| **Modelling complex systems**   1. Agent based models 2. Continuous mathematical system modelling | |
| --- | --- |
| When systems are too complex to model with workable mathematical equations or statistical analysis, agent based models (ABMs) can be useful. ABMs are models of systems that simulate interactions between agents and their environments in a way that can reveal interesting features that are difficult to observe any other way. In this way, ABMs can be thought of as synthetic realities that we ‘play with’ in order to understand how systems work.  ABMs are most useful for systems in which there is an agent making ‘decisions’, usually in response to other agent behaviour, or changes in the environment.  ABMs have many uses, but one of the most interesting features is the idea of [*emergence*](https://en.wikipedia.org/wiki/Emergence). You can read more about ABMs in the social sciences [here](https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291099-0526%28199905/06%294%3A5%3C41%3A%3AAID-CPLX9%3E3.0.CO%3B2-F?casa_token=LM_U38SKoe8AAAAA:gfQ8rW_8jwJzgliCuqIzt_7UnyzDMW4RHcodfsVDAycScuW-muP5Hug1DSifjq1jlt3nHJ7KglkoTqQ). | |
| **Copy and paste the code below, and run it in RStudio:**  #initialise  n <- 20  grid <- matrix(sample(c(0,1),n^2,replace=TRUE),nrow=n)  timeSteps <- 10  #function to calculate the next state  nextState <- function(grid) {  n <- nrow(grid)  newGrid <- matrix(0,nrow=n,ncol=n)  for (i in 1:n) {  for (j in 1:n) {  #this line of code calculates the number of living neighbours for a  #given cell(i,j) in the grid, considering the boundaries of the grid,  #and stores the result in the variable aliveNeighbors.  aliveNeighbors <- sum(grid[max(i-1,1):min(i+1,n),  max(j-1,1):min(j+1,n)])-grid[i,j]  if (grid[i,j]==1 && (aliveNeighbors<2 || aliveNeighbors>3)) {  newGrid[i,j] <- 0  } else if (grid[i,j]==1 || aliveNeighbors==3) {  newGrid[i,j] <- 1  }  }  }  return(newGrid)  }  #run it  for (t in 1:timeSteps) {  grid <- nextState(grid)  image(1:n,1:n,z=grid,col= c("white","black"),axes=FALSE,xlab="",ylab="")  title(paste("Time step:", t))  Sys.sleep(1) #pause for a second to see the evolution  } | |
| **Q1. Looking at the visual output ABM, briefly describe what appears to be happening on the output image. Does the pattern appear ‘random’, or is there something else going on?**  *Note: This is my R implementation of the ‘Game of Life’ simulation, which is similar (but not identical) to one of the first known ABMs. This one was designed by* [*John Conway*](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)*. This model is a cellular automata, which is a simple ABM in which the state of an agent is entirely dependent on the states of agents nearby.* | |
| One of the common features of the simulation code you see in the assignment sheet is the *for loop*. These loops are used to step time forward in discrete time ABMs. If simulations can be thought of as synthetic reality, then the for loop measures the passage of time in discrete steps (such as minutes, hours, days etc.). The more loops, the longer the simulation runs. | |
| **Copy and paste the code below, and run it in RStudio:**  gridSize <- 10  numAgents <- 15  timeSteps <- 10  #initialize agents  colours <- c("red","blue","pink","black","grey","green","cyan","yellow")  style <- c(19,20,21,22)  agents <- data.frame(x = sample(1:gridSize,numAgents,replace = TRUE),  y = sample(1:gridSize,numAgents,replace = TRUE),  c = sample(colours,numAgents,replace=TRUE),  p = sample(style,numAgents,replace=TRUE))  #agent behaviour  updateAgents <- function(agents) {  for (i in 1:nrow(agents)) {  move <- sample(c("up","down","left","right"),1)  if (move == "up" && agents$y[i] < gridSize) agents$y[i] <- agents$y[i] + 1  if (move == "down" && agents$y[i] > 1) agents$y[i] <- agents$y[i] - 1  if (move == "left" && agents$x[i] > 1) agents$x[i] <- agents$x[i] - 1  if (move == "right" && agents$x[i] < gridSize) agents$x[i] <- agents$x[i] + 1  }  return(agents)  }  #simulation  for (t in 1:timeSteps) {  plot(agents$x,agents$y,xlim=c(1, gridSize),  ylim=c(1,gridSize),  pch=agents$p,col=agents$c,xlab="X",ylab="Y")  title(paste("Time Step:", t))  Sys.sleep(0.5)#pause to visualise...  agents <- updateAgents(agents)  } | |
| **Q2. Looking at the code, describe what is happening in this ABM. Try to describe the behaviour of the agents with reference to the code.** | |
| Download [this code](https://drive.google.com/file/d/1zQxKVvdeFR6LULvP575II9jxicpe2YtJ/view?usp=sharing) for a forest simulator. This model is a simple simulation of forest growth. It takes into account some spatial data (‘soil suitability’) as well as some constants (like sunlight, water and growth rate). It can be made more complex by creating tree varieties, as well as the impact of neighbouring trees on growth.  Some of the code is not part of the simulation proper—for example the smooth\_matrix() function just smoothes out the spatial variation in the simulated soil data. If you had real soil data, the simulation could be simplified.  **Q3. Read through and experiment with the code. What are the key functions that define agent behaviour, and how do they work?** | |
| Some simulations are systems of equations that can be explored to understand complex processes. The examples below are not **true** agent based models, but instead, are mathematical models based on systems of equations. They don’t represent the same level of complexity—since they are precisely defined by the mathematical equations. However, they are useful for preparing and building more complex ABMs. | |
| **Copy and paste the code below, and run it in RStudio:**  install.packages("simecol")  library(simecol)  #predator-prey model  predator\_prey <- new("odeModel",  main = function(time, init, parms) {  with(as.list(c(init, parms)), {  dPrey=r\*Prey-a\*Predators\*Prey  dPred=-s\*Predators+e\*a\*Predators\*Prey  list(c(dPrey,dPred))  })  },  parms = c(r=0.5,a=0.02,s=0.5,e=0.1),  #r - prey birth rate  #a - predation rate  #s - death rate of predators  #e - efficiency of converting food (prey) into predator births  times = c(from=0,to=200,by=1),  init = c(Prey=40,Predators=9), #starting populations  solver = "lsoda"  )  predator\_prey <- sim(predator\_prey)  plot(predator\_prey) | |
| The two equations (dPrey and dPred) in the code represent the rate of change in the populations of prey and predators in this model. Note that you can change the values of the four constants (r, a, s and e) and observe changes in the predator and prey population trends over time.  **Q4. Find a way to plot all the results on a single plot instead of on two separate plots. Write out your code.** | |
| **Copy and paste the code below, and run it in RStudio:**  #innovation diffusion model  #A is the proportion of population that has adopted the innovation  #R: The rate at which individuals adopt innovation  #B: The rate at which individuals abandon innovation  #I: The rate at which adopters influence non-adopters to adopt innovation  inn\_diff <- new("odeModel",  main = function(time, init, parms) {  with(as.list(c(init, parms)), {  dA=R\*(1-A)-B\*A+I\*A\*(1-A)  list(c(dA))  })  },  parms = c(R=0.1,B=0.02,I=0.05),  times = c(from=0,to=100,by=1),  init = c(A=0.01),  solver = "lsoda"  )  #simulate  inn\_diff <- sim(inn\_diff)  plot(inn\_diff,main="Innovation diffusion") | |
| This model predicts the diffusion of innovation (technology, ideas, etc.) over time in a population.  **Q5. Come up with a new parameter to add to the equation predicting the rate of diffusion. Don’t worry about mathematical rigour; just modify the code to add a new term and ensure that the code runs without warnings/errors. Explain the change you’ve made, and how it alters the behaviour of the model.** | |
| **Copy and paste the code below, and run it in RStudio:**  #define the SIR model  SIR\_model <- new("odeModel",  main = function(time,init,parms) {  with(as.list(c(init,parms)), {  dS = -b \* S \* I  dI = b \* S \* I - g \* I  dR = g \* I  list(c(dS, dI, dR))  })  },  parms = c(b = 0.3, g = 0.1),  times = c(from = 0, to = 100, by = 1),  init = c(S = 0.99, I = 0.01, R = 0),  solver = "lsoda"  )  SIR\_model <- sim(SIR\_model)  plot(SIR\_model) | |
| The above code is an implementation of a very well known compartmental model for understanding infectious disease dynamics—the SIR model (susceptible, infectious and recovered). This model has two control variables: b and g (typically ‘beta’ and ‘gamma’ in the literature). Beta is the transmission rate, and gamma is the recovery rate. The line:  init = c(S = 0.99, I = 0.01, R = 0)  sets out the proportions of the population in each compartment at the start of the simulation. | |
| **Q6. Modify the code to account for the fact that immunity to infection is not permanent.** Here are the new equations you need to add to the code:  dS = -b \* S \* I + d \* R  dI = b \* S \* I - g \* I  dR = g \* I - d\* R  where d is the lapsing immunity rate (how fast recovered lose immunity). You will need to make other changes to the code to get it to run. | |