Computational Portfolio

Ziyuan Gao

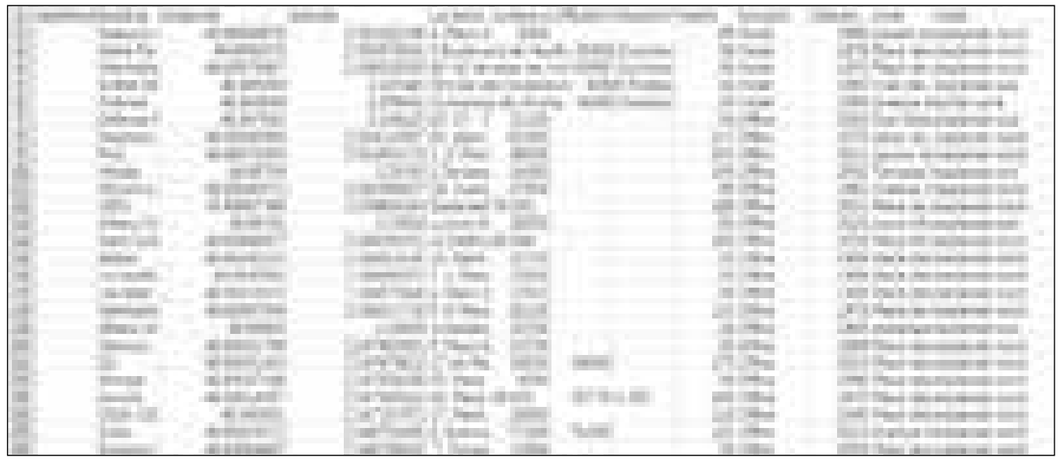
Data Network in Python

Grab geodata directly from the web : display dynamics web maps inside Blender 3d view, requests for OpenStreetMap data (buildings, roads ...), get true elevation data from the NASA SRTM mission.

STEP1: Define a class of ladefensebuildings with its various attributes including the building's name, function, etc. In Python every object is an instance of a class. The class defines the characteristics an object.

```
class ladefensebuilding:
    def __init__(self,name,function,height,delivery,zone,node,longitude,latitude):
        self.building = building
        self.function = function
        self.height = height
        self.delivery = delivery
        self.zone = zone
        self.node = node
        self.longitude = longitude
        self.latitude = latitude
```

STEP2: Import and read csv files



```
with open("ladefensebuildingfunction.csv", encoding='mac_roman') as csvfile:
    reader = csv.reader(csvfile)
    rows = 0
    for r in reader:
        rows = rows + 1
```

```
ladefensebuildings=[]
with open("ladefensebuildingfunction.csv", encoding='mac_roman') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        building = row['Building']
        function = row['function']
        height = int(row['height'])
        delivery = int(row['Delivery'])
        zone = row['zone']
        node = row['node']
        longitude = float(row['longitude'])
        latitude = float(row['latitude'])
        ladefensebuildings.append(ladefensebuilding(building,function,height,delivery,zone,node,longitude,lat
```

STEP3: Assign latitude and longitude to each node

```
for P in ladefensebuildings:
    ladefensebuildinggraph.add_node(P, pos = (P.longitude, P.latitude))
```

```
position=nx.get_node_attributes(ladefensebuildinggraph,'pos')
```

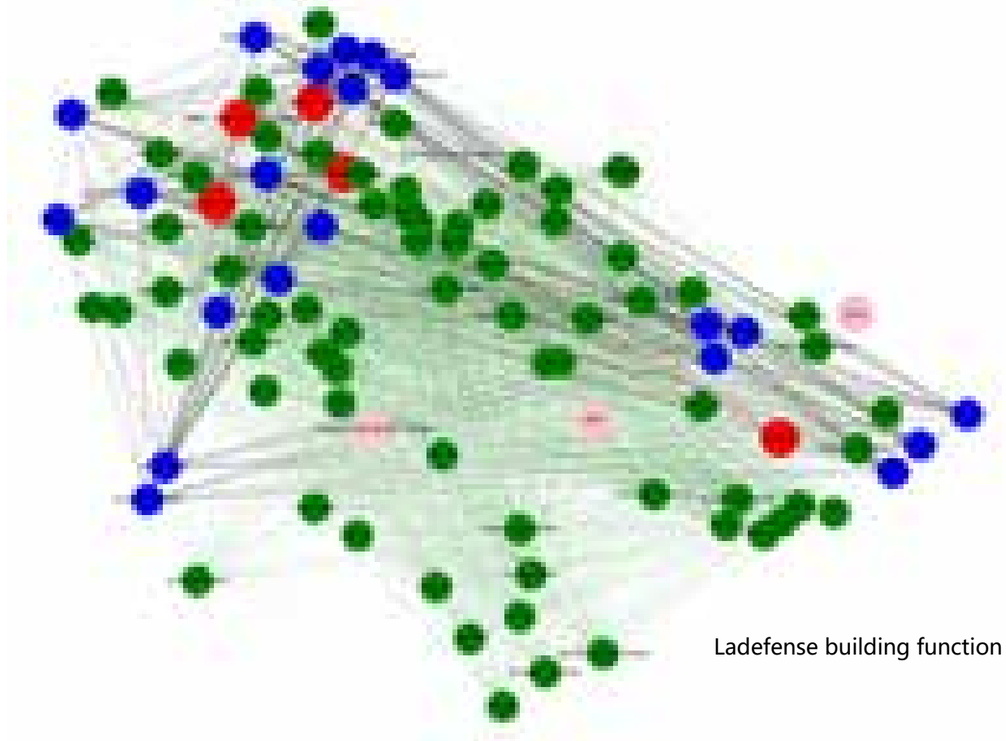

STEP4: Distinguish each node in terms of its function(or other attributes). Add edge to nodes of same function

```
for P1 in ladefensebuildings:
    for P2 in ladefensebuildings:
        if not P1 == P2:
            if P1.function == P2.function=='Hotel':
                    ladefensebuildinghotelgraph.add_edge(P1,P2)
                    ladefensebuildinghotelgraph.add_node(P1, pos = (P1.longitude, P1.latitude))

            if P1.function == P2.function=='Office':
                    ladefensebuildingofficegraph.add_edge(P1,P2)
                    ladefensebuildingofficegraph.add_node(P1, pos = (P1.longitude, P1.latitude))

            if P1.function == P2.function=='Residence':
                    ladefensebuildingresidencegraph.add_edge(P1,P2)
                    ladefensebuildingresidencegraph.add_node(P1, pos = (P1.longitude, P1.latitude))

            if P1.function == P2.function=='Shop':
                    ladefensebuildingshopgraph.add_edge(P1,P2)
                    ladefensebuildingshopgraph.add_node(P1, pos = (P1.longitude, P1.latitude))
```

STEP5: Visualize the graph and label each node

```
pos = nx.spring_layout(ladefensebuildinggraph)
node_labels = {}
for node in ladefensebuildinggraph.nodes:
    node_labels[node] = str(node.building)
```

```
optionoffice={'node_color':'green','node_size':2000, 'width':0.1, 'edge_color':'green'}
optionhotel={'node_color':'red','node_size':4000, 'width':0.8, 'edge_color':'red'}
optionresidence={'node_color':'blue','node_size':3000, 'width':0.5, 'edge_color':'black'}
optionshop={'node_color':'pink','node_size':3000, 'width':1, 'edge_color':'pink'}

plt.figure(1, figsize = (20,20))

nx.draw(ladefensebuildinghotelgraph,position, **optionhotel)
nx.draw(ladefensebuildingofficegraph,position, **optionoffice)
nx.draw(ladefensebuildingresidencegraph,position, **optionresidence)
nx.draw(ladefensebuildingshopgraph,position, **optionshop)
nx.draw_networkx_labels(ladefensebuildinggraph, position, labels=node_labels)
```



Ladefense building function

Scrape image data from flickr with python

The project focuses on building a web scraper for collecting photographs and the key elements behind such as topics, equipment used, camera settings, geographic locations, etc. The collected data will then be analyzed and visualized to give the intuition of which kinds of equipment are applicable for given topics.

```
def download_file(url, local_filename):
    if local_filename is None:
        local_filename = url.split('/')[-1]
    r = requests.get(url, stream=True)
    with open(local_filename, 'wb') as f:
        for chunk in r.iter_content(chunk_size=1024):
            if chunk:
                f.write(chunk)
    return local_filename
```

```
def get_group_id_from_url(url):
    params = {
        'method' : 'flickr.urls.lookupGroup',
        'url': url,
        'format': 'json',
        'api_key': KEY,
        'format': 'json',
        'nojsoncallback': 1
    }
    results = requests.get('https://api.flickr.com/services/rest', params=params).json()
    return results['group']['id']
```

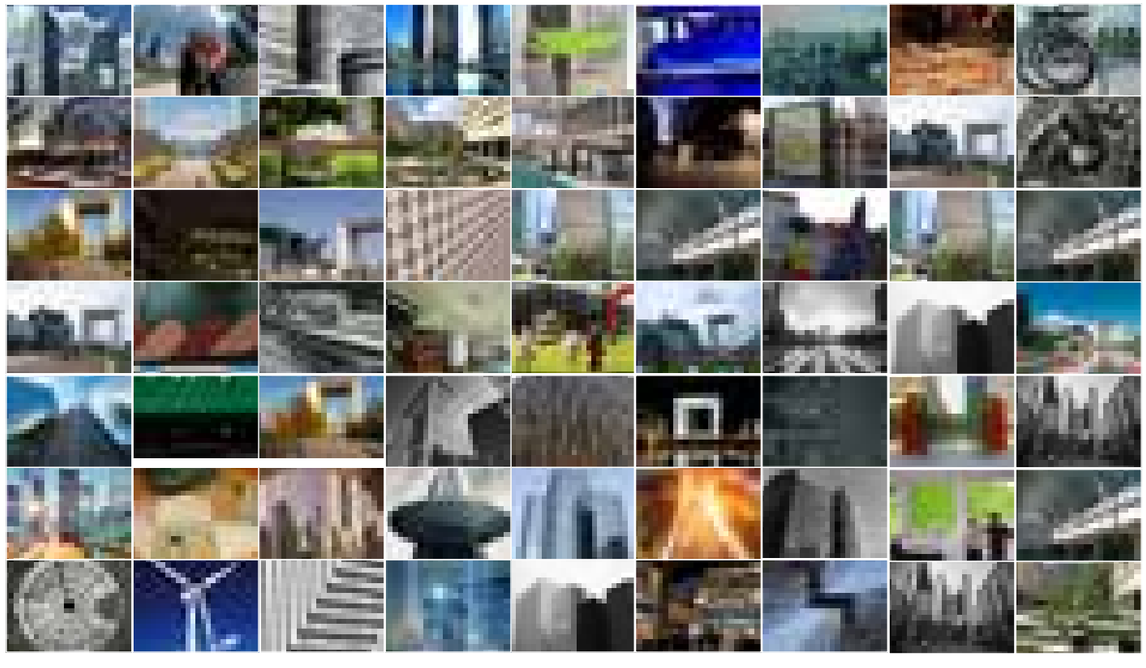
```
def get_photos(qs, qg, page=1, original=False, bbox=None):
    params = {
        'content_type': '7',
        'per_page': '500',
        'media': 'photos',
        'format': 'json',
        'advanced': 1,
        'nojsoncallback': 1,
        'extras': 'media,realname,%s,o_dims,geo,tags,machine_tags,date_taken' % ('url_o' if original else 'u
        'page': page,
        'api_key': KEY
    }

    if qs is not None:
        params['method'] = 'flickr.photos.search',
        params['text'] = qs
    elif qg is not None:
        params['method'] = 'flickr.groups.pools.getPhotos',
        params['group_id'] = qg

    # bbox should be: minimum_longitude, minimum_latitude, maximum_longitude, maximum_latitude
    if bbox is not None and len(bbox) == 4:
        params['bbox'] = ','.join(bbox)

    results = requests.get('https://api.flickr.com/services/rest', params=params).json()
    if "photos" not in results:
        print(results)
        return None
    return results["photos"]
```

```
def search(qs, qg, bbox=None, original=False, max_pages=None,start_page=1):
    # create a folder for the query if it does not exist
    foldername = os.path.join('images', re.sub(r'[\W]', '_', qs if qs is not None else "group_%s"%qg))
    if bbox is not None:
        foldername += '_'.join(bbox)

    if not os.path.exists(foldername):
        os.makedirs(foldername)

    jsonfilename = os.path.join(foldername, 'results' + str(start_page) + '.json')
```



```
    if not os.path.exists(jsonfilename):

        # save results as a json file
        photos = []
        current_page = start_page

        results = get_photos(qs, qg, page=current_page, original=original, bbox=bbox)
        if results is None:
            return

        total_pages = results['pages']
        if max_pages is not None and total_pages > start_page + max_pages:
            total_pages = start_page + max_pages

        photos += results['photo']

        while current_page < total_pages:
            print('downloading metadata, page {} of {}'.format(current_page, total_pages))
            current_page += 1
            photos += get_photos(qs, qg, page=current_page, original=original, bbox=bbox)['photo']
            time.sleep(0.5)

        with open(jsonfilename, 'w') as outfile:
            json.dump(photos, outfile)

    else:
        with open(jsonfilename, 'r') as infile:
            photos = json.load(infile)

    # download images
    print('Downloading images')
    for photo in tqdm(photos):
        try:
            url = photo.get('url_o' if original else 'url_l')
            extension = url.split('.')[-1]
            localname = os.path.join(foldername, '{}.{}'.format(photo['id'], extension))
            if not os.path.exists(localname):
                download_file(url, localname)
        except Exception as e:
            continue
```

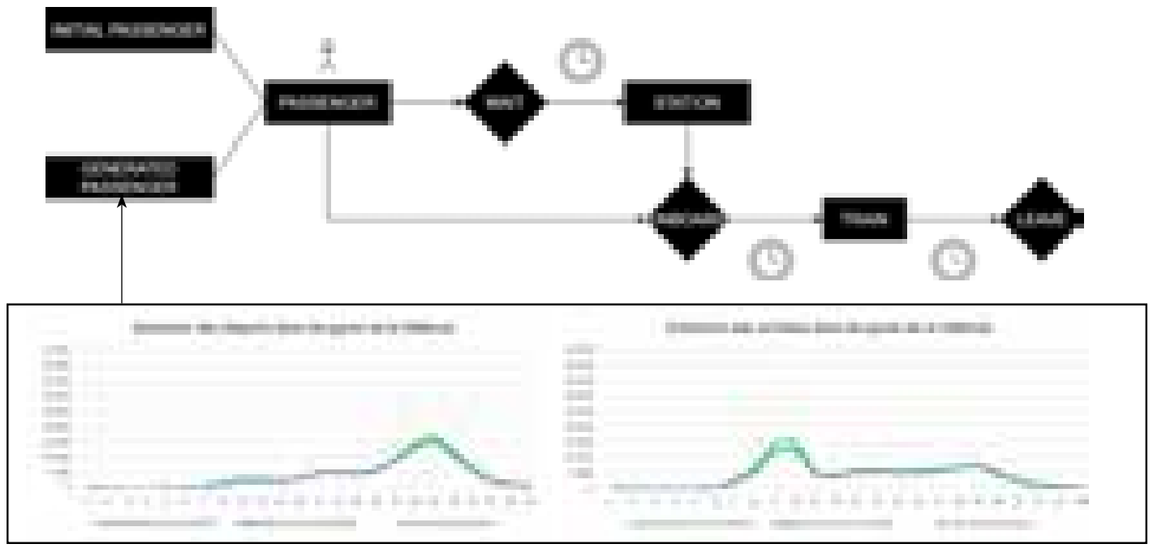
Tagged photo of Ladefense scraping from Flickr



Pedestrian simulation in Python

A simulation is an approximate imitation of the operation of a process or system that represents its operation over time.

Ladefense train station simulation Scenario:
A station system has a limited number of trains and defines a onboard processes.Passenger processes arrive at the station based on the data of the departing and arriving flow. If one train is available, they start the onboard process, the train will stay in the station for 1 minute and leave.If not, they wait until next one.The interval time between each train is 3 minutes.



STEP1: Import csv files. Define a class of ladefense train station with its various attributes including the number of people departing and arrving per hour.

```
NUM_TRAINS = 2
WAITTIME = 3
ONBOARDTIME = 1
T_INTER = 1
SIM_TIME = 1440
```

```
class Flow:
    def __init__(self,time,arrive,depart,flowall):
        self.time = time
        self.arrive = arrive
        self.depart = depart
        self.flowall = flowall
```

```
with open("flow.csv", encoding='mac_roman') as csvfile:
    reader = csv.reader(csvfile)
    rows = 0
    for r in reader:
        rows = rows + 1
```

```
Flows=[]
with open("flow.csv", encoding='mac_roman') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        hour = int(row['time'])
        arrive = row['arrive']
        depart = row['depart']
        flowall = int(row['flowall'])
        Flows.append(Flow(hour,arrive,depart,flowall))
```


STEP2: Define onboard class. A station has a limited number of trains (``NUM_NUM_TRAINS``) to carry passengers in parallel. Passengerss have to request one of the Trains. When they get on one train, they can start the onboard processes and wait 1 minute for it to leave the station (which takes ``onboardtime`` minutes).

```
class Onboard(object):
    def __init__(self, env, num_trains, waittime, onboardtime):
        self.env = env
        self.train = simpy.Resource(env, num_trains)
        self.onboardtime = onboardtime
        self.waittime = waittime
```

The waiting function. It takes a group of``passengers`` processes and tries to wait for it.

```
    def wait(self, passengers):
        yield self.env.timeout(WAITTIME)
```

The passenger function. Each passenger (has a ``name``) arrives at the station (``cw``) and requests a train. It then starts the onboard process, waits for it to finish and leaves to never come back.

```
def passenger(env, name, cw):
    hour=int(env.now/60)
    minute=env.now-hour*60

    print('%s arrives at the station at %s:%s' % (name, hour, minute))
    with open('document7.csv','a') as fd:
        fields=[name,'%s:%s' % (hour, minute)]
        writer = csv.writer(fd)
        writer.writerow(fields)
        fd.close()
    with cw.train.request() as request:
        print('%s enters the train at %s:%s' % (name, hour, minute))
        yield env.process(cw.wait(name))
        print('%s leaves the station at %s:%s' % (name, hour, minute))
```

The setup function. Create a onboard, a number of initial passengers and keep creating passengers approx. every ``t_inter`` minutes.

```
def setup(env, num_trains, waittime, t_inter, onboardtime):
    onboard = Onboard(env, num_trains, waittime, onboardtime)
```

Create 10 initial cars.

```
    for i in range(10):
        env.process(passenger(env, 'Passenger %d' % i, onboard))
```

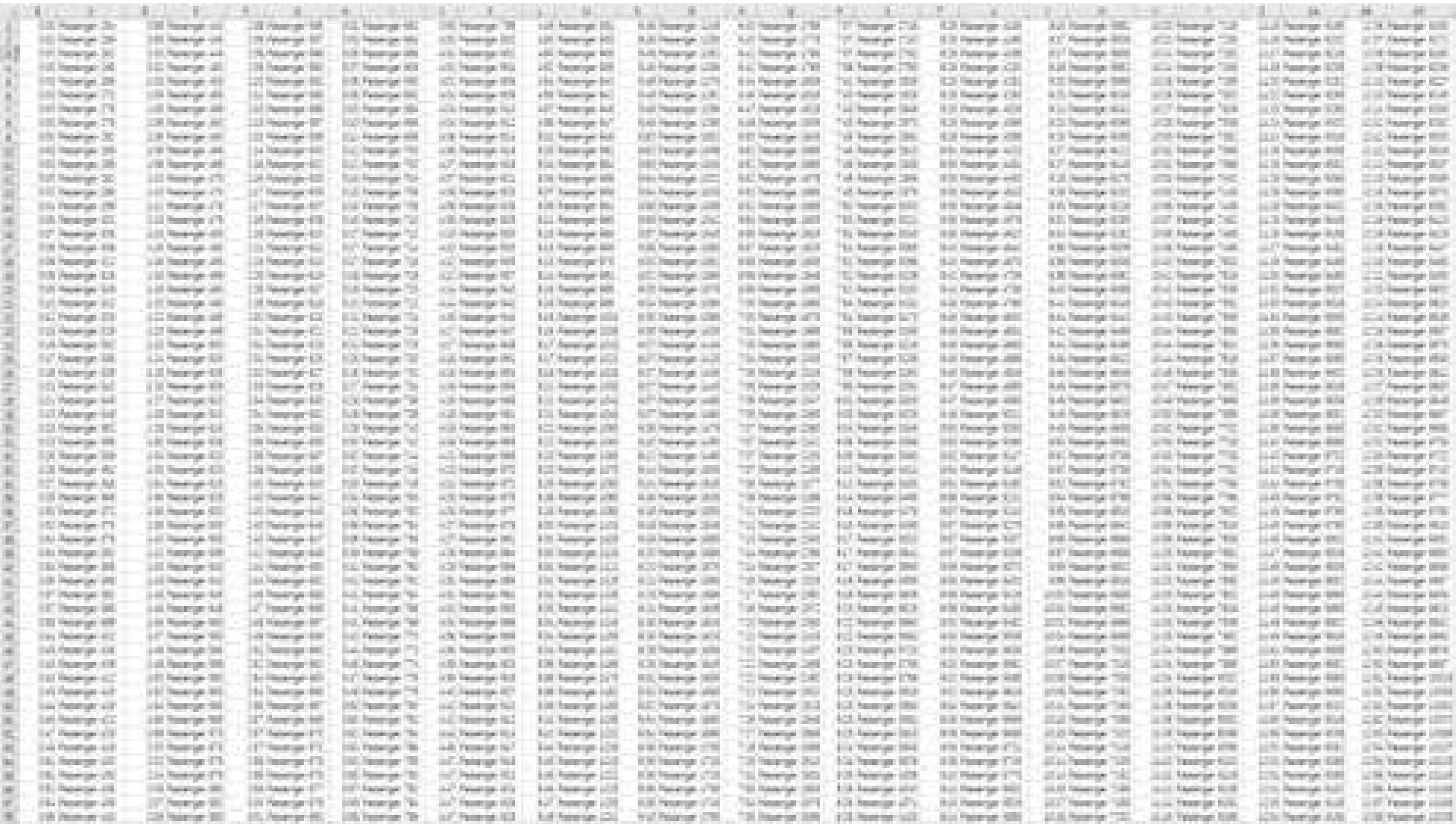
Create more passengers while the simulation is running.

```
    while True:
        yield env.timeout(random.randint(t_inter - 1 , t_inter + 1))
        i += (Flows[int(env.now/60)].flowall)/60
        env.process(passenger(env, 'Passenger %d' % i, onboard))
        print(int(env.now/60))
```

STEP3: Execution and simulation. Create an environment and start the setup process.

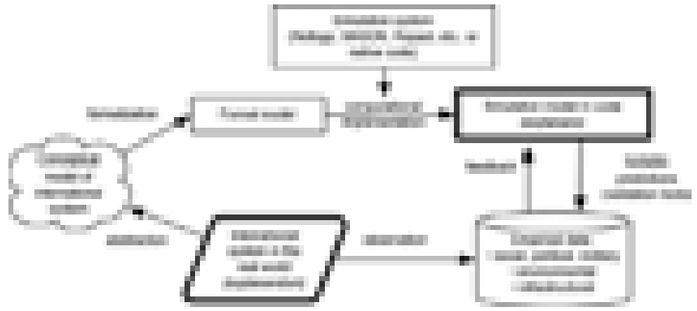
```
env = simpy.Environment()
env.process(setup(env, NUM_TRAINS, WAITTIME, T_INTER, ONBOARDTIME))
env.run(until=SIM_TIME)
```

Passenger simulation of Ladefense train station per minute



Pedestrian simulation in UnityC#

An agent-based model (ABM) is a class of computational models for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) with a view to assessing their effects on the system as a whole. It combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming.



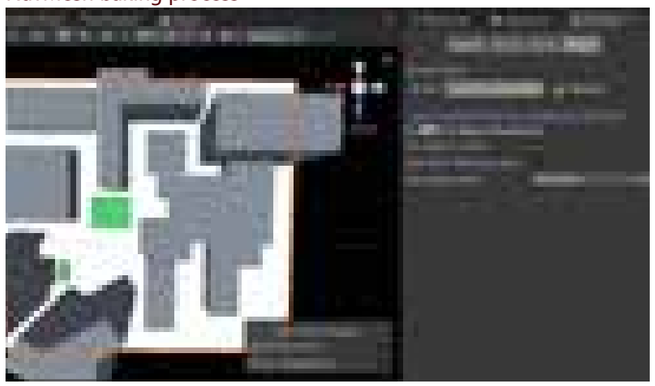
Navmesh in Unity

NavMesh (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from the level geometry.

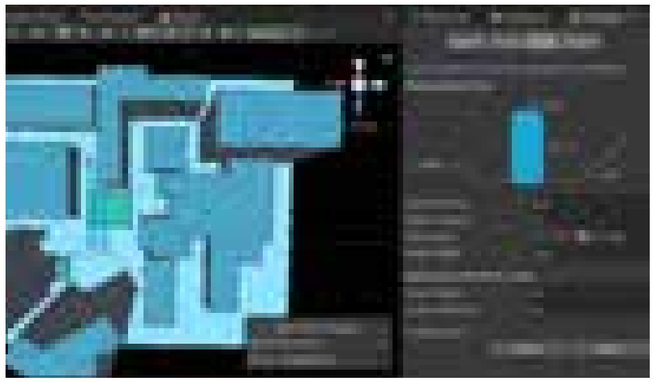
NavMesh Agent component help create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.

NavMesh Obstacle component allows to describe moving obstacles the agents should avoid while navigating the world.
While the obstacle is moving the agents do their best to avoid it, but once the obstacle becomes stationary it will carve a hole in the navmesh so that the agents can change their paths to steer around it, or if the stationary obstacle is blocking the path way, the agents can find a different route.

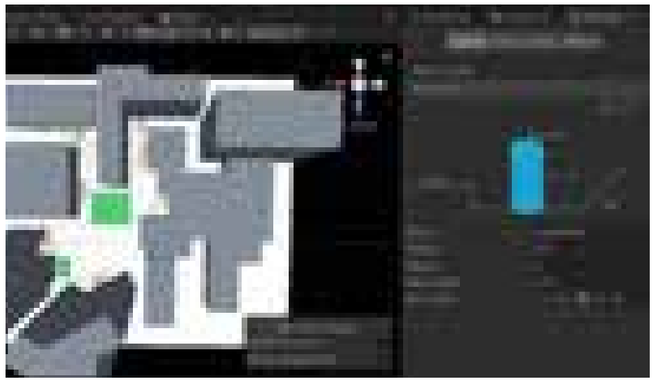
Step1: Check Navigation Static on to include selected objects in the NavMesh baking process



Step2: Adjust the bake settings and bake the navmesh.



Step3: Run!



Define the "allController" class. This class aims at defining and generating new agents.

```
5  public class allController : MonoBehaviour
```

STEP1: Assign attributes

```
7      private List<GameObject> SpawnLocationlist = new List<GameObject>();
8      private List<GameObject> AgentTypeslist = new List<GameObject>();
9      private int AgentTypesCount;
10     private int SpawnLocationType;
11     public int AgentAllCount = 60000;
```

STEP2: [redacted]
"GetAgentType" Method is called before the first frame update. Find and count all agents.

```
25     void GetAgentType()
26     {
27         GameObject parentobject = GameObject.Find("AgentTypes");
28         AgentTypesCount = parentobject.transform.childCount;
29
30         for (int i = 0; i < AgentTypesCount; i++)
31         {
32             GameObject AgentType = parentobject.transform.GetChild(i).gameObject;
33             AgentTypeslist.Add(AgentType);
34         }
35     }
```

STEP3: [redacted]
"GetSpawnLocation" Method is called before the first frame update. Find and count all the location where agents are generated.

```
36     void GetSpawnLocation()
37     {
38         GameObject parentobject = GameObject.Find("SpawnLocation");
39         SpawnLocationType = parentobject.transform.childCount;
40
41         for (int i = 0; i< SpawnLocationType; i++)
42         {
43             GameObject SpawnLocation = parentobject.transform.GetChild(i).gameObject;
44             SpawnLocationlist.Add(SpawnLocation);
45         }
46     }
```

STEP4:
"SpawnAgents" Method is called once per frame. Generate new agents until the total number of agents reach 60000("AgentAllCount")

```
47     void SpawnAgents()
48     {
49         int i = Random.Range(0, AgentTypesCount);
50         GameObject AgentSelected = AgentTypeslist[i];
51
52         int m = Random.Range(0, SpawnLocationType);
53         GameObject SpawnLocationSelected = SpawnLocationlist[m];
54
55         if (AgentTypeslist.Count<= AgentAllCount)
56         {
57             GameObject AgenttoSpawn = Instantiate(AgentSelected,
                 SpawnLocationSelected.transform.position, SpawnLocationSelected.transform.rotation);
58             AgentTypeslist.Add(AgenttoSpawn);
59         }
60     }
```

Define the "Controller" class. This class aims at defining targets and enabling agents move towards targets.

```
7  public class Controller : MonoBehaviour
```

STEP1: Assign attributes

```
9      private NavMeshAgent meshAgent;
10     private List<GameObject> alltarget = new List<GameObject>();
11     private List<GameObject> TargetTypeslist = new List<GameObject>();
12     private float distance;
13     private int rand;
14     private int TargetCount;
15     private GameObject destination;
16     private bool moveAgain = false;
```

STEP2: [redacted]

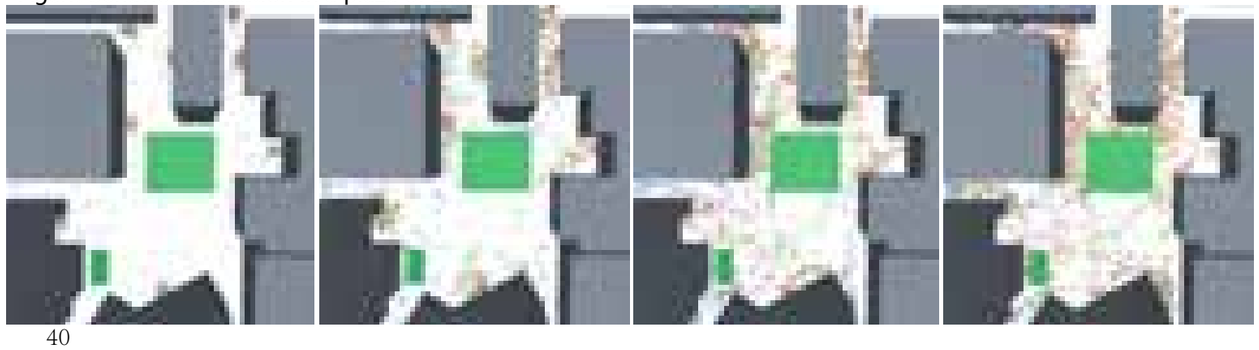
"PinkTarget" Method is called before the first frame update. Find and count all targets.

```
25     void PinkTarget()
26     {
27         GameObject parentobject = GameObject.Find("Target");
28         TargetCount = parentobject.transform.childCount;
29
30         for (int i = 0; i < TargetCount; i++)
31         {
32             GameObject TargetType = parentobject.transform.GetChild(i).gameObject;
33             TargetTypeslist.Add(TargetType);
34         }
35     }
```

STEP2: "Update" Method is called per frame. Agents move towards targets. When the distance is less than 5, agents stop moving.

```
37     void Update()
38     {
39         if (destination != null)
40         {
41             distance = Vector3.Distance(gameObject.transform.position,
                 destination.transform.position);
42         }
43         if (distance < 5)
44         {
45             moveAgain = true;
46         }
47         if(moveAgain == true)
48         {
49             rand = UnityEngine.Random.Range(0, TargetCount);
50             destination = TargetTypeslist[rand];
51             meshAgent.SetDestination(destination.transform.position);
52             Debug.Log("Rand" + rand);
53             moveAgain = false;
54         }
55     }
```

Agent behavior simulation in a plaza of Ladefense

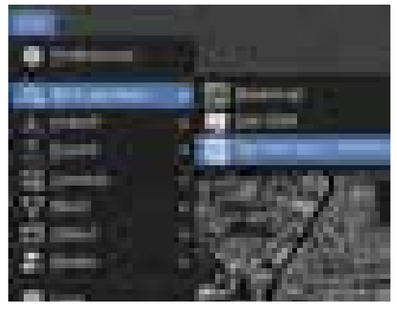


Urban-Scale Modelling based on Blender&GIS pluggin

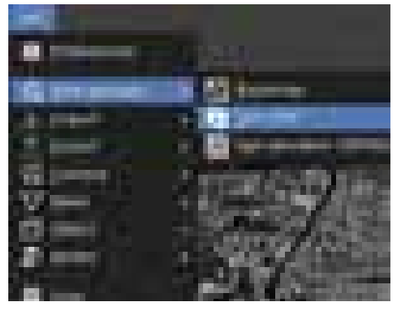
Grab geodata directly from the web : display dynamics web maps inside Blender 3d view, requests for OpenStreetMap data (buildings, roads ...), get true elevation data from the NASA SRTM mission.

STEP1:
Display dynamics web maps inside Blender 3d view by GIS pluggin

STEP2:
Choose the geographical position
Zoom the web maps according to the required scale and accuracy



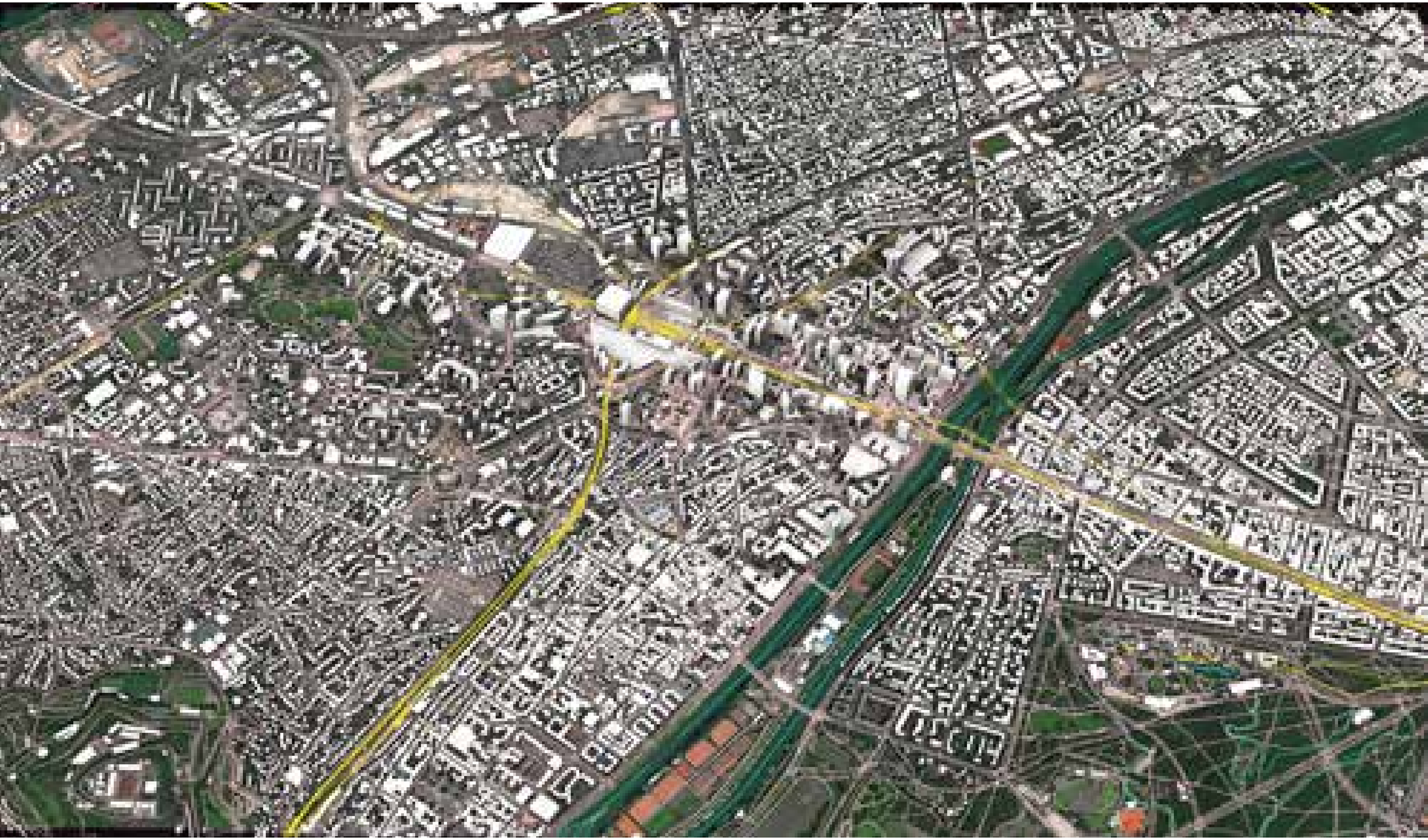
STEP3:
requests for OpenStreetMap data (buildings, roads ...), get true elevation data from the NASA SRTM mission



RELATIVE TUTORIAL: https://github.com/domlysz/BlenderGIS.git

Ladefense urban-scale model within the radius of 1.5 km

RELATIVE TUTORIAL: https://github.com/domlysz/BlenderGIS.git



Clustering analysis

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Clustering

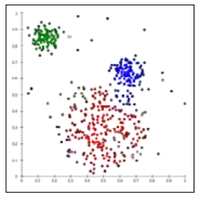
Connectivity-based clustering (hierarchical clustering)

DEFINATION
Hierarchical clustering determines cluster assignments by building a hierarchy.

STRENGTHS
-They often reveal the finer details about the relationships between data objects.
-They provide an interpretable dendrogram.

WEAKNESSES
-They' re computationally expensive with respect to algorithm complexity.
-They' re sensitive to noise and outliers.

Hierarchical clustering



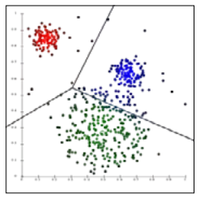
Centroid-based clustering

DEFINATION
Density-based clustering determines cluster assignments based on the density of data points in a region.

STRENGTHS
-They excel at identifying clusters of nonspherical shapes.
-They' re resistant to outliers.

WEAKNESSES
-They aren' t well suited for clustering in high-dimensional spaces.
-They have trouble identifying clusters of varying densities.

k-means algorithm



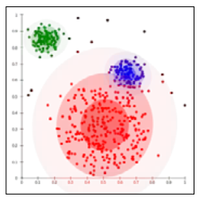
Distribution-based clustering

DEFINATION
Clusters can then easily be defined as objects belonging most likely to the same distribution.

STRENGTHS
-They produce complex models for clusters that can capture correlation and dependence between attributes.

WEAKNESSES
-They put an extra burden on the user: for many real data sets, there may be no concisely defined mathematical model

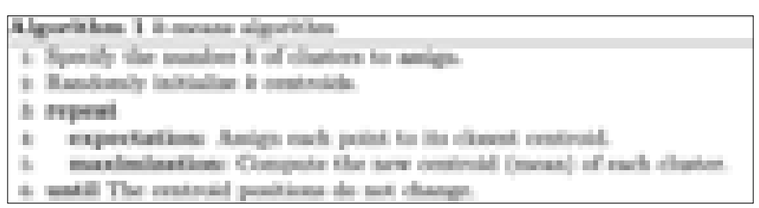
Gaussian mixture



more>>> >>>

Kmeans Clustering

k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.



The new function centroid is based on the tagged photo position of Flicker,where people meet and record most. kmeans is used to reorganize and relocate the function distribution based on the social media data.

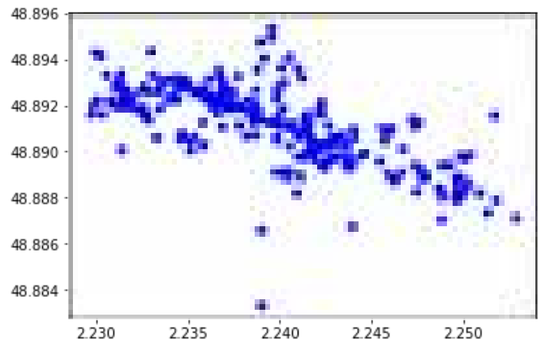
STEP1: Import csv

```
data = pd.read_csv('results1-office.csv')
data.head()
```

	farm	latitude	longitude
0	66	48.889574	2.242777
1	66	48.890086	2.231369
2	66	48.889922	2.250294
3	66	48.892301	2.238204
4	66	48.893192	2.236088

STEP2: Visualise data points

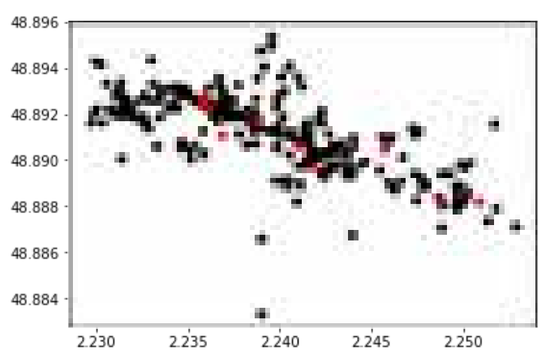
```
X = data[["longitude","latitude"]]
plt.scatter(X["longitude"],X["latitude"],c='blue')
plt.show()
```



STEP3: Specify the number of K(Next page will show how to specify K)
Randomly intialize k centroids

```
K=10
Centroids = (X.sample(n=K))
plt.scatter(X["longitude"],X["latitude"],c='black')
plt.scatter(Centroids["longitude"],Centroids["latitude"],c='red')
```

```
<matplotlib.collections.PathCollection at 0x287c2152fd0>
```



STEP4: Assign all the points to the closest cluster centroid

```
diff = 1
j=0

while(diff!=0):
    XD=X
    i=1
    for index1,row_c in Centroids.iterrows():
        ED=[]
        for index2,row_d in XD.iterrows():
            d1=(row_c["longitude"]-row_d["longitude"])**2
            d2=(row_c["latitude"]-row_d["latitude"])**2
            d=np.sqrt(d1+d2)
            ED.append(d)
        X[i]=ED
        i=i+1
```

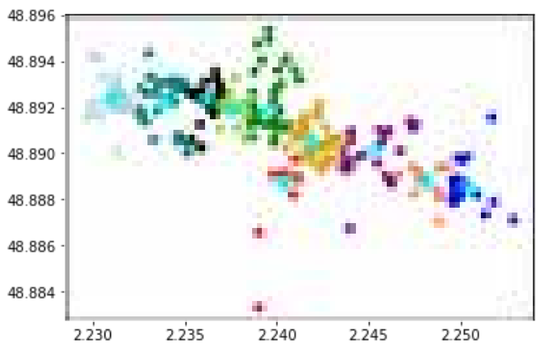
STEP5: Recompute centroids of newly formed clusters

```
    for index,row in X.iterrows():
        min_dist=row[1]
        pos=1
        for i in range(K):
            if row[i+1] < min_dist:
                min_dist = row[i+1]
                pos=i+1
        C.append(pos)
    X["Cluster"]=C
    Centroids_new = X.groupby(["Cluster"]).mean()[["latitude","longitude"]]
    if j == 0:
        diff=1
        j=j+1
    else:
        diff = (Centroids_new['latitude'] - Centroids['latitude']).sum() + (Centroids_new['longitude'] - Cen
    Centroids = X.groupby(["Cluster"]).mean()[["latitude","longitude"]]
```

STEP6: Final clustering and centroids

```
color=['purple','green','red','teal','black','yellowgreen','blue','coral','goldenrod','lightblue']
for k in range(K):
    data=X[X["Cluster"]==k+1]
    plt.scatter(data["longitude"],data["latitude"],c=color[k])
plt.scatter(Centroids["longitude"],Centroids["latitude"],c='cyan',marker='*',s=200)
plt.figure(figsize=(6, 15))
print(Centroids)
```

Cluster	latitude	longitude
1	48.890234	2.245306
2	48.891927	2.239340
3	48.888823	2.240342
4	48.892202	2.233941
5	48.892394	2.236183
6	48.891960	2.237616
7	48.888584	2.250480
8	48.888855	2.248086
9	48.890523	2.241904
10	48.892442	2.231031



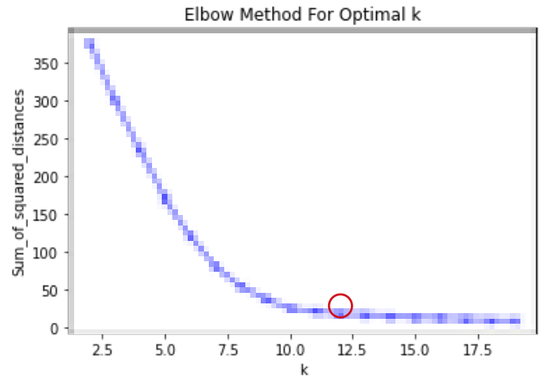
Elbow Method For Optimal k

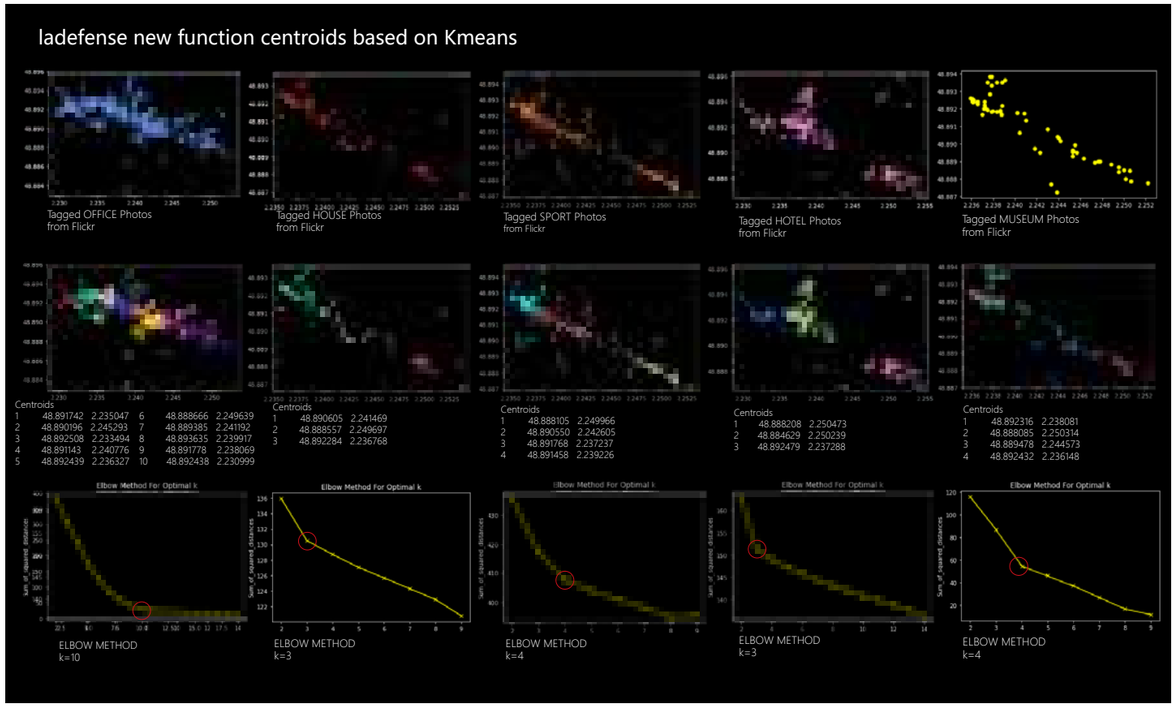
Elbow method is a heuristic used in determining the number of clusters in a data set. The method consists of plotting the explained variation as a function of the number of clusters, and picking the elbow of the curve as the number of clusters to use.

```
mms = MinMaxScaler()
mms.fit(data)
data_transformed = mms.transform(data)
```

```
Sum_of_squared_distances = []
K = range(2,20)
for k in K:
    km = KMeans(n_clusters=k)
    km = km.fit(data_transformed)
    Sum_of_squared_distances.append(km.inertia_)
```

```
plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k')
plt.show()
```





3D-Kmeans

STEP1: Import csv

```
data = pd.read_csv('cluster1-pop.csv')
```

```
X = data[["xcor","ycor","zcor"]]
Y = data[["xcor","ycor"]]
x=np.array(X["xcor"],dtype=float)
y=np.array(X["ycor"],dtype=float)
z=np.array(X["zcor"],dtype=float)
```

STEP2: 3D kmeans implementation

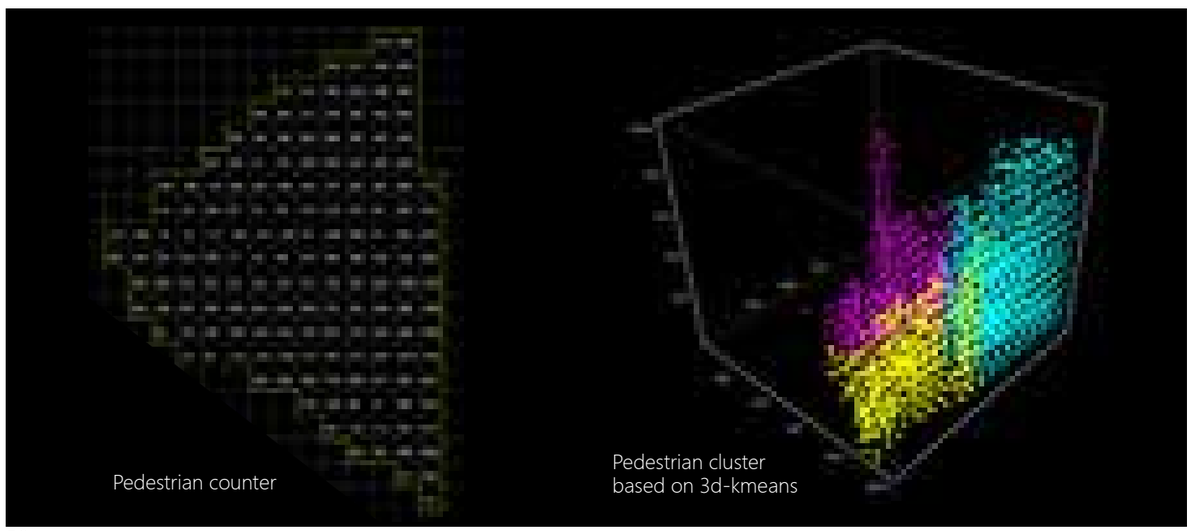
```
fig = ipv.figure(height=600, width=600, layout={'width':'50%', 'height':'50%'})
scatter = ipv.scatter(x,y,z, size=2, marker="sphere",color="red",opacity=0.03, selection=None)
```

```
def hex_to_rgb(hex):
    hex = hex[1:]
    return struct.unpack('BBB', bytes.fromhex(hex))
```

```
def handle_cp_change(labels, **groups):
    group_ids = [int(g.split(' ')[1]) for g in groups.keys()]
    group_color = {k: hex_to_rgb(get_cp_value(cp)) for k, cp in zip(group_ids, groups.values())}
    colors = list(map(lambda x: group_color[x], labels))
    scatter.color = colors

def get_cp_value(cp):
    if type(cp) == ColorPicker:
        return cp.value
    else:
        return cp
```

```
avaliable_colors = {
    0: '#ff0000',
    1: '#00ff00',
    2: '#0000ff',
}
```

```
def color_scatter_with_kmeans(n_clusters):
    kmeans = KMeans(n_clusters=n_clusters)
    kmeans = kmeans.fit(Y)
    labels = kmeans.predict(Y)
    color_pickers = {f'group {k}': ColorPicker(value=avaliable_colors[k%len(avaliable_colors)], description=
                     for k in range(n_clusters)}
    handle_cp_change(labels=list(labels), **color_pickers)
    return interact(handle_cp_change, labels=fixed(list(labels)) , **color_pickers)
```



Schelling Segregation Model

The Schelling model of segregation is an agent-based model that illustrates how individual tendencies regarding neighbors can lead to segregation. The model is especially useful for the study of residential segregation of ethnic groups where agents represent householders who relocate in the city.

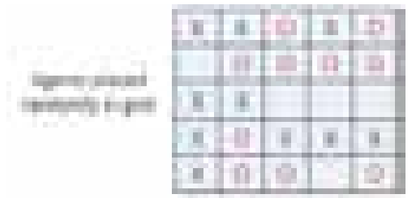
In the model, each agent belongs to one of two groups and aims to reside within a neighborhood where the fraction of 'friends' is sufficiently high: above a predefined tolerance threshold value F. It is known that depending on F, for groups of equal size, Schelling's residential pattern converges to either complete integration (a random-like pattern) or segregation.

The New York City map represents data from the 2010 US Census color-coded by race.

Theory of model

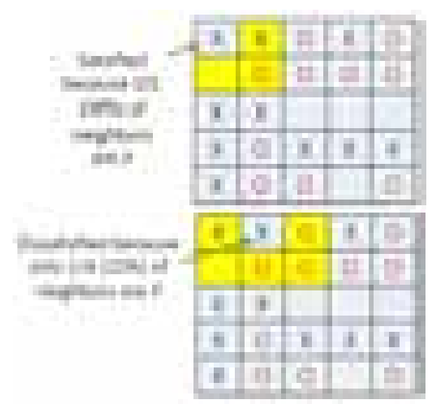
Step1

Suppose there are two types of agents: X and O. The two types of agents might represent different races, ethnicity, economic status, etc. Two populations of the two agent types are initially placed into random locations of a neighborhood represented by a grid. After placing all the agents in the grid, each cell is either occupied by an agent or is empty.



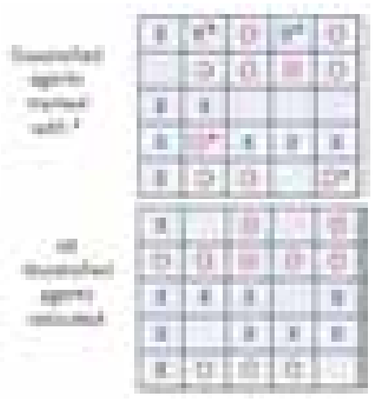
Step2

An important step is to define if each agent is satisfied with its current location.
A satisfied agent is one that is surrounded by at least t percent of agents that are like itself. This threshold t is one that will apply to all agents in the model, even though in reality everyone might have a different threshold they are satisfied with.
Note that the higher the threshold, the higher the likelihood the agents will not be satisfied with their current location.



Step3

When an agent is not satisfied, it can be moved to any vacant location in the grid. Any algorithm can be used to choose this new location. For example, a randomly selected cell may be chosen, or the agent could move to the nearest available location.



STEP1: Define the agent class with various parameters(kind, preference of the neighbours and the world environment). Define its neighbours, the preferred neighbours, its happiness and dissatisficory towards its neighbours(The more similar neighbours the agent have, the happier it is) and the vacany.

```
class Agent:
    def __init__(self, x, y, kind, preference, world):
        self.x = x
        self.y = y
        self.kind = kind
        self.preference = preference
        self.world = world

    def know_neighbours(self):
        self.neighbours = [self.world.coords[x % self.world.size, y % self.world.size]
                           for x, y in itertools.product(range(self.x-1, self.x+2),
                                                         range(self.y-1, self.y+2))]

    def swap(self, partner):
        self.kind, partner.kind = partner.kind, self.kind

    def happiness(self):
        neighbour_kinds = [n.kind for n in self.neighbours]
        numsame = neighbour_kinds.count(self.kind)
        numvacant = neighbour_kinds.count(0)
        if numvacant == 8:
            if self.vacant():
                return 1.0
            return 0.0
        return (numsame - 1) / (8 - numvacant)

    def dissatisfied(self):
        return self.happiness() < self.preference

    def vacant(self):
        return self.kind == 0
```

STEP2: Define the world class with various parameters. A grid of squares represents the world, with each square representing a plot of land in which an agent can live.The world is populated (randomly initially) with a number of different kinds of agents, leaving a proportion of the squares vacant.

```
class World:
    def __init__(self, size, kinds, probs, preference):
        self.size = size
        self.preference = preference
        self.coords = self.populate_world(kinds, probs, preference)
        [a.know_neighbours() for row in self.coords for a in row]
        self.num_dissatisfied = size ** 2
        self.num_vacant = sum([a.vacant() for row in self.coords for a in row])

    def populate_world(self, kinds, probs, preference):
        return np.array([[Agent(x, y, np.random.choice(kinds, p=probs), preference, self) for y in range(se
```

```
    def advance_turn(self):
        dissatisfied = []
        vacant = []
        for row in self.coords:
            for a in row:
                if a.vacant():
                    vacant.append(a)
                elif a.dissatisfied():
                    dissatisfied.append(a)
        self.num_dissatisfied = len(dissatisfied)
        np.random.shuffle(dissatisfied)
        np.random.shuffle(vacant)
        num_swap = min(self.num_dissatisfied, self.num_vacant)
        for indx in range(num_swap):
            dissatisfied[indx].swap(vacant[indx])
```


STEP3: Another measure of segregation is the number of ‘colonies’ formed. That is how many clusters of similar agents are formed. For the purpose of illustration, I will very loosely define a colony as follows: Imagine the grid of squares as a network of vertices, where there is an edge between two vertices if those squares are neighbours and they are occupied by agents of the same kind. Then a colony is a connected component of that network.

```
def number_connected_components(w):
    temp = np.matrix(copy.deepcopy(w.atlas()))
    nodes = ['(' + str(x) + ', ' + str(y) + ')' for x in range(w.size) for y in range(w.size) if temp[x, y]
    G = nx.Graph()
    G.add_nodes_from(nodes)
    for x in range(w.size):
        for y in range(w.size):
            if temp[(x+1) % w.size, y] != 0 and temp[x, y] != 0:
                if temp[(x+1) % w.size, y] == temp[x, y]:
                    G.add_edge('(' + str(x) + ', ' + str(y) + ')', '(' + str((x+1) % w.size) + ', ' + str(y)
            if temp[x, (y+1) % w.size] != 0 and temp[x, y] != 0:
                if temp[x, (y+1) % w.size] == temp[x, y]:
                    G.add_edge('(' + str(x) + ', ' + str(y) + ')', '(' + str(x) + ', ' + str((y+1) % w.size)
    return nx.number_connected_components(G)
```

STEP4: Perference experiment. Set the proportion of each agent and the vacancy.

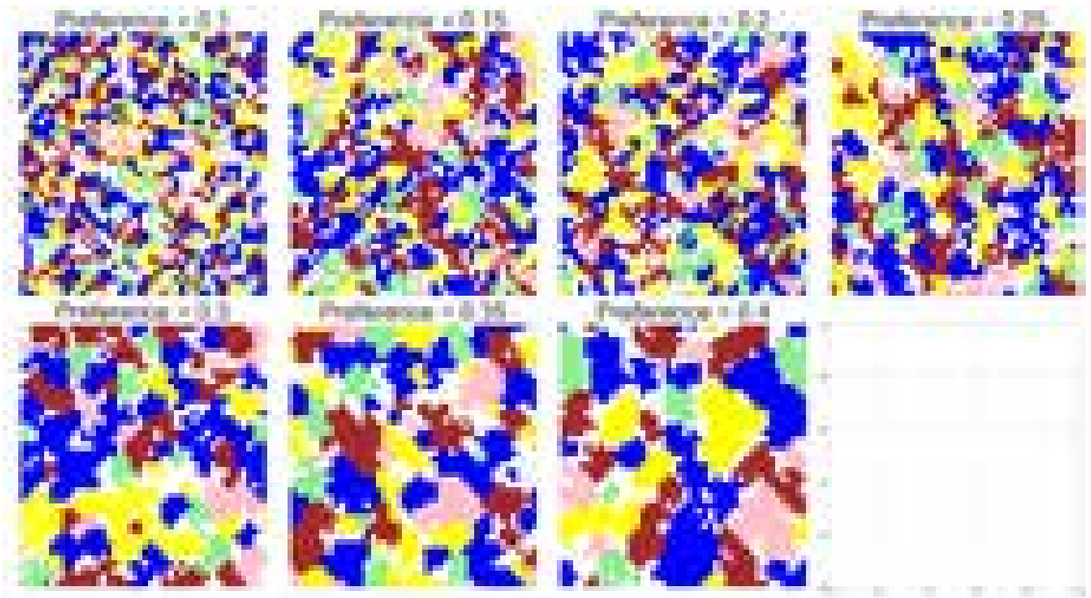
```
np.num_trials = 1
worlds = {str(round(pref, 4)): [World(40, [0, 1, 2, 3, 4, 5], [0.20, 0.15, 0.30, 0.10, 0.15, 0.10]
```

```
for k, trial in tqdm.tqdm(list(itertools.product(worlds, range(num_trials)))):
    worlds[k][trial].play(threshold=0.00005)
```

```
100%|████████████████████████████████████████████████████████████████████████████████| 43/43 [00:12<00:00,  3.53it/s]
```

```
nrows = 3
ncols = 4
cmap = colors.ListedColormap(['white', 'yellow', 'blue','pink','brown','lightgreen']) fig, axarr =
plt.subplots(nrows=nrows, ncols=ncols, figsize=(22, 18))
for i, k in enumerate(['0.1', '0.15','0.2', '0.25', '0.3', '0.35', '0.4']):
    axarr[i // ncols, i % ncols].pcolor(worlds[k][0].atlas(), cmap=cmap)
    axarr[i // ncols, i % ncols].set_xticks([])
    axarr[i // ncols, i % ncols].set_yticks([])
    axarr[i // ncols, i % ncols].set_title('Preference = ' + str(k), fontsize=34)
```

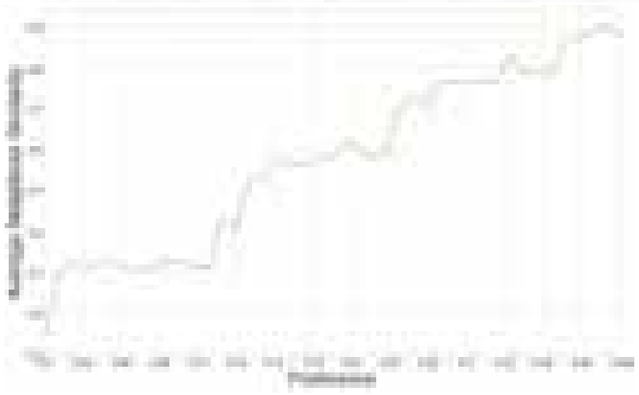
```
plt.tight_layout()
fig.savefig('increase_preference')
```



STEP5: As the individual agents' preferences are increased, the world becomes more and more segregated. We can see in the plot below that as agents' preference is increased, the resulting world's mean happiness also increases.

```
fig = plt.figure(figsize=(16, 10))
sns.tsplot([[np.mean(worlds[k][trial].happiness_distribution()) for k in sorted(worlds)] for trial in range
ticks = np.linspace(0, 42, 16)

plt.xticks(ticks, [list(sorted(worlds.keys()))[int(t)] for t in ticks])
plt.xlabel('Preference', fontsize=28)
plt.ylabel('Average Neighbour Similarity', fontsize=28)
plt.tick_params(axis='both', which='major', labelsize=16)
plt.savefig('preference_meanhappiness')
```

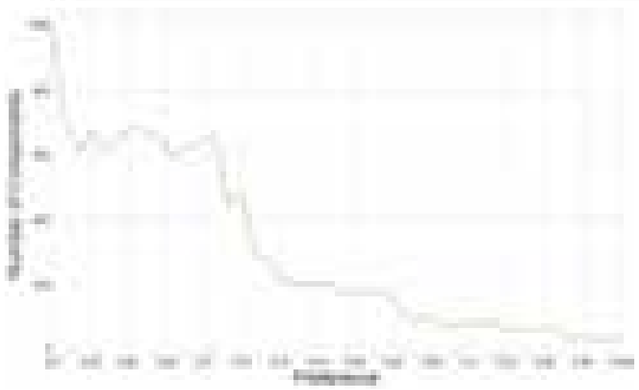
```
C:\Users\claire\miniconda3\lib\site-packages\seaborn\timeseries.py:183: UserWarning: The `tsplot` function is
deprecated and will be removed in a future release. Please update your code to use the new `lineplot` functio
n.
  warnings.warn(msg, UserWarning)
```

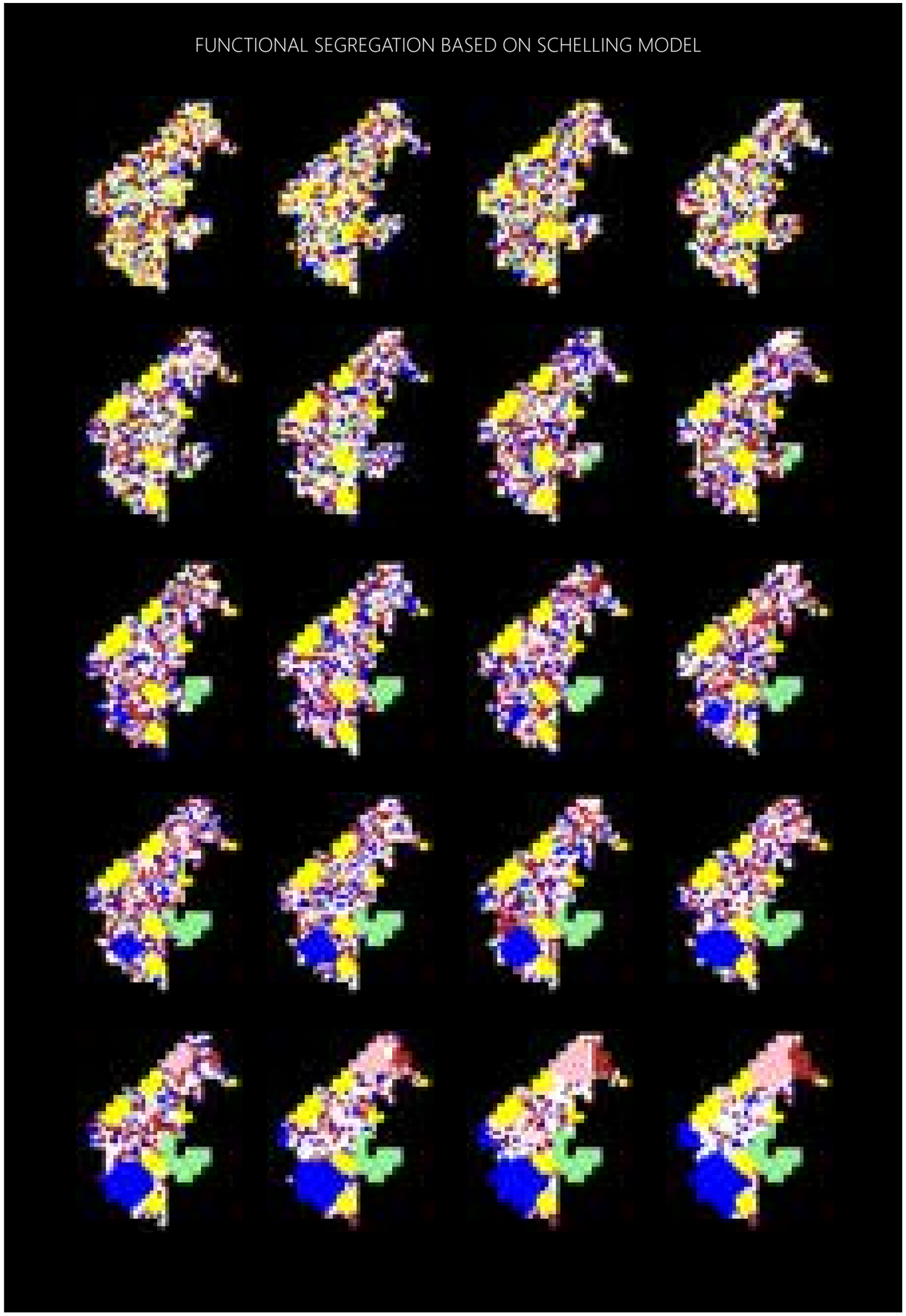


STEP6: Using NetworkX getting the number of connected components is simple. The plot below shows that the number of colonies dramatically decrease as agents' preference increases, and that the largest decrease actually occurs at a very low preference level:

```
fig = plt.figure(figsize=(16, 10))
sns.tsplot(np.transpose(components), ci=99)
ticks = np.linspace(0, 42, 16)
plt.xticks(ticks, [list(sorted(worlds.keys()))[int(t)] for t in ticks])
plt.xlabel('Preference', fontsize=28)
plt.ylabel('Number of Components', fontsize=28)
plt.tick_params(axis='both', which='major', labelsize=16)
plt.savefig('preference_components')
```

```
C:\Users\claire\miniconda3\lib\site-packages\seaborn\timeseries.py:183: UserWarning: The `tsplot` function is
deprecated and will be removed in a future release. Please update your code to use the new `lineplot` functio
n.
  warnings.warn(msg, UserWarning)
```





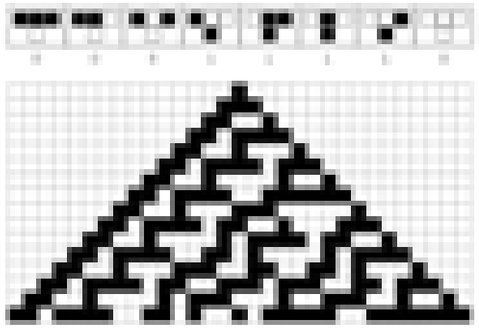
Cellular Automaton in MMA

A cellular automaton is a collection of "colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired. von Neumann was one of the first people to consider such a model, and incorporated a cellular model into his "universal constructor."

Elementary cellular automaton

The simplest nontrivial cellular automaton would be one-dimensional, with two possible states per cell, and a cell's neighbors defined as the adjacent cells on either side of it.

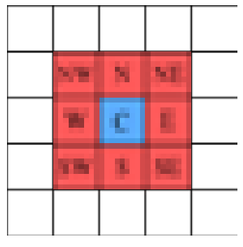
A cell and its two neighbors form a neighborhood of 3 cells, so there are 23 = 8 possible patterns for a neighborhood. A rule consists of deciding, for each pattern, whether the cell will be a 1 or a 0 in the next generation. There are then 28 = 256 possible rules.



Conway's Game of Life

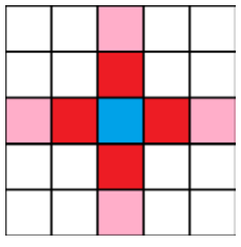
The array of cells of the automaton has two dimensions.
Each cell of the automaton has two states (conventionally referred to as "alive" and "dead", or alternatively "on" and "off")
The neighborhood of each cell is the Moore neighborhood; it consists of the eight adjacent cells to the one under consideration and (possibly) the cell itself.
In each time step of the automaton, the new state of a cell can be expressed as a function of the number of adjacent cells that are in the alive state and of the cell's own state; that is, the rule is outer totalistic (sometimes called semitotalistic).

Moore neighborhood



It is one of the two most commonly used neighborhood types, the other one being the von Neumann neighborhood. The well known Conway's Game of Life uses the Moore neighborhood. It is similar to the notion of 8-connected pixels in computer graphics.

Von Neumann neighborhood



In cellular automata, the von Neumann neighborhood (or 4-neighborhood) is classically defined on a two-dimensional square lattice and is composed of a central cell and its four adjacent cells.

Unit Aggregation Experiment

```
In[ ]:= Manipulate[
          If[bs === Sphere, o = 0];
```

evolution rule

```
With[{u = Part[
    CellularAutomaton[{rn, {2, {{2, 2, 2}, {2, 1, 2}, {1, 2, 2}}}, {1, 1}},
 {{Table[1, {init}]}, 0}, {t, All, All}], 1 + Accumulate[IntegerDigits[c, 2, t]]]},
```

generation rules:Game of Life

initial condition

generation steps

Set parameters

```
{{rn, 121268, "rule number"}, {121268, 124844, 124868}, SetterBar},
 {{c, 1, "choice number"}, 1, 2^20, 1},
 {{init, 5, "size of initial block"}, 2, 25, 1,
  Appearance -> "Labeled"},
 {{t, 6, "steps of evolution"}, 1, 15, 1,
  Appearance -> "Labeled"}, Delimiter,
 {{o, 0, "transparency"}, 0, 1,
  Enabled -> (bs === Cuboid)}, {{se, True, "show edges"}, {True, False}},
ControlPlacement -> Top]
```

Set graphics

```
With[{
   g = Graphics3D[{Opacity[1 - o],
      bs[{#2, #3, -#1}] & @@@ Position[u, 1]}, ImageSize -> 200]},
  If[Grid[{{g, Column[ArrayPlot[Sqrt[Total[#]], ImageSize -> 120] & /@ {u,
        Transpose[u], Transpose[u, {2, 3, 1}]}]}}], g]]],
```