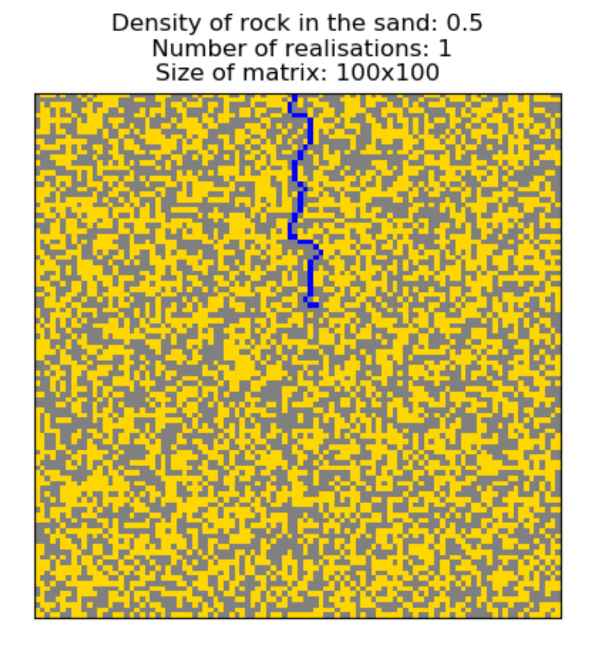
**Computational Mathematics – Summative Coursework**

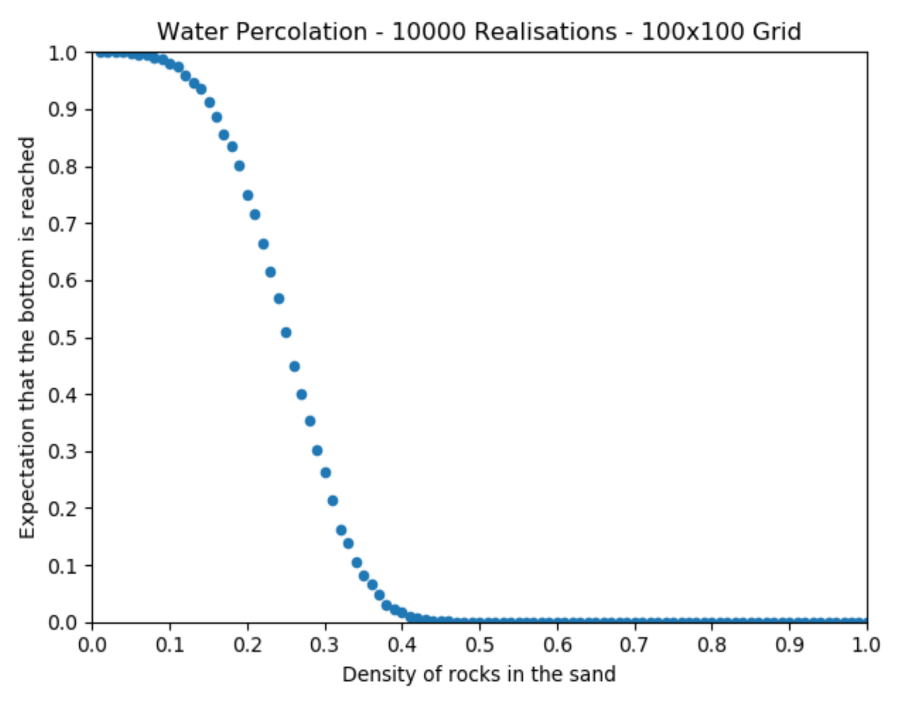
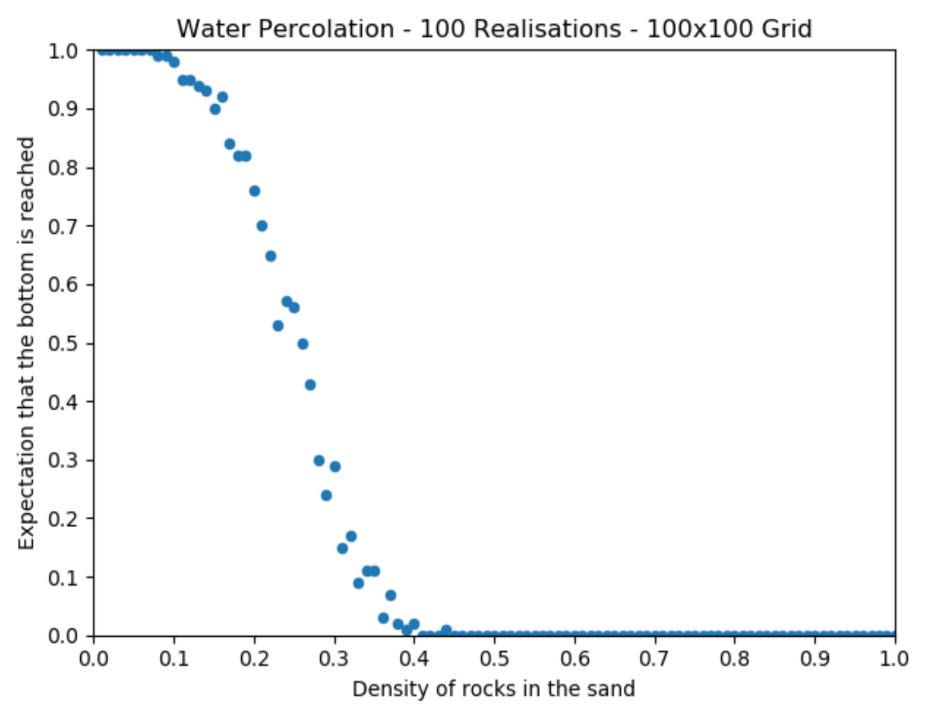
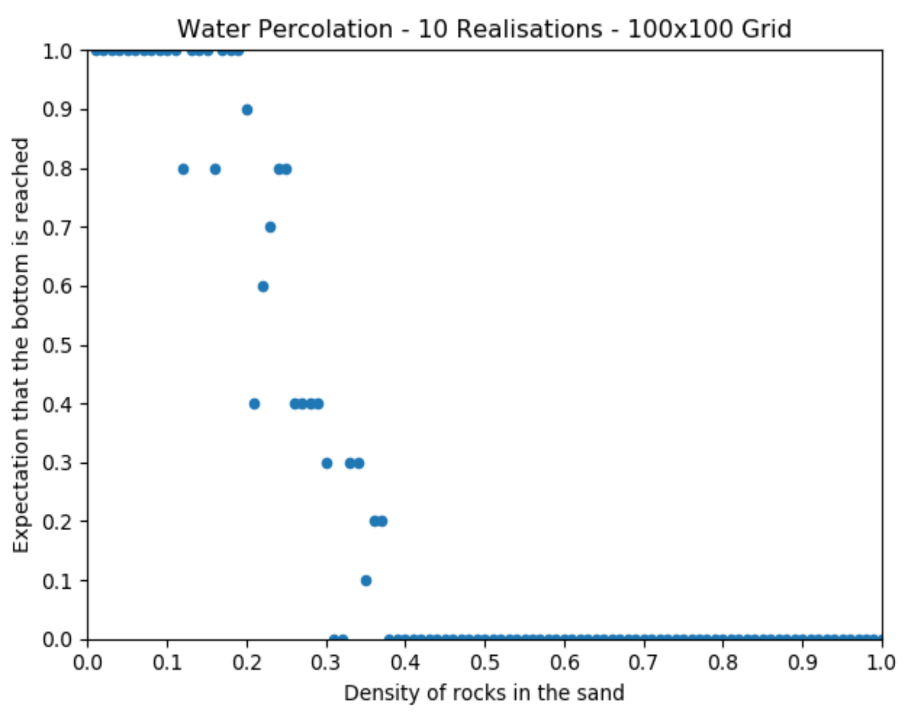
**Written Report - Percolation Systems**

Abstract: This report looks at how an entity percolates through different systems and the analytical applications of such system as they exist in the real world. The two main system that we will look at are water percolating through a mix of rock and sand as well as a fire percolating through a forest.

(i) For 𝑁 = 100 establish where the critical percolation level occurs 𝑝𝑐 by varying 𝑝 and use python graphics to show the 2d system. Use differing numbers of realisations.



The figure to above shows an animated simulation of the types of systems that we will be discussing. In this example we can see that the yellow blocks represent areas of sand where it is possible for the water to move to, whilst the grey block depict rock, which the water is unable to travel through. The user can simulate either individual frames of the water percolating through the sand/rock mixture or instead can chooses to only animate the final frame of each realisation. The density of rock within the sand is shown at the top of the diagram along with the number of realisations and size of the matrix. The animation loads in a webpage where the animation plays on a continuous loop where it can be skipped through and paused to view individual frames.



Critical percolation occurs when the water is first able to reach the bottom of the grid. We can find this value by varying the density of rocks in the sand and evaluating which proportion of rocks corresponds to the water being able to reach the end of the system.

When testing the percolation system using matrix size as the dependent variable and varying the independent variable number of realisations, we can see that for a small number of realisations it is very difficult to predict the value for critical percolation. This is show by the first graph where the points are too spread out to give any clear indication as to where critical percolation occurs.

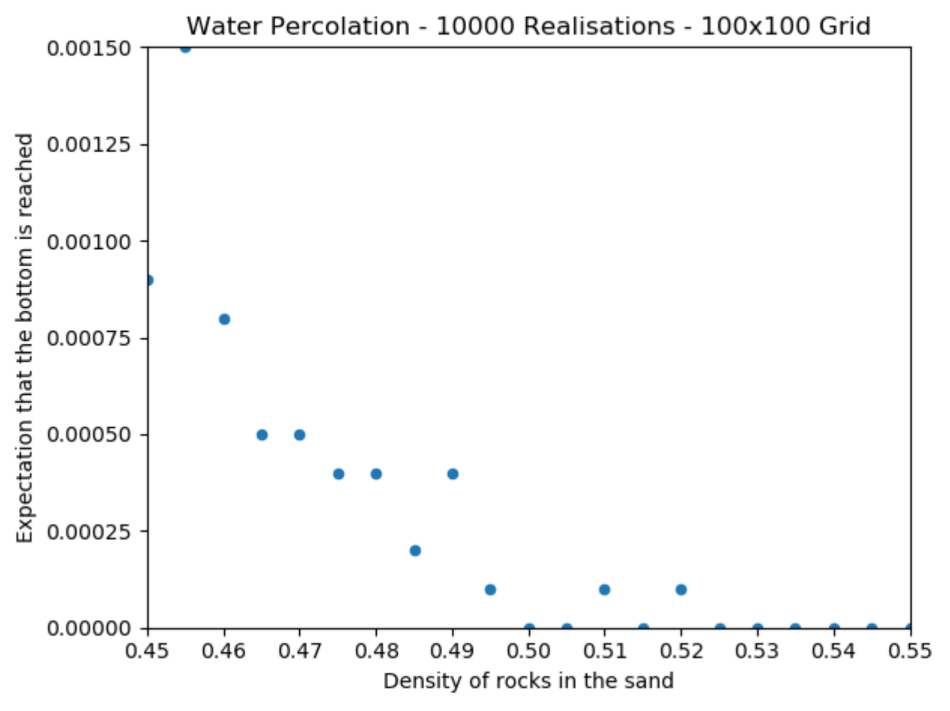
As we increase the number of realisations, depicted in the subsequence graphs, we can see as we increase the number of realisations the points start to line up along a single curve.

By the time we have completed 10000 realisation we can make a reasonable approximation of critical percolation for a given size matrix.

Looking at the graph we estimate that critical percolation occurs somewhere in the region of 0.45 and 0.55 where the curve starts to lift away from the x-axis.

In fact, my system will record the critical percolation at for each number of realisations, as such we are able to compare these values to the approximations given by the trend line of the graph. We can see that our approximation is very similar to the true value of critical percolation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Critical Percolation** | **Number of Realisations** |  |  |
| 0.38 | 10 |  |  |
| 0.41 | 100 |  |  |
| 0.46 | 1000 |  |  |
| 0.52 | 10000 |  |  |
|  |  |  |  |



This graph shows and enlarged portion of the previous graph. From here it is far easier to see that the critical percolation for a 100x100 grid occurs at 0.52 density of rock within the sand.

We can also see from this graph that there is some variance in expectation for small increments in density at each level of realisation, but we can still comfortably say that our value is correct to 1 decimal place. This is because an average is being taken at each level of realisation and since we are using a random generator to form the matrix this is also somewhat likely to be a factor affecting the accuracy.

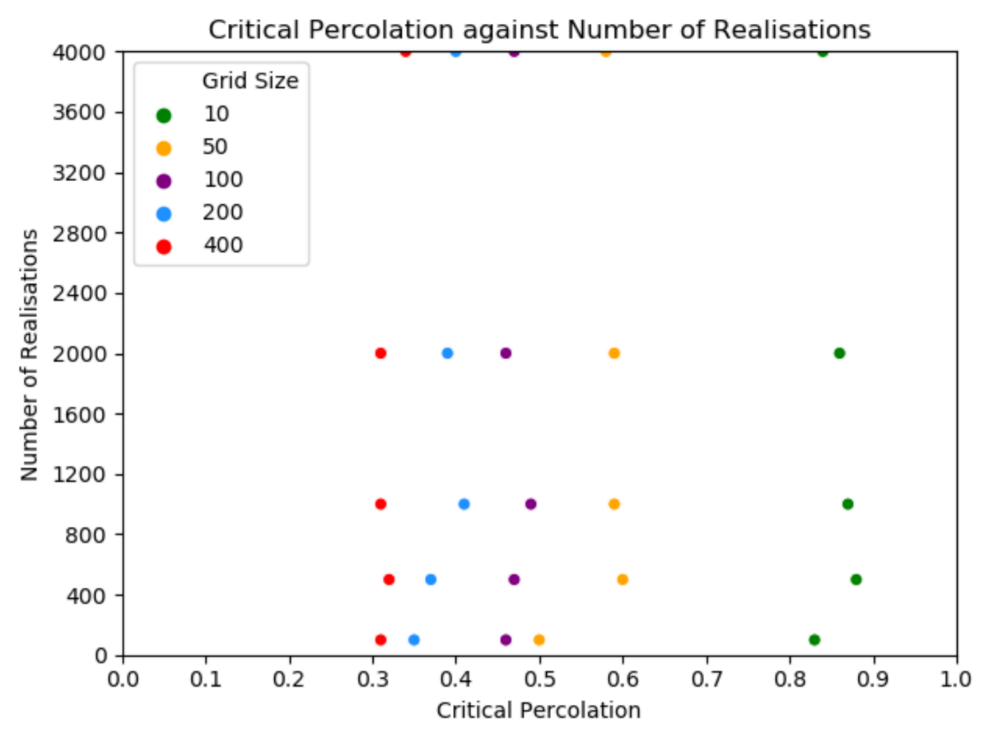
(ii) Assess how 𝑝𝑐 varies for 𝑁 = {10,50,100,200,400} and nrep = {100,500,1000,2000,4000}. In the large 𝑁 limit it is expected that 𝑝𝑐 will give a constant ‘universal’ value, discuss the performance of this algorithm in this context.

The graph below shows the variation in critical percolation for different size grids.

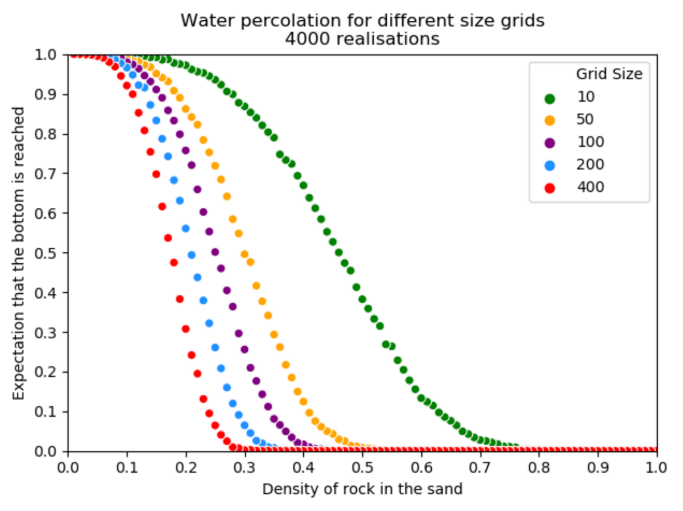
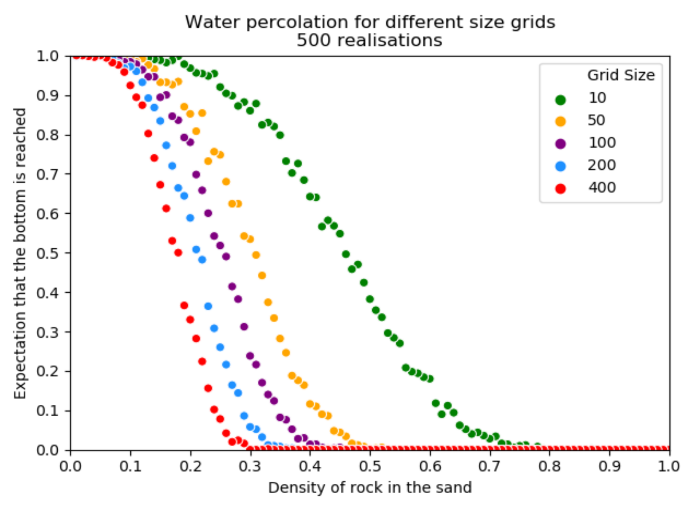
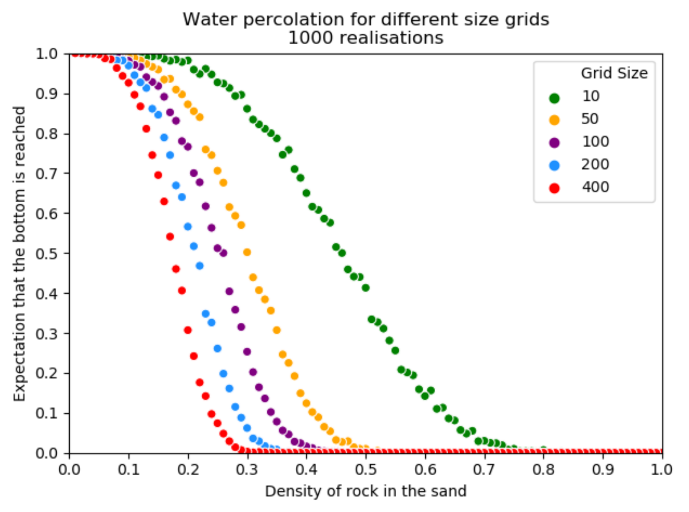
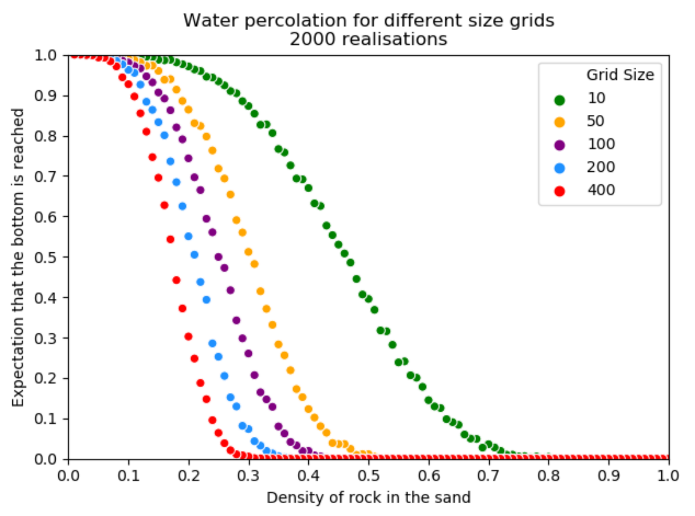
|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Critical Percolation** |  |
| 10 | 0.84 |  |
| 50 | 0.57 |  |
| 100 | 0.55 |  |
| 200 | 0.39 |  |
| 400 | 0.32 |  |
|  |  |  |

The above graph shows how matrix size affects critical percolation. These number were determined using 4000 realisations for each size grid because this give the most accurate value of critical percolation. Increments in density were also made finer reducing the step from 0.05 to 0.01. From this graph we can see a clear trend showing that as matrix size increases critical percolation decreases. One may expect that for larger matrices there are likely to be more alternative routes that the water can follow in the event that any given route is blocked. Therefore, on average the water is more likely to reach the end of a larger grid compared to a smaller one both with the same density of rock. In this instance it appears however that the converse is true, that with a greater number of blocks available there is an increased number of arrangements of rock that could lead to it getting trapped or that because the water has further to travel there is a higher chance of it being stopped. Because the trend line can be closely approximated to a decreasing logarithmic curve there is evidence to suggest that as N 🡪 **∞** critical percolation tends towards a value around 0.25.

|  |  |  |
| --- | --- | --- |
| **Grid Size** | **Number of Realisations** | **Critical Percolation** |
| 10 | 100 | 0.83 |
| 10 | 500 | 0.88 |
| 10 | 1000 | 0.87 |
| 10 | 2000 | 0.86 |
| 10 | 4000 | 0.84 |
| 50 | 100 | 0.5 |
| 50 | 500 | 0.6 |
| 50 | 1000 | 0.59 |
| 50 | 2000 | 0.59 |
| 50 | 4000 | 0.58 |
| 100 | 100 | 0.46 |
| 100 | 500 | 0.47 |
| 100 | 1000 | 0.49 |
| 100 | 2000 | 0.46 |
| 100 | 4000 | 0.47 |
| 200 | 100 | 0.35 |
| 200 | 500 | 0.37 |
| 200 | 1000 | 0.41 |
| 200 | 2000 | 0.39 |
| 200 | 4000 | 0.4 |
| 400 | 100 | 0.31 |
| 400 | 500 | 0.32 |
| 400 | 1000 | 0.31 |
| 400 | 2000 | 0.31 |
| 400 | 4000 | 0.34 |

The above graph shows how critical percolation varies with the number of realisations for different size matrices. There is a subtle trend show in this graph where, as the number of realisations for any matrix size increases the critical percolation value tends towards a specific value. For any chosen size of matrix, the difference in distance in the x-axis representing critical percolation decreases as you move up the graph with increased realisations. Critical percolation gets iteratively closer to its true value as you move up the graph. It would be nice to explore this trend further, but I was restricted by the computational limits of my computer, even after reducing unnecessary computational tasks, to generate this graph still required leaving the simulation running overnight.

This would enable me to find a more accurate value for critical percolation as N 🡪 **∞** and could be repeated for different types of systems of percolation to determine if any underlying pattern exists within these structures.

Perhaps more noticeable on the graph below in how for every increase in grid size the curve representing expectation gets shifted further to the right as it tends towards a higher value. From this we can deduce this matches what we would expect because as N increases the water is less likely to percolate the system and as such a larger percolation probability is required for the system to percolate i.e. the density of needs to be increase though our system takes its measurements relative to the density of rock. The same pattern is observed for other numbers of realisations where varying size matrices are plotted on the same axis. 

The performance of this algorithm is significantly hindered at greater numbers of realisations combined with larger grids. For these bigger simulations it becomes increasing difficult to animate event the final frame of each system. Even having the animations turned off you are left waiting a prolonged period of time for some of the giant percolations to end. To attempt to speed up the rate at which I can compute the required values I have attempted to fix or at least reduce the impact of some the hindrances facing the program. Among these are a reduction in unnecessary computationally expensive tasks such as transferring data to different data structures and only showing key information to the user.

To get around this all data is saved to a single Pandas data frame which is then converted to an excel spread sheet after all the simulations have run to completion. This reduces the need to move data between different structures and has the added benefit of allowing the user to go back and review data even after the program has been closed. Furthermore, I have incorporated a simple Boolean switch to change between displaying and ignoring the animation function built into the program. There is also I secondary ‘Separate’ switch which lets the program know that the user would like to see animations but is concerned with efficiency, as such the program will create one larger animation after everything has finished instead of multiple smaller animations at each level of realisations and matrix sizes.

(iii) More general percolation algorithms are needed to evaluate the percolation threshold 𝑝𝑐 for forest fires, electrical discharge and more widely. Suggest an algorithm you could use to find 𝑝𝑐 in these cases and have a go at implementing it. Read about other percolation theory applications. Explain in 400 to 500 what their main features are (hint: what types of system do they represent) and the type of applications. Include 1 figure or table from parts 7(i) or (ii) to illustrate this.

This leads me on to discussing how one could in the example of a forest fire simulation calculate the necessary values require to analyse the percolation system. Whereas with the water droplet it is fairly simple to count the number the number of layers through which the water has fallen a different approach is required when modelling other types of percolation. Consider a forest fire spreading outwards from central point where lightning has stuck a tree within a forest and set it alight. Here there are multiple approaches to finding data useful for making conclusions about this type of scenario. You could consider counting the total area occupied by the fire compared with the area occupied by trees and mud or alternatively you could measure the spread of the fire by treating each ring of block surrounding the original fire as a tier and counting the number of tiers that the flames have travelled outwards in a concentric circular fashion. This is an example of only one types of system though there are likely to be many more consideration to account for when looking at more complex methods of percolation. This nicely leads me on to my own algorithm for depicting any measuring the spread of fire within such a framework. Here we first need to contemplate how a fire would move through the forest. Whereas before we have these given steps that the water droplet could travel making its way through the sand and rock:

## 1. If the space directly below is sand, move there.  
## 2. If the space below and to the left is sand, move there.  
## 3. If the space below and to the right is sand, move there.  
## 4. If the space directly to the right is sand, move there.

We now need to think about the properties of fire and how it is not confined to only traveling in these directions but can instead grow in any direction. Therefore, for the purposes of my simulation I am considering any individual block within my grid containing fire as being able to make any of these moves in order of precedence left to right:

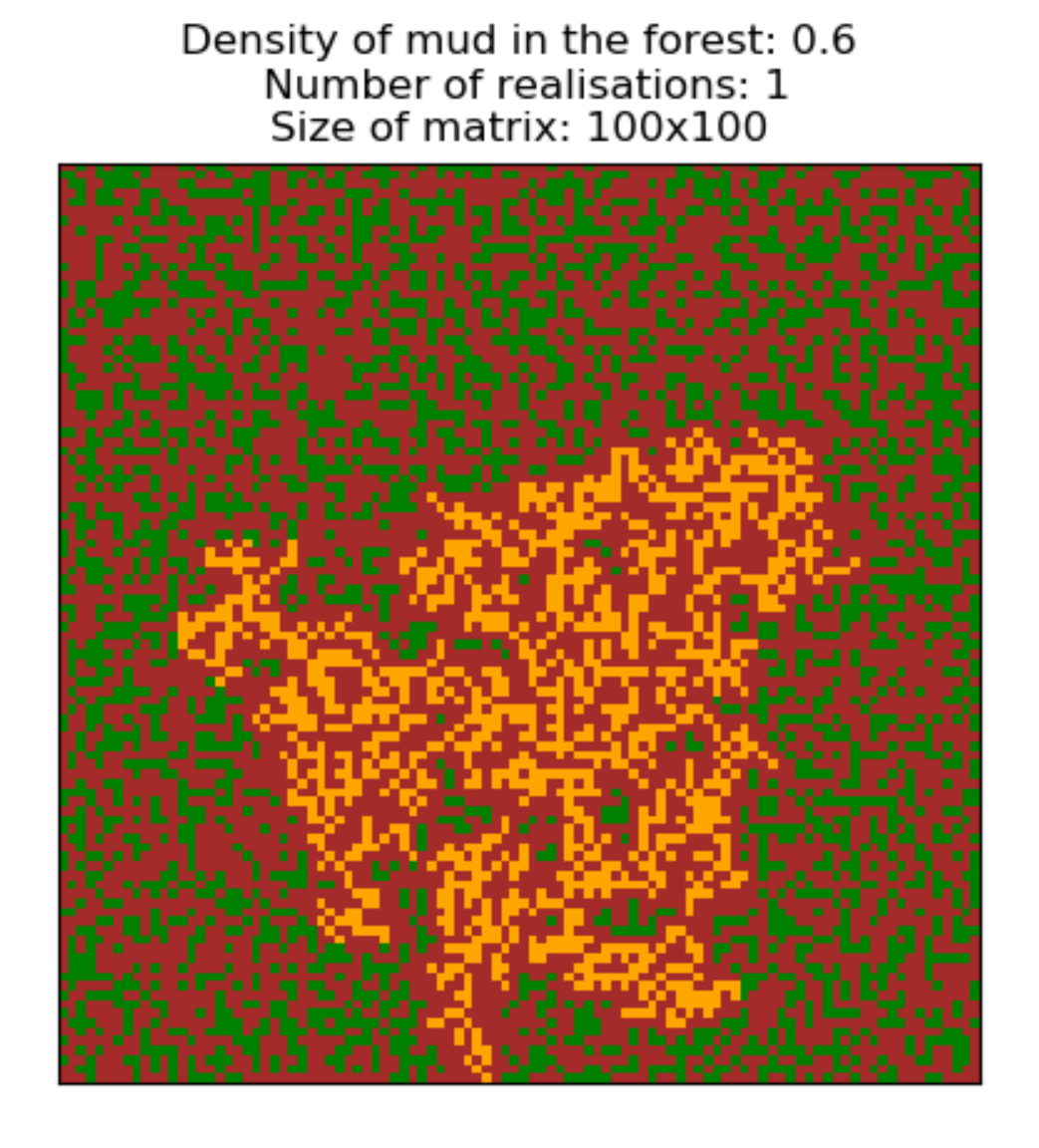
neighbours = ((-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0))

This is equivalent to selecting a fire block and determining whether any of the 8 surrounding blocks contain trees not already on fire starting with the top-left block in a 3x3 square and moving clockwise round all 8 outer blocks until all surrounding trees are on fire. From here the process is repeated for any blocks newly set alight until either the fire reaches the edge of the forest or is stopped by a wall of mud in all directions and so cannot travel any further.

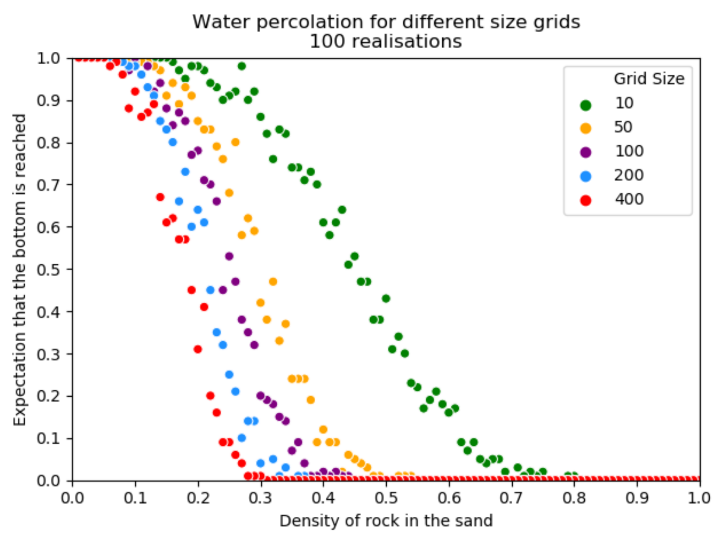
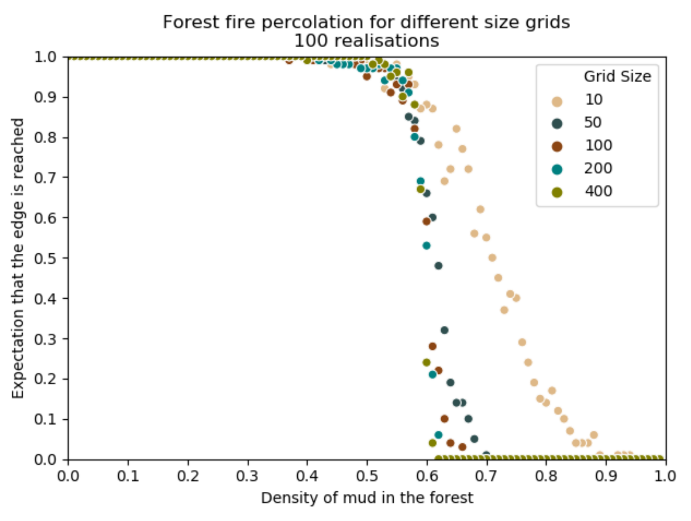
As mentioned previously this system incorporated the method of measuring concentric tiers expanding outwards from the central starting block as being the means to determine distance over which the fire has spread. There is also an incremental counter for whenever the fire is not stopped and continues spreading until it reaches the edge of the forest edge.

To make readability easier my program for the forest fire follows a very similar form to that of the water percolation simulator with the main difference being the ways in which the fire spreads as well and the means by which its progress is measured.

Below is a view of a similar animation as it pertains to the forest fire simulation.



Observe that the simulation is stopped whenever the fire reaches the edge of the forest. If unsuccessful we would still be able to measure distance over which the fire travelled by taking the greatest distance between any two points in the same axis. This could later be expanded to include measurements taken diagonally but for this simple base case I felt the first of these two options was most appropriate as it is less computationally expensive so allows for larger simulation to be run more quickly. This second program has all the same functionality as the first and I have included a link to all the source code and spreadsheets containing the data used to construct the graphs. There is also a folder containing images of more graphs which I chose not to append to this report in order to reduce its length, but which can be viewed via the link provided at the end of the report.

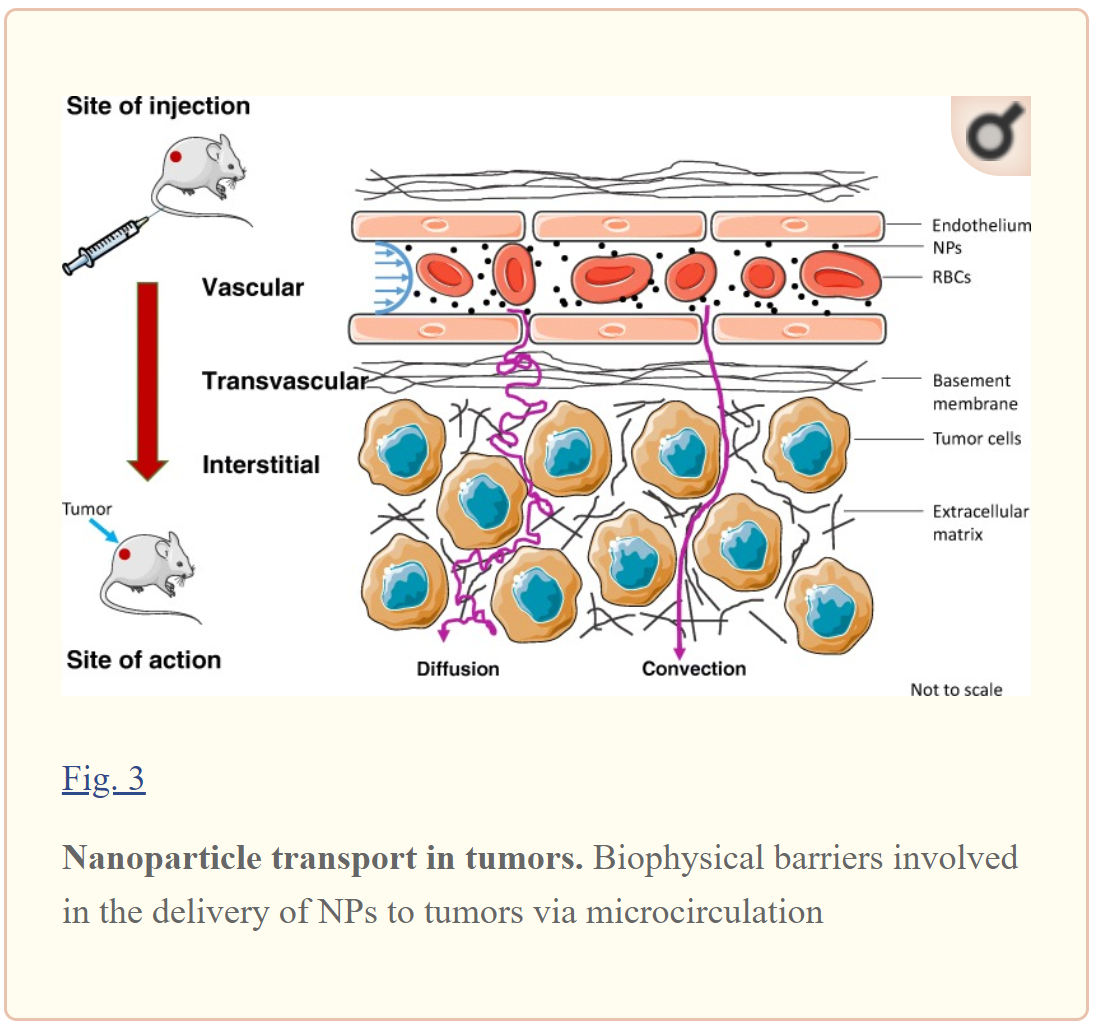
 

These graphs show how the two systems vary in critical percolation for different size grid. From this we can determine that for an equivalently sized forest fire grid the critical percolation level occurs at a greater level that that of the water matrix. This is likely due to the increased number of available positions that the fire can travel to compare with the direction of travel of the water. As mentioned previously the fire can spread to any of the surrounding eight blocks covered in trees whilst the water in only able to follow a specific sequence of moves which greatly reduces the number of pathways through which the water could travel to reach the end of the matrix. Therefore, an increase in the limiting factor rock would have an affect much more quickly for the water system compare to increasing the density of mud for the forest fire simulation which would reduce percolation of the fire more slowly. We can say that both systems follow a similar curve as density of the limiting factor varies but the forest has a right shifted line of best fit so is more easily percolated.

Read about other percolation theory applications. Explain what their main features are (hint: what types of system do they represent) and the type of applications

The theory behind systems such as the ones mentioned in this report could be extended to represent far more complex types of percolations. For my research into the wider applications of these models I wanted to consider the applications in investigating the effectiveness of precision medicine and site-specific drug delivery. Percolation theory can be applied in these instances to determine the quickest means of delivery for targeted therapy. Blood vessels can be mapped to a system of interconnected nodes where the body is considered as a series of compartments through which the nanomedicine travels.

The system considers the travel of gene vectors as vehicles for delivering foreign DNA into recipient cell, which in the case of this study applies to tumour cells. Therefore, it is important to consider the ability of therapeutic agents to percolate into the tumour tissue. In general, solid tumours are known to possess heterogeneous structures composed of blood vessels, and clusters of tumour cells with oxygen deficient regions. Therefore, it might be difficult to deliver the therapeutic agents to tumour cells distant from the vasculature, this could be represented in the model by having weighted routes of travel through the vascular network. Changes could be made to the percolation logic to more accurately represent real world occurrences of blocked or narrow blood vessels and to account for differing rates of diffusion for separate parts of the circulatory system.

****

This diagram gives a visual representation of how such a percolation system operates, with the desired treatment percolating through the different biological levels as it travels through the body making its way to the required site.

By analysing multiple replications of similar simulations, we are better able to predict the path travelled by such treatment to improve our ability to target the cancer's specific genes, proteins, or the tissue environment that contributes to the growth and survival of cancer cells.

For such an advanced system there will inevitable be many other factors that need to be taken into consideration when evaluating the critical percolation against change in independent variables among these could be blood pressure, concentration gradient, size of the particles that are diffusing, and temperature of the body all of which could affect the accuracy of determining when percolation occurs. With a greater number of realisations, a model like the one described above is far less likely to be affected by any individual factor. Furthermore, this same approach can be applied to both global and microscopic aspects of biodistribution. This is analogues to the above approach of taking the water percolation model and applying the same methodology to create a system for modelling forest fires. This biomolecular modelling approach is continuously adaptable as further advances are made in the field of drug therapy and new techniques and tools are developed to better understand the human body. For the time being, applications utilising percolation theory have already allowed for significant improvements in the level of care received by patients.

**References:**

Ncbi.nlm.nih.gov. (1990). *A modelling analysis of monoclonal antibody percolation through tumours: a binding-site barrier. - PubMed - NCBI*. [online] Available at: https://www.ncbi.nlm.nih.gov/pubmed/2362198 [Accessed 4 Mar. 2020].

Acanceresearch.com (2018). *Precision Medicine and Site-Specific Drug Delivery*. [online] Available at: *http://www.acanceresearch.com/cancer-research/precision-medicine-and-sitespecific-drug-delivery.php?aid=7173* [Accessed 4 Mar. 2020].

Ncbi.nlm.nih.gov. (2019). *Mathematical modelling in cancer nanomedicine. - NCBI*. [online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6449316/ [Accessed 4 Mar. 2020].

Ncbi.nlm.nih.gov. (2009). *Enhanced Percolation and Gene Expression in Tumour Hypoxia by PEGylated Polyplex Micelles - NCBI*. [online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2835242/ [Accessed 4 Mar. 2020].

**This link will grant access to all the spreadsheets and graph images relating to this project as well as the original source code: https://1drv.ms/u/s!Ar4UTJegDrU6nt1PDpXt3Qp8\_-6Tvw?e=QkgjUA**

**Below is the source code for my project including both water and forest fire percolation systems. It is worth noting that some of the formatting may have been altered when copying it to the text editor, as such should you wish to run the code, I have provided the final program version v19.**

# -\*- coding: utf-8 -\*-  
"""  
Created on Thu Feb 13 20:23:33 2020  
  
   
"""  
## Simulate a drop of liquid percolating down through a mixture of  
## sand and rock.  
##  
## Generate a random 100x100 matrix of 0's and 1's where 1 corresponds  
## to rock and 0 corresponds to sand. Generate the values in the  
## matrix independently, based on a given probability p that each  
## space is occupied by rock.  
##  
## A drop of liquid starts in the middle of the top layer (row 1,  
## column 50). It then moves according to the following four options,  
## where options with lower numbers have higher precedence.  
##  
## 1. If the space directly below is sand, move there.  
## 2. If the space below and to the left is sand, move there.  
## 3. If the space below and to the right is sand, move there.  
## 4. If the space directly to the right is sand, move there.  
##  
## If none of these moves can be made, the drop of liquid is stuck.  
##  
## Use simulation to calculate the average depth to which the liquid  
## drops before getting stuck, and the proportion of the time that the  
## drop reaches the bottom layer.  
  
*import* numpy *as* np  
*import* matplotlib.pyplot *as* plt  
*import* matplotlib.animation *as* animation  
*from* matplotlib *import* colors  
*import* os  
*from* sys *import* platform  
*import* shlex  
*import* pandas *as* pd  
*from* openpyxl *import* load\_workbook  
*import* seaborn *as* sns  
  
filename = "dynamic\_images.html"  
  
# change the fps to speed up or slow down the animation  
writer = animation.HTMLWriter(fps=4)  
  
# ensure matrix and data frame are not truncated  
np.set\_printoptions(threshold=np.inf)  
pd.set\_option('display.max\_columns', *None*)  
pd.set\_option('display.max\_rows', *None*)  
fig = plt.figure()  
  
# array of images  
ims = []  
  
  
*def* simulation(*p*, *N*, *nrep*, *animate*, *separate*):  
 *if separate* == *True*:  
 # empties animation array after change in density of rock in the sand.  
 *global* ims  
 ims = []  
  
 ## The total depth across the simulation replications.  
 TD = 0  
  
 ## The number of times that the bottom is reached.  
 NB = 0  
  
 ## Simulation replications.  
 *for* j *in* range(int(*nrep*)):  
 ## Randomly lay out the rocks.  
 M = (1 \* (np.random.uniform(0, 1, size=*N* \* *N*) < *p*)).reshape(*N*, *N*)  
  
 ## The initial position of the droplet.  
 r = 0  
 c = int(*N* / 2) - 1  
  
 ## Checks whether the user wishes to animate the simulations  
 *if animate* == *True*:  
 M[r, c] = 2  
  
 ## Let the droplet percolate through the rocks.  
 *while* r < *N* - 1:  
 *try*:  
 ## Always go straight down if possible.  
 *if* (M[r + 1, c] == 0):  
 r = r + 1  
 *if animate* == *True*:  
 M[r, c] = 2  
  
 ## Next try down/left.  
 *elif* ((c > 1) & (M[r + 1, c - 1] == 0)):  
 r = r + 1  
 c = c - 1  
 *if animate* == *True*:  
 M[r, c] = 2  
  
 ## Next try down/right.  
 *elif* ((c < *N*) & (M[r + 1, c + 1] == 0)):  
 r = r + 1  
 c = c + 1  
 *if animate* == *True*:  
 M[r, c] = 2  
  
 ## Next try right.  
 *elif* ((c < *N*) & (M[r, c + 1] == 0)):  
 c = c + 1  
 *if animate* == *True*:  
 M[r, c] = 2  
  
 ## We're stuck  
 *else*:  
 *break*

## We've reached the edge of the screen  
 ## Exception for attempting to index outside of bounds  
 *except* IndexError:  
 *break* ## Keep track of how often we reach the bottom.  
 *if* (r == *N* - 1):  
 NB = NB + 1  
  
 ## Keep track of the total of the final depths.  
 TD = TD + r  
  
 # Draws the final frame of each simulation for a number of realisations  
 *if animate* == *True*:  
 draw(M, j+1, *p*, *N*)  
  
 ## Draws the final frame of the last simulation for number of realisations  
 ## to allow enough time for the user to pause the animation  
 *if animate* == *True*:  
 *for* i *in* range(5):  
 draw(M, j+1, *p*, *N*)  
 create\_animation()  
 *return* NB, TD  
  
  
*def* main():  
 # Would you like to animate the simulations, please note this may take a while  
 animate = *True* # Determines whether to create new animation at each level of density or to  
 # append the new frames to the existing animation.  
 separate = *True* # Grid size  
 sizes = [10, 50, 100, 200, 400]  
 ## The number of simulation replications.  
 nrep = [100, 500, 1000, 2000, 4000]  
 # By how much is p decremented for each realisation  
 step = 0.1  
 pc = pd.DataFrame(columns=["Grid Size", "Realisations", "Crit\_Perc"])  
 k = 0  
 *for* N *in* sizes:  
 excel\_file = "percolation" + str(N) + ".xlsx"  
 print("\nThe grid size is: " + str(N) + "x" + str(N))  
 *for* i *in* nrep:  
 pc\_boolean = *True* i = round(i)  
 print("\nRunning simulation with " + str(i) + " realisations")  
 ## The density of rocks in the sand.  
 p = 1  
 df = pd.DataFrame(columns=["Density", "Number\_Bottom",  
 "Frequency\_Reach\_Bottom", "Total\_Depth",  
 "Average\_Depth"])  
 j = 0  
 *while* p > 0:  
 p = round(p, 2)  
 sim = simulation(p, N, i, animate, separate)  
  
 NB = sim[0]  
 TD = sim[1]  
  
 ## The estimated probability that we reach the bottom.  
 ## Frequency with which the bottom is reached at each probability  
 NBprob = NB / i  
  
 ## The average depth that is reached.  
 TDavg = TD / i  
  
 df.loc[j] = [p, NB, NBprob, TD, TDavg]  
 j += 1  
  
 critperc = p  
 *if* pc\_boolean *and* NB >= 1:  
 critperc = p  
 pc\_boolean = *False* p -= step  
  
 *if* nrep.index(i) == 0:  
 *with* pd.ExcelWriter(excel\_file) *as* writer:  
 writer.save()  
 print("Created the excel file " + str(excel\_file))  
 *try*:  
 *with* pd.ExcelWriter(excel\_file, engine='openpyxl', mode='a') \  
 *as* writer:  
 writer.book = load\_workbook(excel\_file)  
 sheetName = str(i) + 'realisations' + str(N) + "by" + str(N)  
 df.to\_excel(writer, sheetName, index=*False*, header=*True*)  
 *try*:  
 writer.book.remove(writer.book['Sheet1'])  
 *except* KeyError:  
 *pass* writer.save()  
 *except* PermissionError:  
 print("Please ensure the file '" + str(excel\_file) +  
 "' is not open in another program")  
  
 print(df)  
 df.plot(x='Density', y='Frequency\_Reach\_Bottom', kind='scatter')  
 plt.title("Water Percolation - " + str(i) + " Realisations - " +  
 str(N) + "x" + str(N) + " Grid")  
 plt.xlim(0, 1)  
 plt.ylim(0, 1)  
 plt.xlabel('Density of rocks in the sand')  
 plt.ylabel('Expectation that the bottom is reached')  
 plt.xticks(np.arange(0, 1+step, step=0.1))  
 plt.yticks(np.arange(0, 1.1, step=0.1))  
 plt.show()  
 pc.loc[k] = [N, i, critperc]  
 k += 1  
 # method removes trailing zeros from data frame  
 size = pd.Series(pc['Grid Size'])  
 mask = pd.to\_numeric(size).notnull()  
 size.loc[mask] = size.loc[mask].astype(np.int64)  
 pc['Grid Size'] = size  
 print(pc)  
 # Use the 'hue' argument to provide a factor variable  
 sns.scatterplot(data=pc, x="Crit\_Perc", y="Realisations", hue='Grid Size',  
 legend='full', markers=["o", "x", "1", "s", "d"],  
 palette=['green', 'orange', 'purple', 'dodgerblue', 'red'])  
 plt.title("Critical Percolation against Number of Realisations")  
 plt.xlim(0, 1)  
 plt.ylim(0, i)  
 plt.xlabel('Critical Percolation')  
 plt.ylabel('Number of Realisations')  
 plt.legend(loc='upper left')  
 plt.xticks(np.arange(0, 1+0.1, step=0.1))  
 plt.yticks(np.arange(0, i+0.1, step=max(nrep)/10))  
 plt.show()  
 *with* pd.ExcelWriter("percolationpcall.xlsx") *as* writer:  
 writer.save()  
 print("Created the excel file 'percolationpcall.xlsx'")  
 *try*:  
 *with* pd.ExcelWriter("percolationpcall.xlsx", engine='openpyxl', mode='a')\  
 *as* writer:  
 writer.book = load\_workbook("percolationpcall.xlsx")  
 sheetName = "Critical Percolations All"  
 pc.to\_excel(writer, sheetName, index=*False*, header=*True*)  
 *try*:  
 writer.book.remove(writer.book['Sheet1'])  
 *except* KeyError:  
 *pass* writer.save()  
 *except* PermissionError:  
 print("Please ensure the file 'percolationpcall.xlsx' "  
 "is not open in another program")  
  
  
*def* draw(*data*, *j*, *p*, *N*):  
 # Colours for visualization: gold for sand, grey for rock and blue for water.  
 # module is poorly coded so colours and boundary array each need one more  
 # element than there are colours in the animation and numbers in the matrix.  
 colors\_list = ['gold', 'grey', 'black', 'blue']  
 # observe that the colour black appears nowhere in the animation and neither  
 # does the number three appear anywhere within the matrix,  
 # i do not know why the developer indexes the arrays incorrectly.  
 # create discrete colormap  
 cmap = colors.ListedColormap(colors\_list)  
 bounds = [0, 1, 2, 3]  
 norm = colors.BoundaryNorm(bounds, cmap.N)  
 *global* ims  
 plt.xticks([])  
 plt.yticks([])  
 plt.title("Density of mud in the forest: " + str(*p*)  
 + "\n Number of realisations: " + str(*j*)  
 + "\nSize of matrix: " + str(*N*) + "x" + str(*N*))  
 im = plt.imshow(*data*, cmap=cmap, norm=norm, animated=*True*)  
 ims.append([im])  
  
  
*def* create\_animation():  
 print("\nCreating animation, please wait...")  
 ani = animation.ArtistAnimation(fig, ims, blit=*True*)  
 ani.save(filename, writer=writer)  
 # OS X  
 *if* platform == "darwin":  
 *try*:  
 os.system("open " + shlex.quote(filename))  
 *except*:  
 print("You will need to manually open the html file")  
 # Windows  
 *elif* platform == "win32":  
 *try*:  
 os.system("start " + filename)  
 *except*:  
 print("You will need to manually open the html file")  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 main()

**Here is the code relating to the simulation of the forest fire percolation system.**

# -\*- coding: utf-8 -\*-  
"""  
Created on Tue Mar 0 00:05:58 2020  
  
  
"""  
## Simulate a forest fire percolating outwards from the  
# centre of a mass of trees and mud.  
##  
## Generate a random NxM matrix of 0's and 1's where 1 corresponds  
## to mud and 0 corresponds to trees. Generate the values in the  
## matrix independently, based on a given probability p that each  
## space is occupied by mud.  
##  
## A drop of liquid starts at the centre of the matrix grid (row N//2,  
## column M//2). It then moves according to the following system.  
##  
##  
## 1. Consider an individual cell and the 8 neighbouring cell that surround it  
## 2. Starting from the top left neighbouring cell travel clockwise  
## 3. For any cells not already on fire that contain trees set these alight  
## 4. Store the location of an cells containing fire and repeat  
## the above process for the outermost cells that are alight.  
##  
## If none of these moves can be made, the spread of the fire is stopped.  
##  
## Use simulation to calculate the average distance over which the fire  
## spreads before getting stuck, and the proportion of the time that the  
## fire reaches the edge of the forest.  
  
*import* numpy *as* np  
*import* matplotlib.pyplot *as* plt  
*import* matplotlib.animation *as* animation  
*from* matplotlib *import* colors  
*import* os  
*from* sys *import* platform  
*import* shlex  
*import* pandas *as* pd  
*from* openpyxl *import* load\_workbook  
*import* seaborn *as* sns  
  
filename = "dynamic\_images.html"  
  
# change the fps to speed up or slow down the animation  
writer = animation.HTMLWriter(fps=12)  
# ensure matrix and data frame are not truncated  
np.set\_printoptions(threshold=np.inf)  
pd.set\_option('display.max\_columns', *None*)  
pd.set\_option('display.max\_rows', *None*)  
fig = plt.figure()  
  
# array of images  
ims = []  
  
*def* simulation(*p*, *ny*, *nx*, *nrep*, *animate*, *separate*):  
 *if separate* == *True*:  
 # empties animation array after change in density of mud in the forest.  
 *global* ims  
 ims = []  
  
 ## The total distance across the simulation replications.  
 #TD = 0  
  
 ## The number of times that the edge is reached.  
 NB = 0  
  
 # Displacements from a cell to its eight nearest neighbours  
 neighbourhood = ((-1,1), (0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0))  
 ## Simulation replications.  
 *for* j *in* range(int(*nrep*)):  
 # Initialize the forest grid.  
 X = np.random.choice([0, 1], size=*ny* \* *nx*, p=[1-*p*, *p*]).reshape(*ny*, *nx*)  
 # Starting position of fire at centre of grid  
 X[(*ny*//2), (*nx*//2)] = 2  
  
 """Iterate the forest according to the forest-fire rules."""  
  
 iy, ix = *ny* // 2, *nx* // 2  
 positions = []  
 positions.append([iy, ix])  
 boolean = *True  
 if animate* == *True*:  
 draw(X, j+1, *p*, *ny*, *nx*)  
 *while True*:  
 *for* i *in* positions:  
 *for* dx, dy *in* neighbourhood:  
 *try*:  
 ## Keep track of how often we reach the edges.  
 *if* boolean:  
 *if* ((i[0] + dy == *ny* - 1) *or* (i[1] + dx == *nx* - 1)) \  
 *and* X[i[0] + dy, i[1] + dx] == 0:  
 X[i[0] + dy, i[1] + dx] = 2  
 NB += 1  
 boolean = *False  
 break  
 else*:  
 *if* X[i[0] + dy, i[1] + dx] == 0:  
 X[i[0] + dy, i[1] + dx] = 2  
 #TD = TD + 1  
 positions.append([i[0] + dy, i[1] + dx])  
 *if animate* == *True*:  
 draw(X, j+1, *p*, *ny*, *nx*)  
 *continue  
  
 except* IndexError:  
 *pass  
 if animate* == *True*:  
 *for* i *in* range(5):  
 draw(X, j+1, *p*, *ny*, *nx*)  
 *break  
 return* NB  
  
  
*def* main():  
 # Would you like to animate the simulations, please note this may take a while  
 animate = *False* # Determines whether to create new animation at each level of density or to  
 # append the new frames to the existing animation.  
 separate = *True* # Forest size (number of cells in x and y directions).  
 sizes = [10, 50, 100, 200, 400]  
 ## The number of simulation replications.  
 nrep = [100, 500, 1000, 2000, 4000]  
 # By how much is p decremented for each realisation  
 step = 0.01  
 pc = pd.DataFrame(columns=["Grid Size", "Realisations", "Crit\_Perc"])  
 k = 0  
 *for* n *in* sizes:  
 n = round(n)  
 ny, nx = n, n  
 excel\_file = "forest" + str(n) + ".xlsx"  
 print("\nThe grid size is: " + str(ny) + "x" + str(nx))  
 *for* i *in* nrep:  
 pc\_boolean = *True* i = round(i)  
 print("\nRunning simulation with " + str(i) + " realisations")  
 # The density of mud in the forest not occupied by trees  
 p = 1  
 df = pd.DataFrame(columns=["Density", "Number\_Edge",  
 "Frequency\_Reach\_Edge"])  
 j = 0  
 *while* p > 0:  
 p -= step  
 p = round(p, 2)  
 sim = simulation(p, ny, nx, i, animate, separate)  
  
 NB = sim  
 #TD = sim[1]  
  
 ## The estimated probability that we reach the edge.  
 ## Frequency with which the edge is reached at each probability  
 NBprob = NB / i  
  
 ## The average distance that is reached.  
 #TDavg = TD / i  
  
 df.loc[j] = [p, NB, NBprob]  
 j += 1  
  
 ## Draws the final frame of each simulation multiple times  
 ## to allow enough time for the user to pause the animation  
 *if* animate == *True*:  
 create\_animation()  
 *if* pc\_boolean *and* NB >= 1:  
 critperc = p  
 pc\_boolean = *False  
 if* nrep.index(i) == 0:  
 *with* pd.ExcelWriter(excel\_file) *as* writer:  
 writer.save()  
 print("Created the excel file " + str(excel\_file))  
 *try*:  
 *with* pd.ExcelWriter(excel\_file, engine='openpyxl', mode='a') \  
 *as* writer:  
 writer.book = load\_workbook(excel\_file)  
 sheetName = str(i) + 'realisations' + str(n) + "by" + str(n)  
 df.to\_excel(writer, sheetName, index=*False*, header=*True*)  
 *try*:  
 writer.book.remove(writer.book['Sheet1'])  
 *except* KeyError:  
 *pass* writer.save()  
 *except* PermissionError:  
 print("Please ensure the file '" + str(excel\_file) +  
 "' is not open in another program")  
  
 print(df)  
 df.plot(x='Density', y='Frequency\_Reach\_Edge', kind='scatter')  
 plt.title("Forest Fire Percolation - " + str(i) + " Realisations - "  
 + str(n) + "x" + str(n) + " Grid")  
 plt.xlim(0, 1)  
 plt.ylim(0, 1)  
 plt.xlabel('Density of mud in the forest')  
 plt.ylabel('Expectation that the edge is reached')  
 plt.xticks(np.arange(0, 1+step, step=0.1))  
 plt.yticks(np.arange(0, 1.1, step=0.1))  
 plt.show()  
 pc.loc[k] = [n, i, critperc]  
 k += 1  
 # method removes trailing zeros from data frame  
 size = pd.Series(pc['Grid Size'])  
 mask = pd.to\_numeric(size).notnull()  
 size.loc[mask] = size.loc[mask].astype(np.int64)  
 pc['Grid Size'] = size  
 print(pc)  
 # Use the 'hue' argument to provide a factor variable  
 sns.scatterplot(data=pc, x="Crit\_Perc", y="Realisations", hue='Grid Size',  
 legend='full', markers=["o", "x", "1", "s", "d"],  
 palette=['Burlywood', 'DarkSlateGray',  
 'SaddleBrown', 'Teal', 'Olive'])  
 plt.title("Critical Percolation against Number of Realisations")  
 plt.xlim(0, 1)  
 plt.ylim(0, i)  
 plt.xlabel('Critical Percolation')  
 plt.ylabel('Number of Realisations')  
 plt.legend(loc='upper left')  
 plt.xticks(np.arange(0, 1+0.1, step=0.1))  
 plt.yticks(np.arange(0, i+0.1, step=max(nrep)/10))  
 plt.show()  
 *with* pd.ExcelWriter("forestpcall.xlsx") *as* writer:  
 writer.save()  
 print("Created the excel file 'forestpcall.xlsx'")  
  
 *try*:  
 *with* pd.ExcelWriter("forestpcall.xlsx", engine='openpyxl', mode='a') \  
 *as* writer:  
 writer.book = load\_workbook("forestpcall.xlsx")  
 sheetName = "Critical Percolations All"  
 pc.to\_excel(writer, sheetName, index=*False*, header=*True*)  
 *try*:  
 writer.book.remove(writer.book['Sheet1'])  
 *except* KeyError:  
 *pass* writer.save()  
 *except* PermissionError:  
 print("Please ensure the file 'forestpcall.xlsx' "  
 "is not open in another program")  
  
  
*def* draw(*data*, *j*, *p*, *ny*, *nx*):  
 # Colours for visualization: green for trees, brown for mud, orange for fire.  
 # module is poorly coded so colours and boundary array each need one more  
 # element than there are colours in the animation and numbers in the matrix.  
 colors\_list = ['green', 'brown', 'black', 'orange']  
 # observe that the colour black appears nowhere in the animation and neither  
 # does the number three appear anywhere within the matrix,  
 # i do not know why the developer indexes the arrays incorrectly.  
 # create discrete colormap  
 cmap = colors.ListedColormap(colors\_list)  
 bounds = [0, 1, 2, 3]  
 norm = colors.BoundaryNorm(bounds, cmap.N)  
 *global* ims  
 plt.xticks([])  
 plt.yticks([])  
 plt.title("Density of mud in the forest: " + str(*p*)  
 + "\n Number of realisations: " + str(*j*)  
 + "\nSize of matrix: " + str(*ny*) + "x" + str(*nx*))  
 im = plt.imshow(*data*, cmap=cmap, norm=norm, animated=*True*)  
 ims.append([im])  
  
  
*def* create\_animation():  
 print("\nCreating animation, please wait...")  
 ani = animation.ArtistAnimation(fig, ims, blit=*True*)  
 ani.save(filename, writer=writer)  
 # OS X  
 *if* platform == "darwin":  
 *try*:  
 os.system("open " + shlex.quote(filename))  
 *except*:  
 print("You will need to manually open the html file")  
 # Windows  
 *elif* platform == "win32":  
 *try*:  
 os.system("start " + filename)  
 *except*:  
 print("You will need to manually open the html file")  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 main()