifted from ∞ to 1 by moving the graph along the x axis.

Figure 3: Plots of density against simulation time in the 3D case.

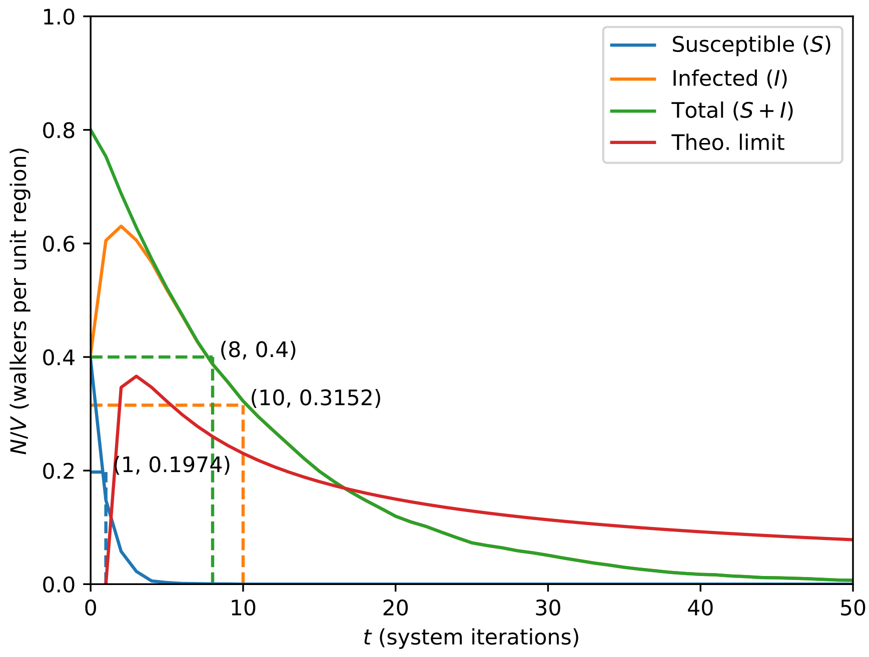
4.4 Zombies! (Infected and Susceptible)



(a) $L = 5$.



(b) $L = 10$.



(c) $L = 50$.

Figure 4: Plots of density against simulation in the 2D "zombies" case.

5 Discussion

The theoretical limits have been seen to roughly behave as expected. The 1D and 2D cases seen in figures 1 and 4 have proved poor fits. This isn't surprising, considering that the limits to these cases are less applicable due to the systems being considered fluctuation dominated. In all the standard (non-zombie) cases, the theoretical observation is seen to develop at a faster rate than the theoretical limit. It could be argued that in the 1D case, the data seems to gradually tend towards the limit as L is increased. A hypothetical explanation for this could be risen and tested, such as a greater volume resulting in greater times between collisions, and thus a raised graph. However, I am not entirely convinced that the data is pointing in this direction. Take for example the 3D case seen in figure 3. The same trend is not seen, with the $L = 5$ and $L = 10$ cases laying slightly above the $L = 50$ case. I wondered if the fact that the theoretical limit intercepts the y axis at infinity is the reason for the slight miss-match. My belief being that the limited range of the experimental density scale must inevitably cause a displacement from the theoretical results. The experimental density scale is limited to $[0.0, 1.0]$, with 1.0 being the limit in which a walker saturates every position in the region. I tried shifting the theoretical limit along the x axis, such that it's y intercept was now roughly at 1.0. The result can be seen in figure 3d. The theoretical limit does not become a perfect for the experimental data, but certainly some progress is made. It may just be that different implementations of the algorithm could yield slightly different results. A deeper dive into the derivations of the theoretical limits would be required to uncover what might be causing the discrepancy!

The half lives of the respective populations are plotted and their coordinates labelled in each of the graphs. They are mostly underwhelming, with the constant theme of the half life being reached after just one iteration of the whole system. This is the case for all but the zombies scenario. Here, the infected number first raises before then falling, resulting in a slower process and non trivial half times!

Using a length of L = 100.
For the 1D case, the time to simulate was $T \approx 2$ s. With a comparatively negligible preparation time.
For the 2D case, initially fully occupied with L = 100. The time to simulate was $T \approx 750$ s, with a preparation time of less than a second. The increase in simulation time is a factor of 375. Whereas I expected to just see an increase in time by the same factor as the volume increase, 100.
The time taken for the 3D case can therefore be expected to increase by at least another factor of 100. Which would $75,000$ s (1250 m) simulation time. Obviously this is not very feasible, and so there is definitely plenty of room for improving the efficiency of the code. My initial thinking is that the efficiency here could be improved by optimising the current method. For example, the grid class and the walker class are both updated whenever movements or interactions are made. There is potential for improvement by utilising the proper object oriented methodology and uniquely storing data just on (i.e.) walker objects, which can then be separately called by the grid object when appropriate.

Some problems remain in the code, with "hacky" solutions. For instance, the animation creation package produces mysteriously faulty results when applied to the 1D case. This has been quick-fixed by checking if it's the 1D case, and saving the raw images instead of directly creating an animation in this instance. This works OK, as the animations are very low resolution and blurry when typically viewed, such that they need to be up-scaled anyway. This final operation is already being done in the Aseprite[1] pixel-art program, which I can conveniently also use to animate the raw images.

The latest changes made to the code were in improving the time efficiency for randomly placing particles throughout the grid. Originally, particles were attempted to be placed at a random position repeatedly until an empty position was found. When fully filling a grid, the worst case scenario, there is a resulting $O(L^d)$ complexity. This was resolved by creating a 1D array of available indices, which could be translated to and from grid coordinates. Whenever a grid index became occupied, it was removed from the available array. Giving perfect results of $O(1)$. The improvements, along with some general cleaning up of the processes, have brought down the previously stated simulation times by a few factors, but there is still a lot of room for progress.

A criticism to myself in the Module A (ODE) Project was that I could have done better if I didn't shy away from object oriented programming. I'm glad that I made sure to embrace object oriented programming in this project, as I have developed my programming skills and broadened my comfort zone in doing so.

6 Conclusion

It has been seen how various flavours of random walkers can be implemented and simulated computationally, with the application to various situations such as random vibrations of molecules in a gas, and zombie apocalypses. A variety of configurations have been run and analysed, with their behaviour confirmed by a combination of visual and graphical findings. The graphical findings have been compared alongside theoretical expectations, with imperfect fits being questioned as a topic of further study. The project has done well to develop my ability to implement computational physics into programming.

References

[1] Igara Studio S.A. Aseprite. `https://www.aseprite.org/`. Accessed 2021-10-13.

Appendix

Python code

```python
import random as rand
import numpy as np
from PIL import Image
import imageio
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import copy
import math
from pprint import pprint
import os
import time

''' ~~~~~~~~~~~~~ GENERATE A NEW DIRECTORY TO SAVE IMAGES TO ~~~~~~~~~~~~~~ '''
i = 0
while True:
    imagesDir = "images_"+str(i)
    try:
        os.makedirs(imagesDir)
        break
    except:
        pass
    i += 1

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~ DEFINE FUNCTIONS ~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
def CreateImage(data, imgs):
    if grid.get_d() == 1:
        img = Image.fromarray(data, 'RGB')
        img = img.convert('RGB')
        saveDir = imagesDir+"/img"+str(len(imgs))+".png"
        img.save(saveDir)
    elif grid.get_d() == 3:
        pass
    imgs.append(data)

def RecordXY(x, y, xArray, yArray):
    xIncluded = False
    if xArray[-1] == x:
        xIncluded = True
    if xIncluded == False:
        xArray.append(x)
    yArray.append(y)
    return(xArray, yArray)

def UnloadPreset(preset):
    name = preset[0]
    L = preset[1]
    d = preset[2]
    bC = preset[3]
    N = preset[4]
    iP = preset[5]
    dP = preset[6]
    types = preset[7]
    mS = preset[8]
    imgPerParticle = preset[9]
```

```
    return name, L, d, bC, N, iP, dP, types, mS, imgPerParticle

def GenDir(d):
    axis = rand.randint(0, d-1)
    tempDir = []
    for i in range(d):
        if(i == axis):
            dirPositive = bool(rand.getrandbits(1))
            if (dirPositive == True):
                tempDir.append(+1)
            else:
                tempDir.append(-1)
        else:
            tempDir.append(0)
    tempDir = np.array(tempDir)
    return(tempDir)

def GenNumpyArrayDims(numDimensions, systemLength, extra = None):
    tempList = []
    if (numDimensions == 1):
        tempList.append(systemLength)
        tempList.append(numDimensions)
        if (extra != None):
            tempList.append(extra)
    else:
        for i in range(0, numDimensions):
            tempList.append(systemLength)
        if (extra != None):
            tempList.append(extra)
    tempList = tuple(tempList)
    return tempList

def PosToNumpyInd(pos):
    tempIndex = []
    #print(pos)
    for i in range(len(pos)):
        tempIndex.append([pos[i]])
    return tempIndex

''' DEPRECATED
def IsIn(toCheck, pool):
    for i in range(len(pool)):
        if (toCheck == pool[i]):
            return True
    return False
'''

def GenNDimPos(d, L):
    tempPos = []
    for i in range(d):
        randPosi = rand.randint(0, L-1)
        tempPos.append(randPosi)
    return(tempPos)

''' NOT USED
def PosToInd(pos, L, d):
    M = L
    S = 0
    if d == 1:
        x = pos[0]
        S = x
    elif d == 2:
        x, y = pos[0], pos[1]
        S = M * (y-1) + x
    elif d == 3:
        x, y, z = pos[0], pos[1], pos[2]
        S = M**2 * (z-1) + M * (y-1) + x
    ind = S
    return(ind)
'''

def IndToPos(ind, L, d):
    M = L
    pos = []
    S = ind
    if d == 1:
        x = S
        pos = [x]
    elif d == 2:
        y = 1 + math.floor((S-1)/M)
        x = S - M * (y-1)
        pos = [x, y]
    elif d == 3:
        z = 1 +  math.floor((S-1)/M**2)
        y = 1 +  math.floor((S - M**2 * (z-1) - 1)/M)
        x = S - M**2 * (z-1) - M * (y-1)
        pos = [x, y, z]
    return (pos)

def ValidatedInput(inputQueue, answerRange, dtype = int):
    '''
    Input(s):
        inputQueue {str}: The text for the question.
        answerRange {length-2 array}: The range of numbers of acceptable values.
        dtype {type}: The expected data-type.
    Output(s):
        theInput {dtype}: Validated input.
    '''
    while True: # Check that the input is a number, and then check that the number is within the desired range
     for answers.
        try:
            theInput = dtype(input(inputQueue))
            while True: # Check that the input, which has already been established to be a number, is within
     the desired range for answers.
                if (theInput >= answerRange[0] and theInput <= answerRange[1]):
                    break # Escape the while loop if the input is within the range.
                else:
                    theInput = dtype(input("Input␣was␣not␣within␣the␣range␣of␣the␣options.␣Please␣re-select:␣")
     )
            break # Escape the while loop if the input is interpreted as a NUMBER within the RANGE.
        except:
            print("Input␣could␣not␣be␣interpreted␣as␣a␣number.␣HELP:␣Type␣the␣desired␣number␣ONLY␣and␣press␣
     enter.")
    return theInput

def ChooseConfig():
    options = ""
    for i in range(0, len(presets)):
        options += str(i+1)+".␣"+presets[i][0]+"\n"
    options += "Select:␣"
    ind = ValidatedInput("Please␣enter␣the␣number␣of␣your␣chosen␣option:␣\n"+options, [1, len(presets)]) - 1

    config = UnloadPreset(presets[ind])

    options = ""
    for i in range(0, len(presetVars)):
        options += presetVars[i]+":␣"+str(config[i])+".\n"
    options += "1.␣confirm\n"+"2.␣deny\n"
    options += "Select:␣"
    confirmInd = ValidatedInput("Here␣are␣the␣selected␣constants.␣Do␣you␣confirm?␣\n"+options, [1, 2])

    if confirmInd != 1:
        print("\nPlease␣start␣again.")
        exit()

    return(config)

def FindClosestIndex(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx

def TheoLim(t, d):
    t0 = np.inf
    if d == 1:
        newT = t[1:]**(-1/2)
    elif d == 2:
        newT = t[1:]**(-1) * np.log(t[1:])
    elif d == 3:
        newT = t[1:]**(-1)
    newT = np.concatenate([[t0], newT])
    return newT
```

```
''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~ DEFINE CLASSES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
class Grid:
    def __init__(self, systemLength, numDimensions, boundCondition):
        self.L = systemLength
        self.d = numDimensions
        self.bC = boundCondition

        self.N = 0
        self.particles = []
        self.t = 0
        self.types = []

        self.V = self.L ** self.d
        self.gridDims = GenNumpyArrayDims(numDimensions, systemLength)
        self.gridImgDims = GenNumpyArrayDims(numDimensions, systemLength, extra = 3)
        self.grid = np.ndarray(self.gridDims, dtype=np.object)
        self.grid.fill(0)

    def get_t(self):
        return self.t

    def set_t(self, t):
        self.t = t

    def get_V(self):
        return self.V

    def get_NoverV(self, particleType = None):
        NoverV = self.get_N(particleType) / self.get_V()
        return NoverV

    def get_d(self):
        return self.d

    def set_N(self, N):
        self.N = N

    def get_N(self, particleType = None):
        N = 0
        if (particleType == None):
            N = self.N
        else:
            for particle in self.particles:
                if particle.get_type() == particleType:
                    N += 1
                else:
                    pass
        return N

    def add_particle(self, particle):
        self.set_grid(particle.get_pos(), particle)
        self.particles.append(particle)
        self.set_N(self.get_N() + 1) # start particle indexing from 1... not 0. (for reasons of when printing
     the grid)
        particle.set_ind(self.N)

    def remove_particle(self, particle):
        self.set_grid(particle.get_pos(), 0)
        self.particles.remove(particle)
        self.set_N(self.get_N() - 1)

    def set_grid(self, pos, newVal):
        self.grid[PosToNumpyInd(pos)] = newVal

    def get_grid(self):
        return self.grid

    def get_grid_for_image(self, recentParticle = None):
        tempGrid = np.ndarray(self.gridImgDims, np.uint8)
        tempGrid.fill(255)
        for particle in self.particles:
            if particle.get_type() == "A":
                tempGrid[PosToNumpyInd(particle.get_pos())] = [0,0,0]
            elif particle.get_type() == "S":
                tempGrid[PosToNumpyInd(particle.get_pos())] = [0,255,0]
```

```
            elif particle.get_type() == "I":
                tempGrid[PosToNumpyInd(particle.get_pos())] = [255,0,0]

            if (particle == recentParticle):
                colour = tempGrid[PosToNumpyInd(particle.get_pos())][0]
                if (self.get_d() == 1):
                    colour = colour[0]
                if (self.get_types()[0] == "A"):
                    colour = [255, 0, 0]
                else:
                    for i in range(3):
                        if colour[i] != 0:
                            colour[i] = 200
                tempGrid[PosToNumpyInd(particle.get_pos())] = [colour]
        return tempGrid

    def in_bounds(self, pos):
        for i in range(self.d):
            if (0 <= pos[i] and pos[i] < self.L):
                pass
            else:
                return (i)
        return(-1)

    def simulate_particle(self, particle):
        # Check if decay
        decayed = particle.decay(self)
        if decayed == None:
            return

        # Check a step
        oldPos = particle.get_pos()
        step = GenDir(self.d) # Randomly select a direction of walk, and take a step.

        # Handle possible overflow over boundaries
        testPos = particle.get_translate_pos(step)
        isOverflow = self.in_bounds(testPos)
        if (isOverflow != -1):
            if self.bC == 0: # Fixed boundary condition, don't move if step overflows.
                return
            else: # Periodic boundary condition, loop position.
                for i in range(d):
                    if (i == isOverflow):
                        testPos[i] = (L-1) - oldPos[i]

        # Check if collision.
        objectAtStep = self.get_grid()[PosToNumpyInd(testPos)][0]
        if (self.get_d() == 1):
            objectAtStep = objectAtStep[0]
        isCollision = (objectAtStep != 0 and objectAtStep != particle)

        if isCollision:
            otherParticle = objectAtStep
            # Interact
            particle.interact(otherParticle, self)
            otherParticle.interact(particle, self)
            return # Don't move.

        # Update particle position
        newPos = testPos
        particle.set_pos(newPos)

        # Update grid
        for particle2 in self.particles: # Shouldn't really be necessary... but for some reason is broken
 without.
            if particle == particle2:
                self.set_grid(oldPos, 0)
                self.set_grid(newPos, particle)


    def get_types(self):
        for particle in self.particles:
            included = False
            for currType in self.types:
                if currType == particle.get_type():
                    included = True
            if included == False:
                self.types.append(particle.get_type())
        return self.types

    def simulate(self, tMax = 100, shouldCreateImage = False, imagePerParticle = False, tArray = [],
     NoverVArrays = []):
        self.set_t(self.get_t() + 1)
        if tMax != -1:
            if self.get_t() > tMax:
                return

        for particle in self.particles:
            if shouldCreateImage and imagePerParticle:
                CreateImage(self.get_grid_for_image(particle), imgs)

            self.simulate_particle(particle)

            if shouldCreateImage and imagePerParticle:
                CreateImage(self.get_grid_for_image(particle), imgs)

        if shouldCreateImage and imagePerParticle == False:
            CreateImage(self.get_grid_for_image(), imgs)

        types = grid.get_types()
        for i in range(0, len(types)):
            if len(types) != len(NoverVArrays):
                print("Number␣of␣types␣does␣not␣match␣number␣of␣NoverV␣arrays!")
            RecordXY(grid.get_t(), grid.get_NoverV(types[i]), tArray, NoverVArrays[i])

    def __str__(self):
        tempGrid = np.ndarray(self.gridDims, np.int16)
        tempGrid.fill(0)
        for particle in self.particles:
            tempGrid[PosToNumpyInd(particle.get_pos())] = particle.get_ind()
        return str(tempGrid)

class ViciousWalkerGeneric():
    def __init__(self, position, walkerType, interactProb, decayProb):
        self.pos = position
        self.type = walkerType
        self.interactProb = interactProb
        self.decayProb = decayProb

    def set_type(self, newType):
        self.type = newType

    def get_type(self):
        return self.type

    def set_pos(self, newPosition):
        self.pos = newPosition

    def get_translate_pos(self, translation):
        testPos = []
        for i in range(len(self.pos)):
            testPos.append(self.pos[i] + translation[i])
        return testPos

    def translate_pos(self, translation):
        self.set_pos(self.get_translate_pos(translation))

    def get_pos(self):
        return self.pos

    def set_ind(self, index):
        self.ind = index

    def get_ind(self):
        return self.ind

    def decay(self, grid):
        if (rand.uniform(0, 1) <= self.decayProb):
            if (self.get_type() == "I"):
                grid.remove_particle(self)
                return True
        else:
            return False

    def interact(self, other, grid):
        if (rand.uniform(0, 1) <= self.interactProb):
            if (self.get_type() == "A"):
                if (other.get_type() == "A"):
                    grid.remove_particle(self)
                else:
                    pass
            elif (self.get_type() == "I"):
                if (other.get_type() == "S"):
                    other.set_type("I")
                else:
                    pass
            elif (self.get_type() == "S"):
                if (other.get_type() == "I"):
                    self.set_type("I")
                else:
                    pass
        else:
            pass

    def __str__(self):
        return str(self.get_ind())

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~ DECLARE VARIABLES ~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
imgs = []

presets = [["1D␣small", 10, 1, 0, int(1*10), 1, 0, ["A"], -1, True],
            ["1D␣big", 100, 1, 0, int(1*100), 1, 0, ["A"], -1, True],
            ["1D␣huge", 1000, 1, 0, int(1*1000), 1, 0, ["A"], -1, False],
            ["2D␣tiny", 5, 2, 0, int(1*(5**2)), 1, 0, ["A"], -1, True],
            ["2D␣small", 10, 2, 0, int(1*(10**2)), 1, 0, ["A"], -1, True],
            ["2D␣big", 100, 2, 0, int(1*(100**2)), 1, 0, ["A"], -1, False],
            ["3D␣tiny", 5, 3, 0, int(1*(5**3)), 1, 0, ["A"], 50, False],
            ["3D␣small", 10, 3, 0, int(1*(10**3)), 1, 0, ["A"], 50, False],
            ["3D␣big␣[WARNING:␣SLOW␣(~240␣s)]", 50, 3, 0, int(1*(50**3)), 1, 0, ["A"], 50, False],
            ["2D␣zombie␣apocalypse␣tiny", 5, 2, 1, int(0.8*(5**2)), 1, 0.1, ["S", "I"], -1, True],
            ["2D␣zombie␣apocalypse␣small", 10, 2, 1, int(0.8*(10**2)), 1, 0.1, ["S", "I"], -1, True],
            ["2D␣zombie␣apocalypse␣big", 50, 2, 1, int(0.8*(50**2)), 1, 0.1, ["S", "I"], -1, False]]

presetVars = ["name", "length␣(L)", "dimensions␣(d)", "boundary␣condition␣(0␣=␣closed,␣1␣=␣open)", "initial␣
     number␣of␣walkers␣(N)", "interaction␣chance", "decay␣chance", "walker␣types", "time␣steps", "create␣an␣
     image␣per␣movement␣(False␣is␣per␣entire␣step)"]

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CHOOSE PRESET ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
preset = ChooseConfig()
name, L, d, bC, N, interactProb, decayProb, initTypes, maxSteps, imagePerParticle = UnloadPreset(preset)

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ START TIMER ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
startTime = time.perf_counter()

''' ~~~~~~~~~~~~~~~~~~~~~~ DECLARE WORKING VARIABLES ~~~~~~~~~~~~~~~~~~~~~~ '''
tArray = [0]
NoverVArrays = []
for i in range(0, len(initTypes)):
    NoverVArrays.append([])

shouldCreateImage = True
if d > 2:
    shouldCreateImage = False

grid = Grid(L, d, bC)

remIndList = np.arange(1, L**d + 1)
for n in range(N):
    if(n >= L**d):
        print("N␣greater␣than␣grid␣volume")
        break

    randInd = rand.randint(0, len(remIndList) - 1)
    tempInd = remIndList[randInd]
    remIndList = np.delete(remIndList, randInd)

    tempPos = IndToPos(tempInd, L, d)
    tempPos = np.asarray(tempPos) - 1
```

```
    randomType = initTypes[rand.randint(0, len(initTypes) - 1)]
    grid.add_particle(ViciousWalkerGeneric(tempPos, randomType, interactProb, decayProb))

''' ~~~~~~~~~~~~~~~~~~~~~~ RECORD INITIAL CONDITIONS ~~~~~~~~~~~~~~~~~~~~~~ '''
CreateImage(grid.get_grid_for_image(), imgs)
types = grid.get_types()
for i in range(0, len(types)):
    if len(types) != len(NoverVArrays):
        print("Number␣of␣types␣does␣not␣match␣number␣of␣NoverV␣arrays!")
    RecordXY(grid.get_t(), grid.get_NoverV(types[i]), tArray, NoverVArrays[i])

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ TIMER CHECK ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
prepFinTime = time.perf_counter()
print("\npreparation␣time:␣", prepFinTime - startTime, "s.")

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~ RUN SIMULATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
prevNoverV = grid.get_NoverV()
dormantCount = 0
while True:
    if maxSteps != -1:
        if grid.get_t() >= maxSteps:
            break

    grid.simulate(tMax = maxSteps, shouldCreateImage = shouldCreateImage, imagePerParticle = imagePerParticle,
     tArray = tArray, NoverVArrays = NoverVArrays)

    NoverV = grid.get_NoverV()
    if maxSteps == -1:
        if NoverV == prevNoverV:
            dormantCount += 1
            if dormantCount >= 5000:
                break
        else:
            dormantCount = 0
    prevNoverV = NoverV

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ TIMER CHECK ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
simFinTime = time.perf_counter()
print("\nsimulation␣time:␣", simFinTime - prepFinTime, "s.")

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~ CREATE ANIMATION ~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
fps = math.ceil(2.5/60 * len(imgs))
if len(imgs) > 0:
    if grid.get_d() == 2:
        imageio.mimwrite(imagesDir+"/animated_from_images.gif", imgs, loop = 1, fps = fps) # save gif

''' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CREATE PLOT ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ '''
leg = []
types = grid.get_types()
for i in range(0, len(types)):
    theType = "Unidentified␣type␣(NA)"
    if types[i] == "A":
        theType = "Classic␣($A$)"
    elif types[i] == "S":
        theType = "Susceptible␣($S$)"
    elif types[i] == "I":
        theType = "Infected␣($I$)"
    leg.append(theType)

halfLives = []
totalPop = np.zeros(len(tArray))
for NoverVArray in NoverVArrays: # Creating density plots.
    x = tArray
    y = NoverVArray

    p = plt.plot(x, y, "-")

    totalPop = np.add(totalPop, y)

    halfLives.append(max(y)/2)
    halfLifeX = x[FindClosestIndex(y, halfLives[-1])]
    plt.hlines(y = halfLives[-1], xmin = 0, xmax = halfLifeX, linestyles = "--", colors = p[0].get_color())
    plt.vlines(x = halfLifeX, ymin = 0, ymax = halfLives[-1], linestyles = "--", colors = p[0].get_color())
    plt.annotate("␣("+str(round(halfLifeX, 4))+",␣"+str(round(halfLives[-1], 4))+")", (halfLifeX, halfLives
     [-1]))
```

```
if (len(NoverVArrays) > 1): # Creating a summed density plot.
    x = tArray
    y = totalPop
    p = plt.plot(x, y, "-")
    leg.append("Total␣($S␣+␣I$)")

    halfLives.append(max(y)/2)
    halfLifeX = x[FindClosestIndex(y, halfLives[-1])]
    plt.hlines(y = halfLives[-1], xmin = 0, xmax = halfLifeX, linestyles = "--", colors = p[0].get_color())
    plt.vlines(x = halfLifeX, ymin = 0, ymax = halfLives[-1], linestyles = "--", colors = p[0].get_color())
    plt.annotate("␣("+str(round(halfLifeX, 4))+",␣"+str(round(halfLives[-1], 4))+")", (halfLifeX, halfLives
     [-1]))

# Plotting thermodynamic limit.
x = tArray
y = TheoLim(np.asarray(tArray, dtype = float), grid.get_d())

# Hacking to make it fit
#x = np.asarray(x) - 1

plt.plot(x, y)
leg.append("Theo.␣limit")

plt.xlabel("$t$␣(system␣iterations)")
plt.ylabel("$N/V$␣(walkers␣per␣unit␣region)")
plt.legend(leg)
plt.xlim([0, tArray[-1]])
plt.ylim([0, 1])
plt.show()
plt.savefig(imagesDir+"/graph.pdf")
```