Agent-Based Simulation of Crowd Flocking & Evacuation Behavior

Team 5

Emily Chen

Napa Vananupong

William Vaughn

Github:

https://github.gatech.edu/echen67/CX4230

The tutorial is formatted as a Jupyter notebook in the github linked above. We have pasted the contents of the notebook below per the instructions, but please view part 1, 2, and 3 in the github to see them as they are meant to look and to view them as interactive notebooks.

Tutorial: Agent-Based Simulation of Crowd Flocking & Evacuation Behavior (BOIDS Problem)

This 3-part tutorial illustrates different ways to model the of simulation of crowd evacuation behaviour, which is sometimes referred to as the BOIDS problem. The parts, in brief, are as follows:

- Part 1 (Setup): We'll show how a working 2D BOIDS model can be modeled over a geometric region using Realtime Simulation GUI for PyCX (Copyright 2012 Chun Wong & Hiroki Sayama)
- Part 2: The agent-based simulation of crowd flocking behaviour. This part explains how to use and implement a realtime simulation GUI for PyCX for modeling crowd behaviour. You can think of this method as a way to analyze the dynamics of the BOIDS model without having to run many simulations. This part shows how our Agent.py models these behaviours and how to create one.
- Part 3: The agent-based simulation of evacuation behaviour

Part 1: A Simple Boid Simulation

This is the main program for running a flocking simulation in python
It utilizes 2 objects: a boid (birdlike - behaving object) and a 3D Cartesian vector
To see the objects and their respective methods, you can look at boid.py
Part 1: A Simple Boid Simulation

Conceptual model:

Each boid has direct access to the whole scene's geometric description, but flocking requires that it reacts only to flockmates within a certain small neighborhood around itself. The neighborhood is characterized by a distance (measured from the center of the boid) and an angle, measured from the boid's direction of flight. Flockmates outside this local neighborhood

are ignored. The neighborhood could be considered a model of limited perception (ie. birds in foggy skies) but it is probably more correct to think of it as defining the region in which flockmates influence a boids steering.
This is the main program for running a flocking simulation in python It utilizes 2 objects: a boid (birdlike - behaving object) and a 3D Cartesian vector To see the objects and their respective methods, you can look at boid.py

Setup

Run the following code cells to get everything set up for the boid simulation
In [ ]:

```
import matplotlib
from IPython.display import clear_output
%matplotlib inline
import pylab as PL
import random as RD
import numpy as NP
from scipy.spatial import distance as dist
from Agent import Agent

RD.seed()

populationSize = 64
noiseLevel = 1
avoidanceRadius = 30
flockRadius = 100
boardDimension = 1000

#strengths are proportion of new weight vs proportion of old weight (0-1)
avoidanceStrength = 0.4
approachStrength = 0.6
alignStrength = 0.8
totalStrength = 0.2

def init():
  global time, agents

  time = 0

  agents = []
  for i in range(populationSize):
     row = i % 8.0
```

```
        col = i / 8.0
        newAgent = Agent(row*50.0+50.0, col*50.0+50.0, RD.gauss(0, noiseLevel), RD.gauss(0,
noiseLevel), 0, 0)
        agents.append(newAgent)

def draw():
    x = [ag.posX for ag in agents]
    y = [ag.posY for ag in agents]
    PL.axis('scaled')

    PL.cla()
    PL.axis([0, boardDimension, 0, boardDimension])
    PL.title('t = ' + str(time))
    PL.plot(x, y, 'bo')

    PL.pause(0.03)

def step():
    """Update positions of each agent each time step"""
    global time, agents

    time += 1

#     d = dist.euclidean([agents[0].posX, agents[0].posY], [agents[1].posX, agents[1].posY])
#     print(d)

    for ag in agents:
        colVect = ag.collisions(ag, avoidanceRadius, agents)
        avgLoc, avgVel = ag.getFlock(agents, flockRadius)
        alignVect = ag.align(avgVel, populationSize)
        apprVect = ag.approach(avgLoc)

        weightTot = avoidanceStrength + alignStrength + approachStrength
        weights = [avoidanceStrength / weightTot, alignStrength / weightTot, approachStrength /
weightTot]
        ag.newVelX = (1 - totalStrength) * ag.oldVelX + totalStrength * (colVect[0] * weights[0] +
alignVect[0] * weights[1] + apprVect[0] * weights[2])
        ag.newVelY = (1 - totalStrength) * ag.oldVelY + totalStrength * (colVect[1] * weights[0] +
alignVect[1] * weights[1] + apprVect[1] * weights[2])

        # Update positions using new velocities
        # Last number changes speed on screen
        ag.posX += ag.newVelX * 0.2
```

```
    ag.posY += ag.newVelY * 0.2

    # Wraps agents around when they leave the screen
    # And shifts new weights to old weight position
    for ag in agents:
        ag.posX = ag.posX % boardDimension
        ag.posY = ag.posY % boardDimension
        ag.oldVelX = ag.newVelX
        ag.oldVelY = ag.newVelY
```

. . .

<h3>Start Simulation</h3>
Run this to see the simulation in action

Start Simulation

Run this to see the simulation in action
In [ ]:

```
init()
while(True):
  clear_output(wait=True)
  draw()
  step()
```

Part 2: This part explains how to use and implement a realtime simulation GUI for PyCX for modeling crowd behaviour. You can think of this method as a way to analyze the dynamics of the BOIDS model without having to run many simulations. This part shows how our Agent.py models these behaviours and how to create one.

The Boids Simulation Problem: To model swarming schooling behavior, so long as you follow these three basic rules:Move in the same direction as your closest neighbors.

Don’t stray off by yourself - stay close
But not too close. Avoid collisions with your neighbors.

The phenomenon to be modeled and simulated:

Suppose we wish to model the crowd flocking and evacuation behaviour in a population distributed with the above three basic rules. Each individual boid must move in the same direction as its closest neighbors, meaning that the model exhibits a 'follow the leader' sort of behaviour. The second rule is to not stray off by oneself, meaning the flock of boids must move in a group, combining this with the first rule gives a flock that always faces the same direction. The third rule is to not be to close as to avoid collision with other boids. These are the main things addressed in part 1.

Conceptual model

Separation: steer to avoid crowding local flockmates
Alignment: steer towards the average heading of local flockmates
Cohesion: steer to move toward the average position of local flockmates

Let's start by importing the necessary files and creating the agent class

The following code cell defines the boundaries x and y for the old and the new agent, first initialising them.
In the Method __init__ we initialise the parameters with new values.
Let's start by importing the necessary files and creating the agent class
The following code cell defines the boundaries x and y for the old and the new agent, first initialising them. In the Method **init** we initialise the parameters with new values.
In [1]:

```
import matplotlib
import pylab as PL
import random as RD
import numpy as NP
from scipy.spatial import distance as dist
from sklearn import preprocessing

class Agent:

  posX = 0.0
  poxY = 0.0
  oldVelX = 0.0
  oldVelY = 0.0
  newVelX = 0.0
```

```
  newVelY = 0.0
  flock = []

  def __init__(self, posx, posy, oldvelx, oldvely, newvelx, newvely):
      self.posX = posx
      self.posY = posy
      self.oldVelX = oldvelx
      self.oldVelY = oldvely
      self.newVelX = newvelx
      self.newVelY = newvely
. . .
```

This method takes care of finding the average position of the flock so that we can use this to implement our flocking behaviour. It is neccessary for the implementation of the align rule, which causes boids that are part of the same flock to have the same general direction. For every flockmate within the alignment range, a boid will feel a force to match its heading to that of the flockmate. If there are multiple flockmates in the alignment range, the boid tries to move towards the average direction of those flockmates. The boid in the center wants to align itself with each of these flockmates.

This method takes care of finding the average position of the flock so that we can use this to implement our flocking behaviour. It is neccessary for the implementation of the align rule, which causes boids that are part of the same flock to have the same general direction. For every flockmate within the alignment range, a boid will feel a force to match its heading to that of the flockmate. If there are multiple flockmates in the alignment range, the boid tries to move towards the average direction of those flockmates. The boid in the center wants to align itself with each of these flockmates.

In [2]:

```
def approach(self, flock):
      """Get average position of the flock"""
      distX = flock[0] / len(flock) - self.posX
      distY = flock[1] / len(flock) - self.posY
      return [distX * 0.1, distY * 0.1]
. . .
```

Implementation of the Align rule:

Alignment attempts to change a boid's velocity (i.e., direction and magnitude) to match that of its nearby boids (using the same definition of "nearby" as before). Cohesion drives each boid towards its nearest neighbours, but there's a couple of notable problems we ran into :boids moving in substantially different directions tend to perform dramatic fly-bys, rather than "flocking" together: the cohesive force in the above example isn't enough to slow boids moving in opposing directions. Once the boids pass each other, they no longer "see" each other, so the turning stops.

given that a boid's visible range extends mostly in its direction of travel, the cohesive forces generated will predominantly increase its speed.
Implementation of the Align rule: Alignment attempts to change a boid's velocity (i.e., direction and magnitude) to match that of its nearby boids (using the same definition of "nearby" as before). Cohesion drives each boid towards its nearest neighbours, but there's a couple of notable problems we ran into :boids moving in substantially different directions tend to perform dramatic fly-bys, rather than "flocking" together: the cohesive force in the above example isn't enough to slow boids moving in opposing directions. Once the boids pass each other, they no longer "see" each other, so the turning stops. given that a boid's visible range extends mostly in its direction of travel, the cohesive forces generated will predominantly increase its speed.
In [3]:

```
def align(self, avgVel, populationSize):
    """Get average direction of flock"""
    alignX = avgVel[0] / populationSize
    alignY = avgVel[1] / populationSize

    return [alignX, alignY]
. . .
```

This method makes sure the boids do not collide: if they get too close, we redistribute theirdistance via proxVect.
Avoidance radiaus is the radius around the agent where the proximity is deemed too close and the agent will try to move away. Suppose the member of a flock is aiming directly for the center of the obstacle object.
1. We define P, C, r, and V as follows:

P = (1, 1)

C = (3, 3)

r = 0.5

V = (1, 1)

2. Determine if a collision might occur

s = |C – P|

= |(3, 3) – (1, 1)|

= |(2, 2)|

= (22 + 22) ½

= 8½

= 2.828

k = (C – P) • V/ |V |

= ((3,3) – (1, 1)) • (1, 1)/|(1, 1)|

= (2, 2) • (1, 1)/ (12 + 12) ½

= (2, 2) • (1, 1)/ (2) ½

= (2, 2) • (0.707, 0.707)

= 1.414+ 1.414

= 2.828

t = (s2 – k2) ½

= (2.8282 – 2.8282) ½

= 0

t < r

0 < 0.5 is true, so we have a hit. In fact, we will have a hit for any radius. This would be the definition of a collision.
This method makes sure the boids do not collide: if they get too close, we redistribute theirdistance via proxVect. Avoidance radiaus is the radius around the agent where the proximity is deemed too close and the agent will try to move away. Suppose the member of a flock is aiming directly for the center of the obstacle object.

1. We define P, C, r, and V as follows:

P = (1, 1)
C = (3, 3)
r = 0.5

V = (1, 1)

1. Determine if a collision might occur

s = |C – P|
= |(3, 3) – (1, 1)|

= |(2, 2)|

= (22 + 22) ½

= 8½

= 2.828
k = (C – P) • V/ |V |
= ((3,3) – (1, 1)) • (1, 1)/|(1, 1)|

= (2, 2) • (1, 1)/ (12 + 12) ½

= (2, 2) • (1, 1)/ (2) ½

= (2, 2) • (0.707, 0.707)

= 1.414+ 1.414

= 2.828
t = (s2 – k2) ½
= (2.8282 – 2.8282) ½

= 0
t < r
0 < 0.5 is true, so we have a hit. In fact, we will have a hit for any radius. This would be the definition of a collision.
In [4]:

```
def collisions(self, ag, avoidanceRadius, agents):
    closestRad = avoidanceRadius
    proxVect = [self.oldVelX, self.oldVelY]
    for other in agents:
        if other.posX != self.posX and other.posY != self.posY:
            distBtw = dist.euclidean([self.posX, self.posY], [other.posX, other.posY])
            if distBtw < closestRad:
                #avoid that one
```

```
                closestRad = distBtw
                proxVect = [(self.posX - other.posX) * (avoidanceRadius - distBtw),
                        (self.posY - other.posY)  * (avoidanceRadius - distBtw)]
        return proxVect
. . .
```

Forms the flock and positions them and returns their average location and velocity to keep track of their flock movement. (flocking behaviour).
flock centering

· members try to stay as close as possible to the center of the flock or school

· centering the flock locally reduces computational complexity

· this type of tendency enables the flock to re-group if separated by the collision avoidance tendency

Forms the flock and positions them and returns their average location and velocity to keep track of their flock movement. (flocking behaviour). flock centering
· members try to stay as close as possible to the center of the flock or school
· centering the flock locally reduces computational complexity
· this type of tendency enables the flock to re-group if separated by the collision avoidance tendency
In [5]:

```
def getFlock(self, agents, flockRadius):
        """Get average location and velocity of nearby agents"""
        """Get agents nearby current agent - we only care about these ones"""
        avgLoc = [0,0]
        avgVel = [0,0]
        for other in agents:
            if dist.euclidean([self.posX, self.posY], [other.posX, other.posY]) < flockRadius:
                avgLoc[0] += other.posX
                avgLoc[1] += other.posY
                avgVel[0] += other.oldVelX
                avgVel[1] += other.oldVelY

        return [avgLoc, avgVel]
. . .
```

Potential Collision Detection and Avoidance

- simulating the cognition (thinking) of the group members, makes them more natural appearing than the pure physics-based approach because the group members sense the environment in front of themselves and plan for it

- sensing: a flock member could emit virtual “feelers” to sample the environment around itself
- model each potentially colliding object with a bounding sphere or the silhouette edges of the collision object
- when a potential collision is detected, run a “steer-to-avoid” procedure

Potential Collision Detection and Avoidance

- simulating the cognition (thinking) of the group members, makes them more natural appearing than the pure physics-based approach because the group members sense the environment in front of themselves and plan for it
- sensing: a flock member could emit virtual “feelers” to sample the environment around itself
- model each potentially colliding object with a bounding sphere or the silhouette edges of the collision object
- when a potential collision is detected, run a “steer-to-avoid” procedure

In [6]:

```
def avoidObstacles(self, obstacles, avoidanceRadius, flag):
    closestRad = avoidanceRadius
    proxVect = [self.oldVelX, self.oldVelY]
    for other in obstacles:
        if other[0] != self.posX and other[1] != self.posY:
            distBtw = dist.euclidean([self.posX, self.posY], [other[0], other[1]])
            if distBtw < closestRad:
                #avoid that one
                closestRad = distBtw
                proxVect = [(self.posX - other[0]) * (avoidanceRadius - distBtw),
                            (self.posY - other[1])  * (avoidanceRadius - distBtw)]
                flag[0] = True
    return proxVect
. . .
```

Finish up Agent model and normalize
Finish up Agent model and normalize
In [9]:

```
def goal(self, goalPos):
    goalVect = [0,0]
```

```
        goalVect = self.normalize(goalPos[0]-self.posX, goalPos[1]-self.posY)
        strength = 50000/dist.euclidean([self.posX, self.posY], [goalPos[0], goalPos[1]])
        goalVect = [strength * goalVect[0], strength * goalVect[1]]
        return goalVect

def normalize(self, x, y):
    mag = float(NP.sqrt(x*x + y*y))
    newX = x/mag
    newY = y/mag
    return [newX, newY]
```

Part 3: Agent Crowd Based Evacuation Behavior

In this part, we modeled the boids evacuation behaviour using cellular automata.

Conceptual Model:

The particle's movement probability grid is modified by the static and dynamic fields. The probability for each action is computed by: pij = NMijDijSij Dij is the modifier value from the dynamic field. Sij is the modifier value from the static field. N is a normalization factor to insure probabilities sum to 1. Note: Formula for pij does not have to be a simple product. Generalized as pij = N f(Mij, Dij, Sij). f(…) may include exponentials for different behavior.
In [ ]:

```
import matplotlib
#matplotlib.use("qt4agg")
import pylab as PL
import random as RD
import numpy as NP
from numpy import linalg as LA
from scipy.spatial import distance as dist
from Agent import Agent
from math import sqrt
from IPython.display import clear_output
%matplotlib inline

matplotlib.rcParams['figure.figsize'] = [5, 5]
```

```
RD.seed()

populationSize = 64
noiseLevel = 5
avoidanceRadius = 50
flockRadius = 100
boardDimension = 1000

#strengths are proportion of new weight vs proportion of old weight (0-1)
avoidanceStrength = 0.8
approachStrength = 0.5
alignStrength = 0.3
obstacleStrength = 1
goalStrength = 1
totalStrength = 0.1

flag = [False]
. . .

<h3>Rules for Dynamic Field CA:</h3>
If a pedestrian leaves a cell (x, y) the dynamic floor field Dxy corresponding to this cell is increased by ΔDxy.
A virtual trace is left by the motion of pedestrians.
A certain amount of the field is distributed among the neighboring cells to model diffusion.
Diffusion is necessary because pedestrians do not necessarily follow exactly in the footsteps of others.
The field strength is reduced by a decay constant δ to model decay of the field.
Implies that the lifetime of the trace is finite.
```

Rules for Dynamic Field CA:

If a pedestrian leaves a cell (x, y) the dynamic floor field Dxy corresponding to this cell is increased by ΔDxy. A virtual trace is left by the motion of pedestrians. A certain amount of the field is distributed among the neighboring cells to model diffusion. Diffusion is necessary because pedestrians do not necessarily follow exactly in the footsteps of others. The field strength is reduced by a decay constant δ to model decay of the field. Implies that the lifetime of the trace is finite.
In [ ]:

```
def init1():
  global time, agents, walls, goalPos
```

```
  time = 0

  scenario1()
. . .
```

Scenario 1: There is one exit for the agents to leave from, in the goal position (goalpos [500,100])
Scenario 1: There is one exit for the agents to leave from, in the goal position (goalpos [500,100])
In [ ]:

```
def scenario1():
  global agents, walls, goalPos, goals
  agents = []
  for i in range(populationSize):
    row = i % 8
    col = i / 8
    newAgent = Agent(row*50+300, col*50+300, RD.gauss(0, noiseLevel), RD.gauss(0, noiseLevel), 0, 0)
    agents.append(newAgent)

  walls = []
  for i in range(60):
    newWall = [i*5+100, 100]
    walls.append(newWall)
  for i in range(61):
    newWall = [i*5+600, 100]
    walls.append(newWall)
  for i in range(160):
    newWall = [i*5+100, 900]
    walls.append(newWall)
  for i in range(160):
    newWall = [100, i*5+100]
    walls.append(newWall)
  for i in range(160):
    newWall = [900, i*5+100]
    walls.append(newWall)

  goals = 1
  goalPos = [500,100]
. . .
```

Draw the grid.

Draw the grid.
In [ ]:

```
def draw():
  PL.cla()
  PL.plot([wall[0] for wall in walls], [wall[1] for wall in walls], 'ro')
  x = [ag.posX for ag in agents]
  y = [ag.posY for ag in agents]
  PL.plot(x, y, 'bo')

  PL.axis('scaled')
  PL.axis([0, boardDimension, 0, boardDimension])
  PL.title('t = ' + str(time))
  PL.pause(0.01)
. . .
```

Step function updates positions of each boid every time step and changes their respective velocities. Once each boid agent has reached the target they will 'escape', which is the escape function and will be effectively removed.
Step function updates positions of each boid every time step and changes their respective velocities. Once each boid agent has reached the target they will 'escape', which is the escape function and will be effectively removed.
In [ ]:

```
def step():
  """Update positions of each agent each time step"""
  global time, agents, walls, goalPos, flag

  time += 1

  for ag in agents:
    colVect = ag.collisions(ag, avoidanceRadius, agents)
    avgLoc, avgVel = ag.getFlock(agents, flockRadius)
    alignVect = ag.align(avgVel, populationSize)
    apprVect = ag.approach(avgLoc)
    avoidVect = ag.avoidObstacles(walls, avoidanceRadius, flag)
    goalVect = ag.goal(goalPos)


    # Limit the max velocity
    limit = 50
    if ag.oldVelX > limit:
      ag.oldVelX = sqrt(ag.oldVelX - limit) + 0.8 * limit
```

```
    if ag.oldVelY > limit:
        ag.oldVelY = sqrt(ag.oldVelY - limit) + 0.8 * limit

    # If there are multiple exits, agents should move to the closest one
    if goals == 2:
        d1 = dist.euclidean([ag.posX, ag.posY], goalPos)
        d2 = dist.euclidean([ag.posX, ag.posY], goalPos2)
        if d1 < d2:
            goalVect = ag.goal(goalPos)
        else:
            goalVect = ag.goal(goalPos2)
        escape(goalPos2)
    escape(goalPos)

    weightTot = avoidanceStrength + alignStrength + approachStrength + obstacleStrength +
goalStrength
    weights = [avoidanceStrength / weightTot, alignStrength / weightTot, approachStrength /
weightTot, obstacleStrength / weightTot, goalStrength / weightTot]

    ag.newVelX = (1 - totalStrength) * ag.oldVelX + totalStrength * (colVect[0] * weights[0] +
alignVect[0] * weights[1] + apprVect[0] * weights[2] + avoidVect[0] * weights[3] + goalVect[0] *
weights[4])
    ag.newVelY = (1 - totalStrength) * ag.oldVelY + totalStrength * (colVect[1] * weights[0] +
alignVect[1] * weights[1] + apprVect[1] * weights[2] + avoidVect[1] * weights[3] + goalVect[1] *
weights[4])

    # Update positions using new velocities
    # Last number changes speed on screen
    ag.posX += ag.newVelX * 0.1
    ag.posY += ag.newVelY * 0.1

    # Wraps agents around when they leave the screen
    # And shifts new weights to old weight position
    for ag in agents:
        ag.posX = ag.posX % boardDimension
        ag.posY = ag.posY % boardDimension
        ag.oldVelX = ag.newVelX
        ag.oldVelY = ag.newVelY
. . .
```

The Escape method removes the agents when they leave successfully:
The Escape method removes the agents when they leave successfully:
In [ ]:

```
def escape(goalPos):
  global agents
  for ag in agents:
    if ag.posX < 100 or ag.posX > 900 or ag.posY < 100 or ag.posY > 900:
      agents.remove(ag)


def normalize(x, y):
  mag = float(NP.sqrt(x*x + y*y))
  newX = x/mag
  newY = y/mag
  return [newX, newY]
. . .

<h3>Scenario 2:</h3> There are two exits for the agents to leave from
1. ) in the goal position (goalpos [500,100])
2. ) in the goal position 2 (goalpos2 [500,900])
```

Scenario 2:

There are two exits for the agents to leave from 1.) in the goal position (goalpos [500,100]) 2.) in the goal position 2 (goalpos2 [500,900])

In [ ]:

```
def scenario2():
  global agents, walls, goalPos, goalPos2, goals
  agents = []
  for i in range(populationSize):
    row = i % 8.0
    col = i / 8.0
    newAgent = Agent(row*50+300, col*50+300, RD.gauss(0, noiseLevel), RD.gauss(0,
noiseLevel), 0, 0)
    agents.append(newAgent)

  walls = []
  for i in range(60):
    newWall = [i*5+100, 100]
    walls.append(newWall)
  for i in range(61):
    newWall = [i*5+600, 100]
    walls.append(newWall)
```

```
  for i in range(60):
    newWall = [i*5+100, 900]
    walls.append(newWall)
  for i in range(61):
    newWall = [i*5+600, 900]
    walls.append(newWall)
  for i in range(160):
    newWall = [100, i*5+100]
    walls.append(newWall)
  for i in range(160):
    newWall = [900, i*5+100]
    walls.append(newWall)

  goals = 2
  goalPos = [500, 100]
  goalPos2 = [500, 900]

def init2():
  global time, agents, walls, goalPos

  time = 0

  scenario2()

. . .
```

<h3>Run A Scenario</h3>
itit1() will run scenario 1. To run scenario 2, change to init2()

Notice that the time it takes each frame to load increases as the program runs, this is because the 'lag' is caused by all of the calculations that are being done in real time so when there are less agents, there are less calculations to do.

Run A Scenario

itit1() will run scenario 1. To run scenario 2, change to init2()
Notice that the time it takes each frame to load increases as the program runs, this is because the 'lag' is caused by all of the calculations that are being done in real time so when there are less agents, there are less calculations to do.
In [ ]:

```
init1()
while(True):
```

```
    clear_output(wait=True)
    draw()
    step()
```

Group work division:

The Boids Problem - William
Adapting Boids Problem to Evacuation - Emily
Jupyter Notebook Tutorial Complilation - Napa