NIHR | Applied Research Collaboration South West Peninsula

@penCHORD_UoE
@peninsula_ARC

Module 6 : Modelling Behaviour
Session 6B : Agent Based Simulation using MESA
Dr Daniel Chalk

"Little Computer People"

HSMA 5

#hsma5isalive

An Introduction to Agent Based Simulation and MESA

Dr Daniel Chalk

What is Agent Based Simulation?

- A simulation method that focuses on the behaviours, interactions and motivations of individuals, and observes the population-level **emergent behaviour** that arises from these complex interactions.
- Commonly used in ecology, where often much is known about individual forager behaviours (typically known as Individual Based Modelling here)
- Extensively used in modelling disease spread / outbreaks in public health
- Different to ‘conventional’ mathematics

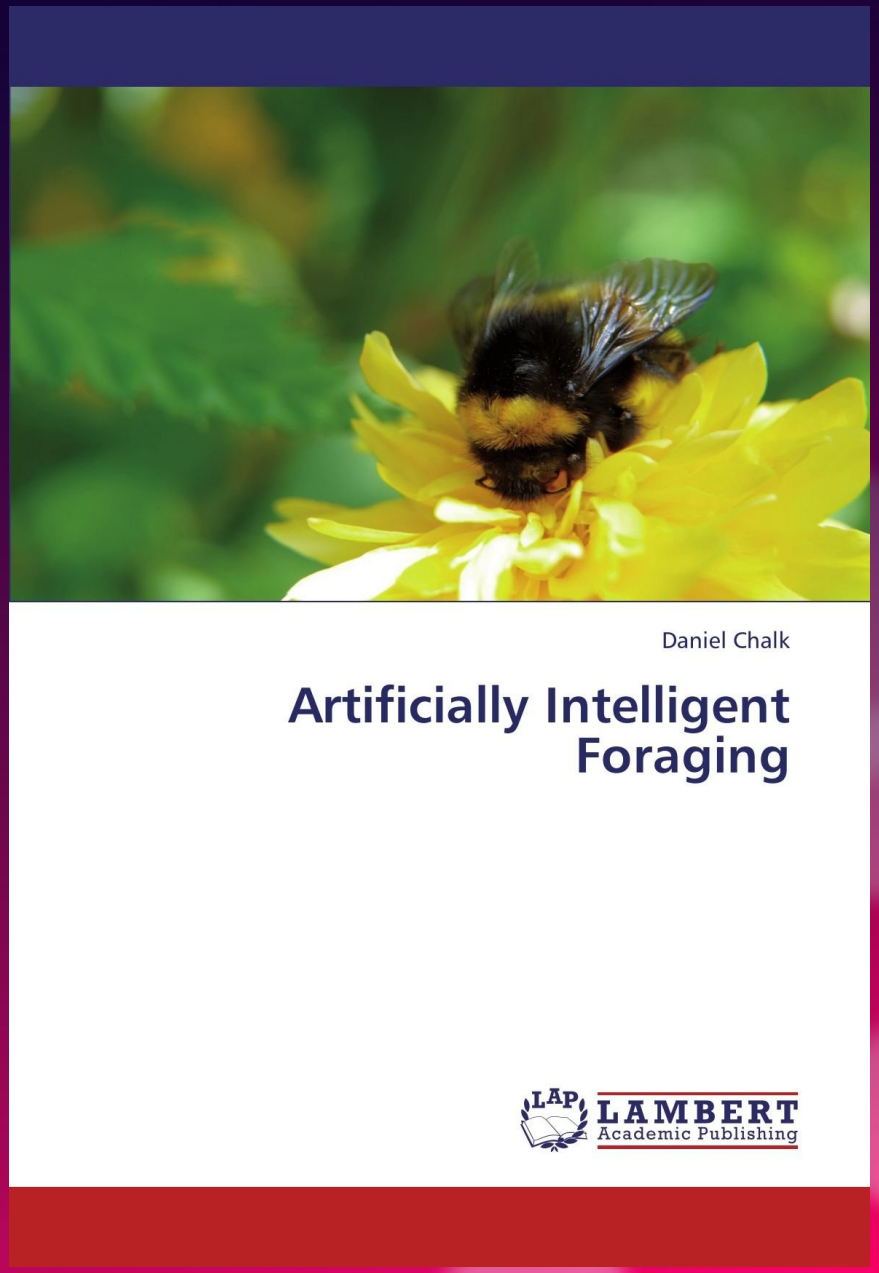
Elements of an Agent Based Simulation

- **Agents** that move and interact within an **environment**
- **Individual-level** behaviours and interactions
- **Emergent** behaviour = the outcome

Conway’s Game of Life

- Most famous example of a **Cellular Automaton** (a type of Agent Based Model)
- A grid of cells that are each alive or dead
- Simple rules :
  - Any live cell with 0 or 1 neighbours dies from solitude / starvation
  - Any live cell with 2 or 3 neighbours survives
  - Any live cell with 4 or more neighbours dies from overpopulation
  - Any dead cell with three neighbours becomes alive (birth)

An ecological Agent Based Simulation from an “esteemed” academic



Just £69 on Amazon!

HARVEST (Harvesting Animals with Reinforced Values and ESTimates)

Field A

30% full
Estimate = 50%

Field B

100% full
Estimate = 70%

Field C

10% full
Estimate = 40%

GOAL :
To fill up with nectar (food) as quickly as possible

WHICH FIELD?
What do I think is the probability of me finding a full flower here / elsewhere?
What's the movement cost of moving elsewhere?

MESA

- Python-based framework for Agent Based Simulation
- Free and Open Source (FOSS)
- Wonderful visualisation capabilities
- Quick and easy to put together models

To install MESA :

pip install mesa

MESA Model Structure

Model Module

- Class for each type of agent, consisting of :
  - relevant agent attributes
  - constructor method
  - methods defining things the agent can do (movement, other actions)
  - 'step' method that defines which actions will be taken in what order at each time step of the model

- Class for the Model, consisting of :
  - relevant model attributes
  - constructor method
  - 'step' method that defines what needs to happen model-wide at each time step

- Functions to calculate results

Run Module

Runs the model, either as a single run, or as a batch, and with or without live visualisation of the model

Server Module (optional)

Launches a server to send data to Javascript to draw the model live in a web browser

Each module lives in a separate .py file, and we import the modules and bits of modules we need into each

Disease Spread Model

Imagine we have been asked to model the spread of a new disease.

The disease is spread through close contact with infected individuals. The probability of someone catching the disease after coming into contact with an infected individual (transmissibility) is 20%. The disease lasts, on average, for 10 days.

We're going to build a model of the spread of this disease within a population.

The Model Module – disease_model.py

First we need to import the libraries that we'll need to build our model

```
from mesa import Agent, Model
from mesa.time import RandomActivation # random order of agent actions
from mesa.space import MultiGrid # multiple agents per cell
from mesa.datacollection import DataCollector

import random
```

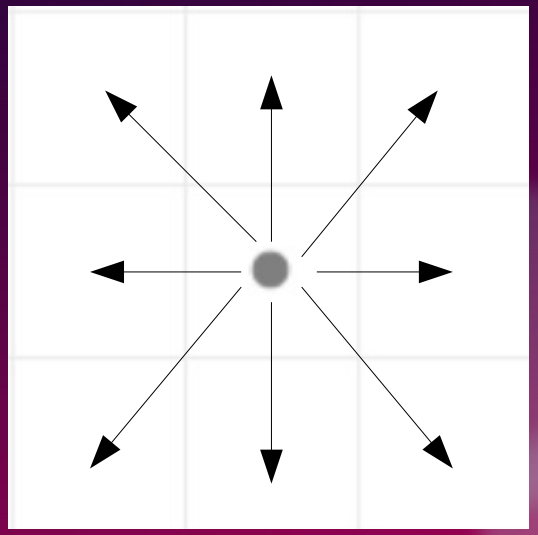
Writing the Agent Class

- For this model, we'll create a single agent class, representing a "person"
- In our Agent class, we will have multiple functions that define what an agent does. This includes a function to move the agent, a function to infect others, and a function to define all the things to do at each "tick of the clock" in the model.
- But the first function we need to define is our constructor __init__
- In the constructor, we'll specify the transmissibility, the level of movement of the agent, the average disease duration, and whether or not the agent is infected at the start of the model, according to a probability we specify.

```
# A class representing a 'person' agent.  Note we're passing in the Agent class
# we imported from the mesa library.  Remember that this means our class here
# is inheriting from the 'parent' Agent class, and our class is the 'child',
# which inherits all the attributes and methods of the parent, but may have
# some of its own.
class Person_Agent(Agent):
    # Constructor
    def __init__(self, unique_id, model, initial_infection, transmissibility,
                 level_of_movement, mean_length_of_disease):
        # Call the constructor from the parent Agent class, which will do all
        # the hard work of defining what an agent is - we just give it an ID
        # and a model that it will live in
        super().__init__(unique_id, model)

        # Now we define the attributes of our Person Agent that aren't in the
        # parent class
        # First, we specify the level of transmissibility (the probability of
        # passing the disease onto someone else after contact with them,
        # assuming this agent is infected, and the other one isn't)
        self.transmissibility = transmissibility

        # The probability that the agent will move from its current location
        # at any given time step
        self.level_of_movement = level_of_movement

        # Average duration of being infected with the disease
        self.mean_length_of_disease = mean_length_of_disease

        # We're going to set up our model so that some agents are already
        # infected at the start.  We've got a parameter value passed in
        # (initial_infection) that defines the probability of any given agent
        # being infected at the start.  So, we just randomly sample from a
        # uniform distribution between 0 and 1, and if the sampled value is
        # less than this probability, then we say that the agent is infected -
        # we set their Boolean infected attribute to True, and randomly
        # sample a duration for which they'll have the disease, based on the
        # mean duration we passed in.  Otherwise, we set their infected
        # attribute to false
        if random.uniform(0, 1) < initial_infection:
            self.infected = True
            self.disease_duration = int(round(
                    random.expovariate(1.0 / self.mean_length_of_disease), 0))
        else:
            self.infected = False
```

Movement

- We need to write a movement function in the Person Agent class to specify how our agents will move around the grid of cells in our environment.
- In this model, agents will move to a random neighbouring cell (including diagonals – a “Moore” Neighbourhood). More than one agent can be in any one cell.
- We first work out what the possible movement locations are, based on the cells around the agent, then pick one at random, then tell the model to move the agent to that cell.



Movement

```
# Agent movement method - this is called if it is determined the agent
# is going to move on this time step
def move(self):
    # Get a list of possible neighbouring cells to which to move
    # We use the get_neighborhood function, giving it the agent's current
    # position on the grid, stating we want a Moore neighbourhood (which
    # includes diagonals), and that we don't want to include the centre
    # (where the agent is currently) in the returned neighbourhood list
    possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False)

    # Select new position at random
    new_position = random.choice(possible_steps)

    # Move the agent to the randomly selected new position
    self.model.grid.move_agent(self, new_position)
```

Infection

- In our model, people can infect other people. So, we need a function to specify how this works.
- Infection here works as follows – if a person is infected, and they are in the same location as non-infected people, there is a probability that each non-infected individual will be infected – *transmissibility*.
- We do this by first getting a list of all the agents in the current cell. If there are more than 1 (ie more than just the current agent), then we go through each agent in the cell and, if they're not already infected, randomly infect them (also randomly sampling the disease duration for the newly infected agent at the same time).

Infection

```
# Agent infection method
def infect(self):
    # Get list of agents in this cell.  We use the get_cell_list_contents
    # function of the grid object and pass it our current position
    cellmates = self.model.grid.get_cell_list_contents([self.pos])

    # Check if there are other agents here - if the list of cellmates is
    # greater than 1 then there must be more here than this agent
    if len(cellmates) > 1:
        # for each agent in the cell
        for inhabitant in cellmates:
            # infect the agent with a given probability (transmissibility)
            # if they're not already infected.  If they become infected,
            # then we set their infected attribute to True, and their
            # disease_duration attribute to a value randomly sampled based
            # on the mean_length_of_disease attribute.
            if inhabitant.infected == False:
                if random.uniform(0, 1) < self.transmissibility:
                    inhabitant.infected = True
                    inhabitant.disease_duration = int(round(
                            random.expovariate(
                                    1.0 / self.mean_length_of_disease), 0))
```

Defining a “step”

- As well as our constructor, movement method, and behaviour method(s), we also need to specify a “step” function. This defines the series of things the agent will do at each time step.
- Here, our agents may move, may infect others (if they’re infected) and may recover (if they’re infected).
- First we check whether the agent is going to move on this time step (based on the level of movement probability). Then, if the agent is infected, the infect() function is called to potentially infect others, and the agent’s disease duration is decremented by one time step.

MOVE? → INFECT? → GET BETTER

Exercise 1

Over to you! I want you to take the disease_model_ex1.py file, open it in Spyder, and write the step method in the Person_Agent class.

The method should :

- First randomly determine whether the agent will move on this time step, based on the defined probability of this happening.
- If so, then make this happen.
- Next, you need to start infecting any other agents in the same cell, but only if this agent is infected.
- If the agent is infected, their remaining disease duration also needs to be decremented, and if the disease has now run its course, then this needs to be updated in the agent's attributes too.

Your code should run without errors, although it won't visibly do anything at this stage, as you're just declaring a class and its attributes and methods.

You have 20 minutes + 5 minute break. Work in your groups.

Writing the Model Class

- Now we need to write the Model class. This will control the agents and their environment, controlling how the model works.
- Again, we need to specify the __init__ method (constructor) first.
- We need to setup the number of agents, the grid that they'll live in, a schedule to determine the order in which agents act in each time step, and the starting cells for the agents.
- Quick question – why might the schedule be particularly important here…?

```python
class Disease_Model(Model):
    # 2D Model initialisation function - initialise with N agents, and
    # specified width and height.  Also pass in the things we need to pass
    # to our agents when instantiating them.
    # The comment below which uses triple " will get picked up by the server
    # if we run a live display of the model.
    """A model of disease spread.  For training purposes only."""
    def __init__(self, N, width, height, initial_infection, transmissibility,
                 level_of_movement, mean_length_of_disease):
        self.running = True # required for BatchRunner
        self.num_agents = N # assign number of agents at initialisation

        # Set up a Toroidal multi-grid (Toroidal = if the agent is in a cell
        # on the border of the grid, and moves towards the border, they'll
        # come out the other side.  Think PacMan :) The True Boolean passed in
        # switches that on.  Multi-Grid just means we can have more than one
        # agent per cell)
        self.grid = MultiGrid(width, height, True)
        # set up a scheduler with random order of agents being activated
        # each turn.  Remember order is important here - if an infected agent
        # is going to move into a cell with an uninfected agent, but that
        # uninfected agent moves first, they'll escape infection.
        self.schedule = RandomActivation(self)

        # Create person_agent objects up to number specified
        for i in range(self.num_agents):
            # Create agent with ID taken from for loop - we pass in the i
            # value as the unique_id, self (the model) as the model, and then
            # the various parameter values we specified
            a = Person_Agent(i, self, initial_infection, transmissibility,
                             level_of_movement, mean_length_of_disease)
            self.schedule.add(a) # add agent to the schedule

            # Try adding the agent to a random empty cell
            try:
                start_cell = self.grid.find_empty()
                self.grid.place_agent(a, start_cell)
            # If you can't find an empty cell, just pick any cell at random
            except:
                x = random.randrange(self.grid.width)
                y = random.randrange(self.grid.height)
                self.grid.place_agent(a, (x,y))
```

The Model's Step Function

- The model also needs a step function to define what it needs to do at each time step.
- Here, we just need it to step forward the schedule – ie activate the step function for each agent in the model (we're using a random activation schedule in this model, so the order will be random each turn)

```
# Function to advance the model by one step
def step(self):
    self.schedule.step()
```

The Run file

- Now we've written the model module, we need to write the run module, which will control how the model runs – either as a single run, a batch run, or using a server to visualise the model so we can watch it run live.
- For now, we're going to run the model as the last of these options – setting up a server so we can watch what happens in the model.
- This makes the writing of the run file very easy – we simply need to tell it to launch a server.

The Run file

- We’ll start a new .py file and call it disease_run.py. The entire code for this file will simply be as follows :

```
from disease_server import server
server.port = 8521 # the default
server.launch()
```

The Server file

- Now we've chosen to launch a server to visualise the model live, we need to write the server module to specify how the server will visualise the model.
- We'll call this file disease_server.py

The Server file

- First, we need to import the libraries we need :

```
# We'll first import from the module that we ourselves created!  We only need
# the Disease_Model class bit of the module here, so that's all we'll import
from disease_model import Disease_Model
# This will import the type of grid we want to visualise our agents
from mesa.visualization.modules import CanvasGrid
# This will import the ModularServer class, which allows us to create a new
# server to host the visualisation of our model
from mesa.visualization.ModularVisualization import ModularServer
# We'll also the UserSettableParameter class, which allows us to create
# user-interface elements, such as sliders, so the user can change model
# parameters
from mesa.visualization.UserParam import UserSettableParameter
```

The Portrayal Function

- We need to write a portrayal function. This function defines how agents will be visualised ("portrayed")
- We'll set it up so agents are represented as filled circles with radius 0.5, and are coloured red if infected and smaller and grey otherwise. The "layer" value determines which ones should go on top if more than one in a cell (higher value = on top).

```
# Portrayal function that defines how agents will be drawn onto the grid
# We specify that the function takes an agent as its input - it will draw the
# agent passed to it in the manner we define in this function
def agent_portrayal(agent):
    # Set up portrayal dictionary to store the key aspects of our portrayal
    portrayal = {"Shape":"circle", "Filled":"true", "r":0.5}

    # Specify visual characteristics for infected agents & non-infected agents
    # Here, we specify that if an agent is infected, then colour them red, and
    # put them on the lowest layer; otherwise, colour them grey, change their
    # radius to 0.2 (compared to the default of 0.5 we specified above), and
    # put them on a higher level so they'll appear on top of infected agents
    # if both occupy a cell
    if agent.infected == True:
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["r"] = 0.2
        portrayal["Layer"] = 1
```

Set up visualisation elements

- We now need to set up the elements that will make up our visualisation. For now, we're just going to set up one – a grid containing agents that will be portrayed in the manner we specified in the portrayal function. The grid will have 10 x 10 cells, and be 500 x 500 pixels.

```
# Set up visualisation elements
# Set up a CanvasGrid, that portrays agents as defined by the portrayal
# function we defined, has 10 x 10 cells, and a display size of 500x500 pixels
grid = CanvasGrid(agent_portrayal,10,10,500,500)
```

Set up user interface

- MESA allows us to easily add an interface to allow users to change parameter values to see how they impact the model.
- Here, we're going to add five sliders to allow the user to change :
  - The total number of agents
  - The probability that an agent is infected at the start of the model
  - The transmissibility of the disease (probability of infecting another agent)
  - The level of movement (probability of an agent moving at each time step)
  - The mean length of the disease in days (our time units)

Set up user interface

- For each user settable parameter, we need to specify the type (‘slider’ in this case), the text for the slider’s label, the default value, the minimum value allowed, the maximum value allowed, and the increment of the slider.

```
# Set up user sliders.  For each, we create an instance of the
# UserSettableParameter class, and pass to it that we want a slider, the
# label for the slider, the default, minimum and maximum values for the
# slider, and the increments of change for the slider.
number_of_agents_slider = UserSettableParameter(
        'slider', "Number of Agents", 20, 2, 100, 1)
initial_infection_slider = UserSettableParameter(
        'slider', "Probability of Initial Infection", 0.3, 0.01, 1, 0.01)
transmissibility_slider = UserSettableParameter(
        'slider', "Transmissibility", 0.2, 0.01, 1, 0.01)
level_of_movement_slider = UserSettableParameter(
        'slider', "Level of Movement", 0.5, 0.01, 1, 0.01)
mean_length_of_disease_slider = UserSettableParameter(
        'slider', "Mean Length of Disease (days)", 10, 1, 100, 1)
```

Set up the server

- The final step in the server file is to set up the server. This will be the server that will be launched by the run file.
- We specify the name of our model class (which we imported from our model module), the list of elements to visualise (just our grid at this stage), the title to display, and the sliders we've set up linked to the model variables they change.

```
# Set up the server as a ModularServer, passing in the model class we
# imported earlier, the list of elements we want to visualise (just the grid
# here), the title to display for the server visualisation, and each user
# interface we want to include (our sliders here) in a dictionary, where the
# index name in " marks must match the respective variable name in the Model
# Class, and the lookup value is the name of the slider we declared above
# (ie "name_of_variable":name_of_slider)
server = ModularServer(Disease_Model,
                       [grid],
                       "Disease Spread Model",
                       {"N":number_of_agents_slider, "width":10, "height":10,
                        "initial_infection":initial_infection_slider,
                        "transmissibility":transmissibility_slider,
                        "level_of_movement":level_of_movement_slider,
                        "mean_length_of_disease":mean_length_of_disease_slider}
                       )
```

Let’s run the model

We now should have a working model. Open up the following three files (in the disease_model directory) in Spyder:

disease_model.py

disease_run.py

disease_server.py

Then, run the disease_run.py file. In your groups, spend the next 15 minutes playing with the model. Try playing around with the sliders to see how things change. Click on the “About” button to see the comment we put in triple “ earlier. Read back through the code in each file to make sure you understand how it’s working.

A few things to note :

- If you change the sliders, you need to click “Reset” at the top of the model to reset and kick in the new values
- If you close the model and then want to launch it again, you’ll need to restart the Kernel, otherwise the port will be blocked. In Spyder, just hit CTRL + . in the iPython console window (or click the three horizontal lines (or cog for older versions) in the top right corner of the iPython console and select “Restart Kernel”.
- It’s possible you may have problems with launching the server, depending on how locked down your browser security is. If you do have problems, get someone in your group who can get it working to share their screen. Don’t worry – if you were building a real ABS model, you’d only use the visualisation server for showing people what’s going on, not for results generation.

DataCollectors

- It would be really useful to collect data throughout the run of the model so we can collate results and / or display the output on a graph
- In MESA we can do this easily using DataCollectors
- You may remember we imported the DataCollector class at the start of the disease_model.py file
- Now we’re going to use it!

Write a data collection function

- Let’s set up a DataCollector to monitor the total number of infected agents over time.
- First we need to write a function to calculate the result we want to collect (in this case, the total number of agents infected) in the disease_model module.
- We’ll do this as a standalone function in the module (ie not within a class)
- Well, I say “we”...

Exercise 2 – Writing the calculate_number_infected function

I want you to write the calculate_number_infected function in the disease_model module. Remember, this is to be a standalone function in the module – not a method of one of the classes we've defined. The function needs to :

- Accept a single input of a MESA model object (I'd recommend just calling it 'model' here)
- Sets up a counter that will keep count of the total number of infected agents in the model over time (which should default to 0).
- A list of agents in a MESA model can be obtained by accessing schedule.agents on the model object. Use a list comprehension to generate a new list which contains the Boolean 'infected' attribute of each agent in the schedule.
- Loop through your new list to count up how many infected agents there are, and update the counter you set up at the start of the function. (You could, if you wish, combine this and the above into a single list comprehension)
- Return this number as an output from the function.

You have 30 minutes. You should work in your groups.

Set up the DataCollector

- Next, we need to set up a DataCollector and specify the function we've just written as one of the "model reporters" (we can have "agent reporters" too).
- In the Disease_Model class, at the end of the __init__ method, we insert the following line of code :

```
# Create a new datacollector, and pass in a model reporter as a
# dictionary entry, with the index value as the name of the result
# (which we'll refer to by this name elsewhere) and the lookup value
# as the name of the function we created below that will calculate
# the result we're reporting
self.datacollector = DataCollector(
        model_reporters={"Total_Infected":calculate_number_infected},
        agent_reporters={}
        )
```

Set up the DataCollector

- We also need to tell the model to collect from the data collector at each step of the model. We add an instruction for the datacollector to collect data to the end of the Disease_Model class step() function (after calling the step() function on the schedule) :

```
# Function to advance the model by one step
def step(self):
    self.schedule.step()
    # Tell the datacollector to collect data from the specified model
    # and agent reporters
    self.datacollector.collect(self)
```

Output the results as a graph

- Now we've got the model collecting the number of infected people over time, all we need to do now is decide how to output this.
- Here, we're going to ask our server to draw a dynamic graph that shows the total number of infected people over time as the model runs.
- We need to go back into the disease_server module to do this.

Output the results as a graph

- We just need to do three things. First, add the ChartModule (used for drawing graphs) to our list of imports in disease_server.py

```
from mesa.visualization.modules import ChartModule
```

Output the results as a graph

- Next, specify the graph (we do this under where we specified the grid with grid = ...). We need to tell it we want a chart, the label for the data (which we specified when we set up the model reporters in the Data Collector), the colour of the line for that data, and the name of the datacollector that we'll be using to draw the graph (containing all the data series we want to plot – just one at the moment)

```
# Set up a chart to represent the total infected over time.  We instantiate a
# ChartModule for this, and pass in a dictionary containing the label for the
# data we're plotting and the colour of the line for that data, along with the
# name of the datacollector we want the chart to link to ('datacollector' here)
total_infected_graph = ChartModule(
        [{"Label":"Total_Infected", "Color":"Red"}],
        data_collector_name='datacollector'
        )
```

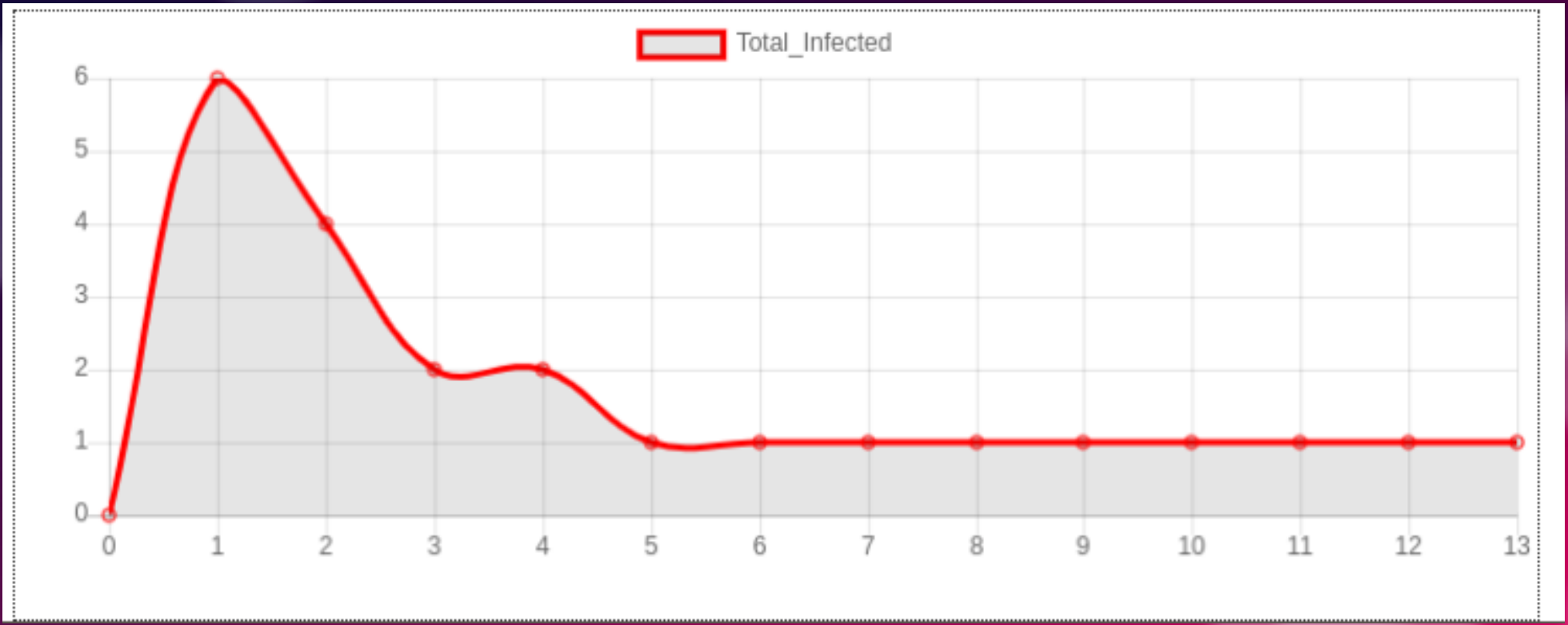
Output the results as a graph

- Finally, add the graph we've just specified to the list of elements we want the server to draw by adding it to the list we pass in (alongside 'grid') :

```
server = ModularServer(Disease_Model,
                       [grid, total_infected_graph],
                       "Disease Spread Model",
                       {"N":number_of_agents_slider, "width":10, "height":10,
                        "initial_infection":initial_infection_slider,
                        "transmissibility":transmissibility_slider,
                        "level_of_movement":level_of_movement_slider,
                        "mean_length_of_disease":mean_length_of_disease_slider}
                       )
```

Running with a graph

- Let’s run the model again



Agent Based Simulation and MESA Part 2

Dr Daniel Chalk

Batch Runs

- Live runs of the model are useful, but if we want to really explore the results of the model, we need to run our model in batch, running the model lots of times and testing different parameter values.
- MESA allows us to do this very easily using a library called batchrunner
- All we need to do is specify which parameter values we want to remain the same, which ones we want to change, how many times we want to run the model (and for how long) and batchrunner does the rest.
- I'll demonstrate using a modified version of the code we worked on in the last session that includes immunisation (this can be found in the folder disease_model_with_batchrunner; the immunisation version without batchrunner can be found in disease_model_with_immunisation). You can check this out in your own time.

Changing our disease_run module

- Up to now, we've just used the disease_run module to launch a server so we can watch the model run live.
- Now we're going to use the disease_run module to run the model in batch.
- First, let's comment out the server stuff from the run file as we don't want to launch the server if we're running in batch. But we can leave it here so if we want to view it live again, we can just uncomment it (and comment out the new code)

```
"""Use this section for watching model run on a server with visualisation"""

#from disease_server import server
#server.port = 8521 # the default
#server.launch()
```

Changing our disease_run module

- Now let’s write the file for a batch run. First, let’s import the libraries we need.

```
"""Use this section for running the model in batch runs"""

# Import everything from our disease_model module (all classes and functions)
from disease_model import *
# The arange function of numpy allows us to create evenly spaced numbers
# within a given interval - we'll use that below
from numpy import arange
# Import matplotlib so we can plot our results
import matplotlib.pyplot as plt
# Import the BatchRunner class so we can run our model in batch
from mesa.batchrunner import BatchRunner
```

Changing our disease_run module

- Now, let's specify the fixed parameter values – those inputs to the Model class constructor that we want to stay the same throughout all runs

```
# Set up a dictionary to store the attributes we want to keep fixed
fixed_params = {"width":10,
                "height":10,
                "initial_infection":0.8,
                "transmissibility":0.5,
                "mean_length_of_disease":10,
                "mean_imm_duration":20,
                "prob_being_immunised":0.05}
```

Changing our disease_run module

- Then, we need to specify those that we want to change on different runs. We need to specify the range of values to test, and the increment.
- Here, we'll test different numbers of agents from 2 to 100 (in increments of 1) and different levels of movement from 0 to 1 (in increments of 0.1)

```
# Set up a dictionary to store the attributes we want to change between
# tested scenarios, and the values to test.  Here, we'll test different
# numbers of agents in the model between 2 and 100 (in increments of 1), and
# different levels of movement from 0 to 1 in increments of 0.1
variable_params = {"N":range(2,100,1),
                   "level_of_movement":arange(0.0,1.0,0.1)}
```

Changing our disease_run module

- We also need to specify the number of iterations to run for each combination of parameter values tested, and the number of time steps in each run

```
# Run each combination 5 times, for 365 days of simulated time
num_iterations = 5
num_steps = 365
```

Changing our disease_run module

- Now we just need to create a batchrunner with all this information and tell it to run through all combinations. We'll also store the results in a Pandas DataFrame, and have a look at it once it runs.

```
# Set up a BatchRunner with the above information
batch_run = BatchRunner(Disease_Model,
                        fixed_parameters=fixed_params,
                        variable_parameters=variable_params,
                        iterations=num_iterations,
                        max_steps=num_steps,
                        model_reporters=
                        {"Total_Infected":calculate_number_infected,
                         "Total_Imm":calculate_number_immunised}
                        )

# Tell the BatchRunner to run
batch_run.run_all()

# Store the results in a Pandas DataFrame
run_data = batch_run.get_model_vars_dataframe()
```

CAUTION!!!!!

- Be careful how many different combinations of parameter values you test in your batch run. You can very easily get to huge numbers of combinations that can bring down even the fastest computers...

Exercise 3

You are now going to use everything you've learned in this module, including your new skills in MESA, to **design** and **build** an Agent Based Simulation.

You will work in your groups for 2 hours. You should do the following :

- Discuss some ideas about potential problem areas that you could use Agent Based Simulation to tackle. You may choose one of the ideas you came up with in the Cellular Automata session if you like, or think of something new!
- Choose one of these ideas, and draw up a design for how the model would work – what will be your agents? What behaviours and movement patterns will they have? Will they interact, and if so, how? What will your cells represent? What kind of cell neighbourhoods will you have etc
- Build a prototype "dummy" model using MESA. I'm not expecting you to have the data you need to parameterise it (although maybe you will) but just have a go at building your ABS design in MESA. Put on a nice interface, including any user interface elements (e.g. sliders) to allow the user to play around with your model.

After 2 hours, I'll ask each group to present their models, and we'll pick a winner! And I'll also ask you all to share them on Slack so other people can play with them too!

Further Reading

- MESA Documentation and Tutorials :

  https://mesa.readthedocs.io/en/master/
- My thesis (if you don’t want to pay £69) :

  https://ore.exeter.ac.uk/repository/bitstream/handle/10036/96455/ChalkD.pdf?sequence=1
- Individual Based Modelling and Ecology

  Book by Volker Grimm and Steven F Railsback

  (ignore the title – good for general ABS interest)
- Cellular Automata

  http://mathworld.wolfram.com/CellularAutomaton.html