Application of Sand Pile Modelling Techniques to Traffic Simulation

COMP90072: The Art of Scientific Computation

Peter Ljubisic
993443

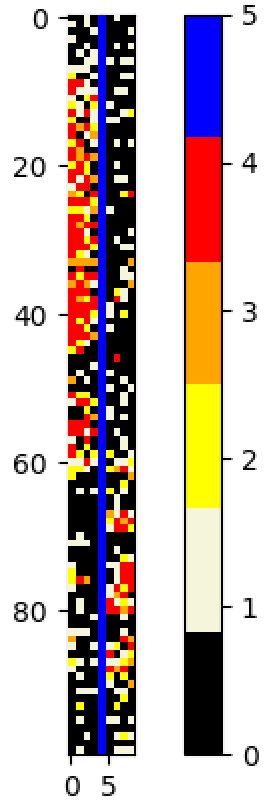


Table of Contents

0: Introduction

Within today's environment, the ability to problem-solve using computational methods is an essential skill in a wide variety of fields, ranging from the likes of simple data analysis of spreadsheets to modelling epidemics. As such, the complexity of programming tasks can range from being relatively simple and straightforward to being a delicate labyrinth of tedious bug fixing. In this report, the simple sand pile model will be explored and extended to serving as a suitable method of modelling traffic along a highway.

The sand pile model is quite simple: a grid of arbitrary size is created and grains of sand are randomly placed on the grid one at a time. The catch is that if the number of sand grains in one entry of the grid becomes too large, it will "topple" and distribute some sand to its neighbours. The goal of modelling this simple system is to observe an "avalanche" effect, where if there are enough grains placed on the grid, then a chain reaction of topples would occur that would propagate across the grid. The goal for recreating this computational model is to reproduce and observe this phenomenon, and to apply it to real world problems. Naturally the larger the grid becomes, the more effective it would be at showcasing the properties of complex systems, but at the cost of increasing the amount of time needed to compute the task. The advancement of developed versions of this model will be outlined in this report.

In successfully modelling the basic sand pile system, it would function as a prototype for the grand focus of this report: the stochastic modelling of traffic along a highway. Rather than dealing with sand grains, it would be cars that are imposed on the grid. Unlike the sand pile system however, the cars can evolve in position without a topple actually occurring, and would make extensive use of probabilities to determine how they evolve based on traffic densities in regions around them. The goal would be to observe avalanche effects of traffic congestion along a highway and how it would propagate under time evolution.

1: Sand Pile Model

1.1: Theoretical Foundations of Sand Pile Modelling

For basic sand pile modelling, extensive use of the Bak-Tang-Wiesenfeld (BTW) Model is employed. The BTW model works by simulating the presence of sand grains by dividing an area into discrete positions that can be occupied by a sand grain. This grid is of arbitrary size (N x M) and can be visualised in Figure 1.1.0 below. However, as N and M approach infinity the grid becomes better at approximating a continuous distribution of position. One at a time, each sand grain is randomly distributed on the grid, and each entry of the grid can have numbers 0, 1, 2, 3 or 4 which correspond to how many sand grains are in that entry. When four sand grains appear on one tile, it collapses and distributes one sand grain to each neighbouring non-diagonal tile. This phenomenon is referred to as a “topple” and is visualised in Figure 1.1.1 below. If this topple occurs at the edge of the grid, then some of the toppled sand grains will fall out of bounds and be removed from the system. When one of the neighbouring entries of a topple site has a value of three, it will take a value of four and then also become a topple site. This chain reaction is referred to as an “avalanche” and the primary goal of the sand pile model is to produce and observe an avalanche. One may assume it would be beneficial to make the grid infinitely large to best approximate a continuum of positions which would reflect reality, but obviously an infinite number of positions would require infinite memory on a computer to store so this would be nonsense. A more fundamental issue is that the probability of a sand grain randomly spawning on top of another one reduces to zero from the infinite choices available regardless of how many have spawned, hence the avalanche effect effectively becomes practically irreproducible. Therefore, this model requires the grid size to be suitably large but finite.

1	1	3	0	4
0	0	2	2	3
2	1	0	3	0

Figure 1.1.0: A 3x5 grid of a sand pile model: Numbers ranging from 0 to 5 correspond to how many sand grains are in that position on the grid.

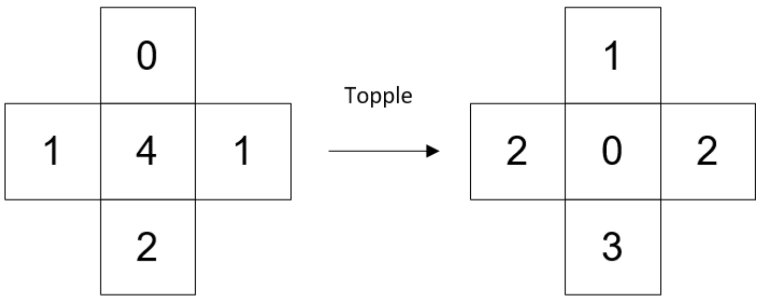


Figure 1.1.1: A topple occurring from the entry in the centre into its four non-diagonal neighbouring entries.

1.2: Computational Methods of BTW Model Version 1

When beginning the development of the BTW version of the sand pile program, it was important to make sure that the necessary packages were installed into the computer system for access. PyCharm was used as the interface tool for all development in Python, and the pip module had to be set as one of the computer's available pathways so that the terminal interface could readily install all required packages to develop the model. The packages numpy (for creating arrays and matrices) and matplotlib (for graphing the matrix representation of sand drops) were promptly installed since both were vital to the development of the program.

The first version of the code is shown as a flowchart in terms of function to the right. Note that all versions of sand pile modelling use simple square matrices, for these serve as prototypes for the traffic application and it seemed more convenient to reduce the number of inputs necessary for bugfixing. Moving on, nested loops are used to iterate over each entry in the matrix (going from top row to bottom row, leftmost entry per row to rightmost entry) and checking to see if the conditions for a sand topple have been met, which is displayed in Figure 1.2.1 below. The idea was to get the program working before optimisation would take place, and intuitively nest loops are a rather simple method for getting the job done even if inefficient. When all topples for that time step have occurred, a graph of the matrix is displayed which can be closed by the user to initiate the next time step. There are two noticeable problems with this version of the program however:

Firstly, the topple loop is biased. Consider Figure 1.2.1 again where a drop of sand is applied (resulting in the 4 entry) in the matrix:

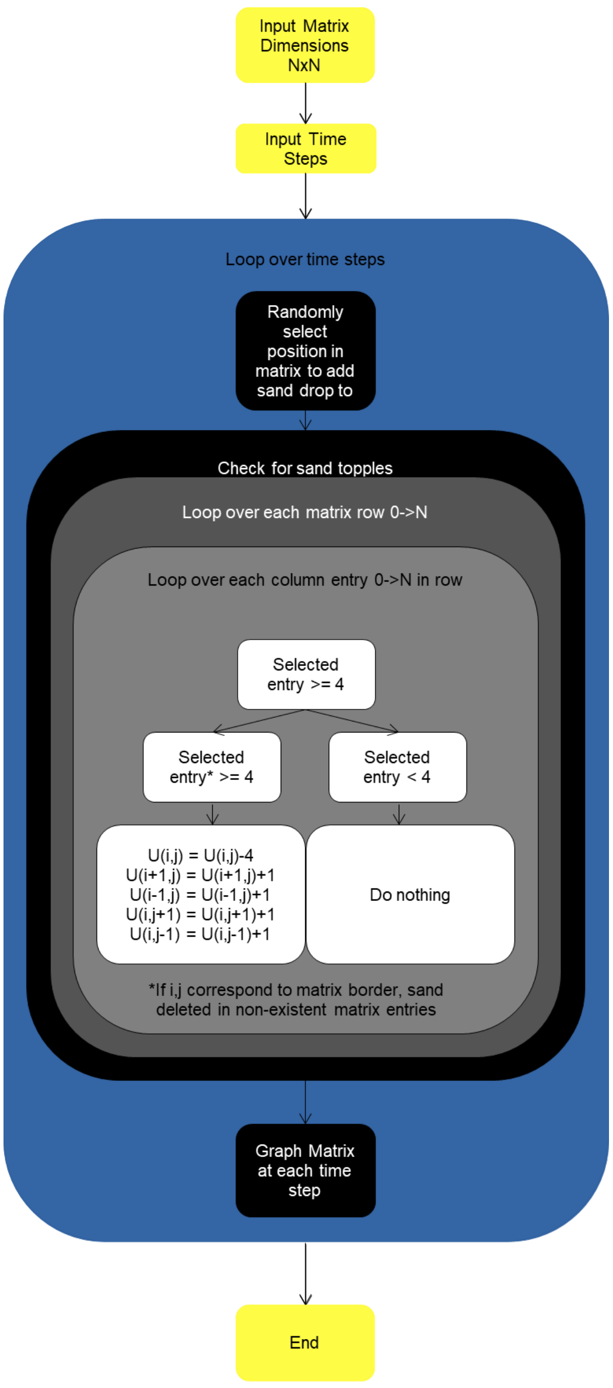
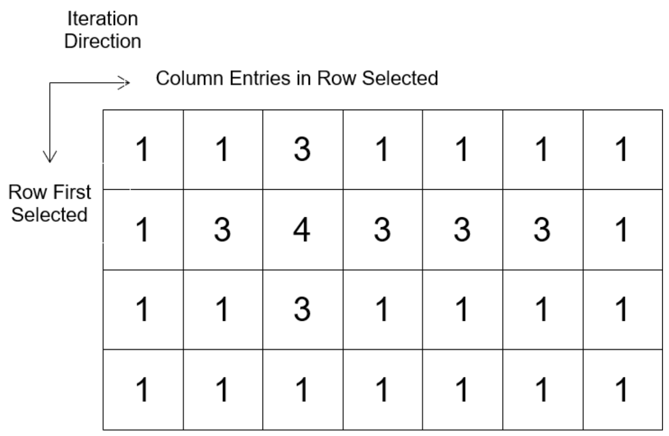

Input Matrix Dimensions NxN
Input Time Steps
Loop over time steps
Randomly select position in matrix to add sand drop to
Check for sand topples
Loop over each matrix row 0->N
Loop over each column entry 0->N in row
Selected entry >= 4
Selected entry* >= 4
Selected entry < 4
U(i,j) = U(i,j)-4
U(i+1,j) = U(i+1,j)+1
U(i-1,j) = U(i-1,j)+1
U(i,j+1) = U(i,j+1)+1
U(i,j-1) = U(i,j-1)+1
Do nothing
*If i,j correspond to matrix border, sand deleted in non-existent matrix entries
Graph Matrix at each time step
End

Figure 1.2.0: A flowchart of Version 1 of the BTW model code for sand piling.



1	1	3	1	1	1	1
1	3	4	3	3	3	1
1	1	3	1	1	1	1
1	1	1	1	1	1	1

Figure 1.2.1: A diagram highlighting how the program evaluates entry values of the matrix grid.

Along the second row, the topple happens in the third column, so the entries above and below as well as to the left and right gain an increase in value by one and reach a value of four and become ready to topple. But because of the iteration direction, a chain of topples occur as the program iterates to the right, and the entry below the original topple site will topple as well. But the entries above and to the left of the original topple site do not topple in the same timestep despite being ready to, because those entries were already iterated upon. Hence there is an inherent bias of direction in toppling which can cause confusing results.

Finally, it is impractical to create a new graph every timestep, and thus it is better to create a video of the evolution of the matrix with respect to each timestep. This first iteration of the program was primitive in the sense that the graphs could not evolve on their own without user input, so one tediously had to close each graph window to progress to the next time step.

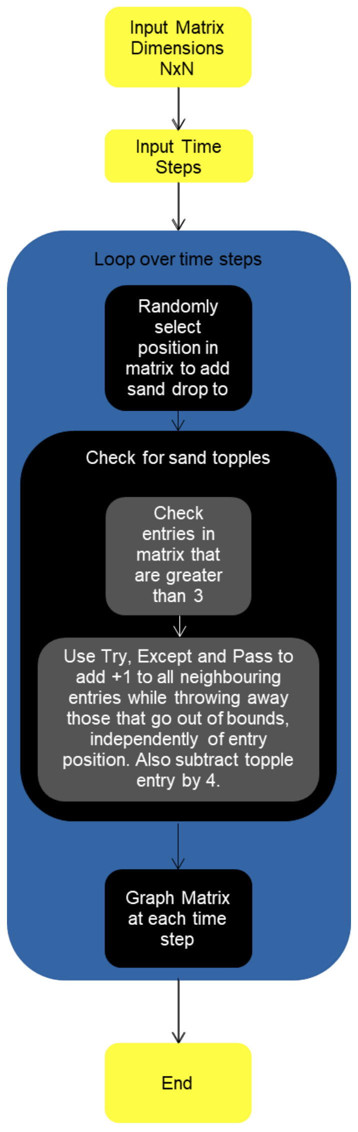
1.3: Computational Methods of BTW Model Version 2

Because of the issues explained above, and another concern which will be covered shortly, Version 1 was a good start but fundamentally flawed. Heavy revisions had to be made to improve efficiency and reduce possibility of error. The flowchart of Version 2 is displayed to the right. It is immediately clear that it is significantly more efficient.

The process by which Version 2 checks for topple entries in the matrix greatly differs from Version 1. Rather than iterating over each entry in the matrix, the program uses the argwhere tool from the numpy library to simply check for any the entries in the matrix that have a value greater than three, and topples them all simultaneously. Not only is it greatly efficient, but it is not subject to directional bias from iterating over entries as it merely locates the entries first and then topples them irrespective of whether it forms new topple sites or not. This makes the avalanche effects of the system very clear to the observer.

Version 2 is also able to update its visualisation of the matrix on its own. Version 1 required the user to close the current graph window to progress to the next timestep. If a large matrix is evaluated, one would need many timesteps to witness even a single topple, rendering that method horribly tedious. This method updates the matrix visualisation at a speed that lets the user appreciate the eventual avalanches the model produces.

One aspect of Version 2 that is quite significant is how it handles topple sites at and away from the borders of the matrix. Version 1 used multiple nested if loops to determine which directions a sand grain could be distributed to from a topple. At the bottom row of the matrix for instance, one cannot add a sand grain to the row below because no such row would exist and an error would arise, so Version 1 used if statements to prevent this from occurring. Version 1 did this for all border sides, and the corners required their own if statements too. Version 2 rather uses the commands Try, Except and Pass to add one sand grain to all four directions regardless of whether the topple site was at the border or not. If an error occurs, then that operation is skipped and the system carries out its procedure in the other directions as if there was no issue. This is effective because no if statements are required to specialise sand distributions from topples, and means the computer does less work executing the code. This causes problems however that will be covered in Section 2.



Input Matrix Dimensions NxN
Input Time Steps
Loop over time steps
Randomly select position in matrix to add sand drop to
Check for sand topples
Check entries in matrix that are greater than 3
Use Try, Except and Pass to add +1 to all neighbouring entries while throwing away those that go out of bounds, independently of entry position. Also subtract topple entry by 4.
Graph Matrix at each time step
End

Figure 1.3.0: A flowchart of Version 2 of the BTW model code for sand piling.

Finally, when graphing the matrix every timestep and placing it into an animation, two approaches were used. The first was to display the next frame of the animation as soon as it was generated, thus the animation would continue as the code would. There was a flaw in that the animation window would frequently need to be closed to refresh the speed at which the next frame would be displayed, otherwise eventually each frame would require at least second to be displayed. This was rectified by using the second approach: playing the animation after all the code had finished running. The frames of the animation became elements of an initially empty array that would be appended with each timestep. Doing this meant the user could wait a while before seeing the animation, but that in turn it would be displayed at a consistent, fixed rate. This approach was favoured over the initial method for its ability to continue without requiring unnecessary and tedious user input.

1.4: Computational Methods of Gradient Model

Now that the BTW model is smoothly running, it is time to transition to a more sophisticated, nuanced program for sand pile modelling. The previous BTW model is limited in that if an entry in the matrix is ready to topple, it will equally distribute sand to one of its neighbours, regardless of the amount they already have. For instance, consider Figure 1.4.1 below:

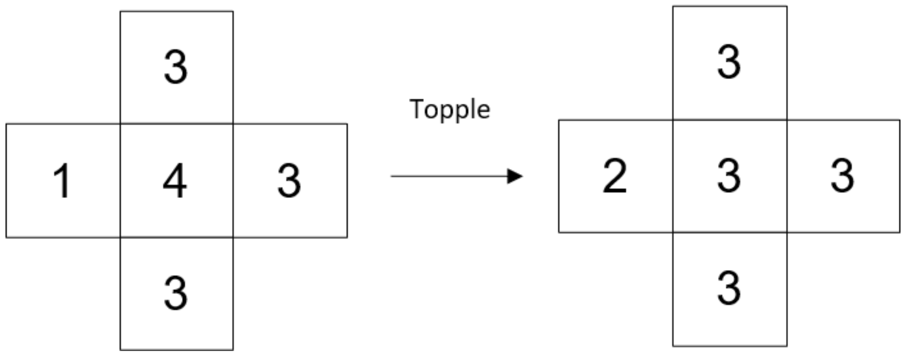


Figure 1.4.1: A demonstration of the new gradient toppling method.

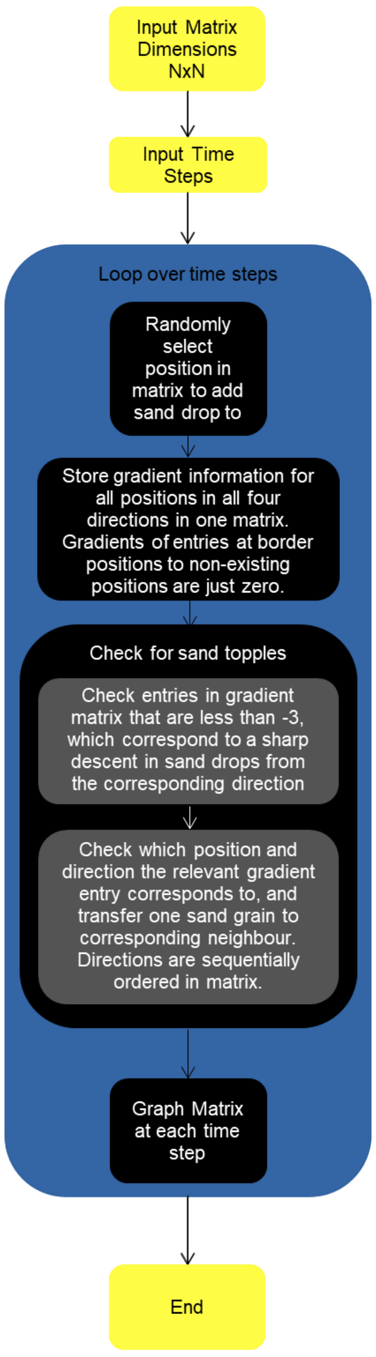

Input Matrix Dimensions NxN
Input Time Steps
Loop over time steps
Randomly select position in matrix to add sand drop to
Store gradient information for all positions in all four directions in one matrix. Gradients of entries at border positions to non-existing positions are just zero.
Check for sand topples
Check entries in gradient matrix that are less than -3, which correspond to a sharp descent in sand drops from the corresponding direction
Check which position and direction the relevant gradient entry corresponds to, and transfer one sand grain to corresponding neighbour. Directions are sequentially ordered in matrix.
Graph Matrix at each time step
End

Figure 1.4.0: A flowchart of the Gradient version of the code for sand piling.

When it topples, the previous two versions would distribute one piece of sand equally to its neighbours. But is this a realistic approach to the situation? Of course not, and thus a new version of the model is developed that takes into account the amount of sand the neighbouring entries of a topple site contain in order to determine where best to distribute sand from a topple.

The gradient version operates on similar principles to Version 2, only differing in how it distinguishes topple sites from non-topple sites and how much sand an entry can lose from toppling.

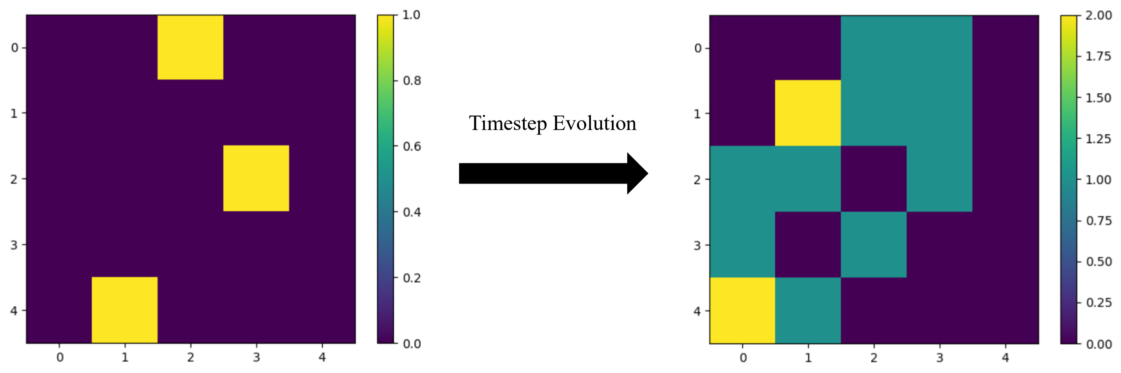
In each timestep, the difference between a neighbouring entry and the selected base entry is stored as a gradient value. Because the BTW model operates in four directions, this is done four times per entry with a differing neighbouring entry each time. This process is looped over all rows and columns, and the resulting gradient matrix has the same number of rows but four times as many columns. Taking Figure 1.1.0 as an example, a 3x5 matrix would produce a 3x20 gradient matrix. Each row of the gradient matrix corresponds to the same row as in the regular matrix, and every four consecutive entries per row in the gradient matrix correspond to all four gradients of one entry in the normal matrix: gradients in sequential order of directions up, down, right and left. Taking the same example, entry (2,1) in the normal matrix would have its upwards, downwards, right and left gradients be at positions (2,1), (2,2), (2,3) and (2,4) respectively in the gradient matrix. When one gradient entry is less than -3 (i.e., entry has a direction of steep descent), it means the site is ready to topple in at least one direction. The topple position is found by deriving the row of the topple entry in the gradient matrix, and the column is found by dividing the column of the entry in the gradient matrix by 4 and rounding down the result. The remainder of the division operation determines the direction that a sand grain is transported to. For instance, if the relevant gradient entry is at position (2,2) in the gradient matrix, then the position of the topple is at (2, floor(2/4) = 0) in the normal matrix, and the remainder of 2/4 is 0 so the sand grain is transferred to the entry above it. All in all, it is quite simple, but sophisticated compared to the earlier iterations.

Overall, not much else needs to be said about this program because it functions so similarly to Version 2. However, it is of interest to outline how each version displays the simulation.

1.5: Results, Output and Comparison

In this section, the focus is on the visualisation of the sand pile model displayed by the code.

In version 1, the visual output resembled that of Figure 1.5.0:



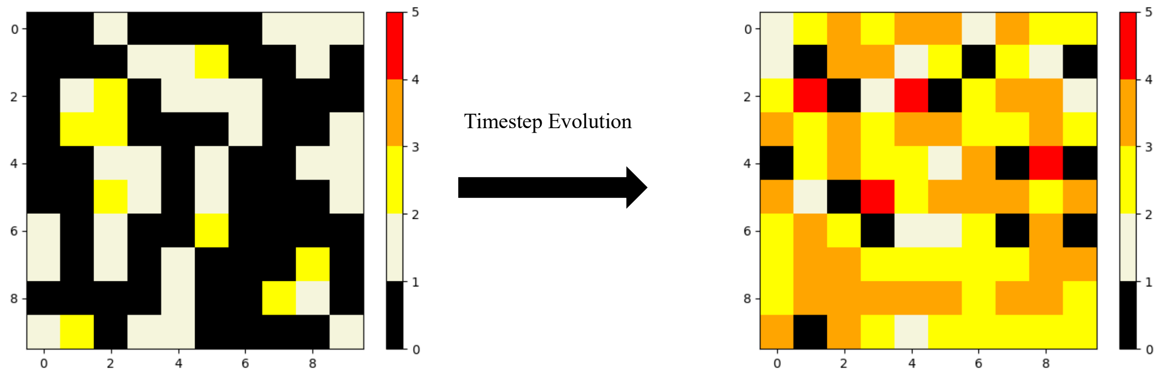
A 5x5 sand pile grid at the 3rd timestep

Figure 1.5.0

A 5x5 sand pile grid at the 14th timestep

At first glance, this version makes a decent output of the evolution of the sand pile matrix with time evolution. There were numerous issues however. Firstly, notice that the colour gradient indicated to the left of each graph changes with respect to the highest amount of grains in one entry. At timestep four, yellow corresponds to entries of value one while at timestep fourteen, yellow corresponds to entries of value two. This makes any attempt to analyse the development of topples cumbersome as colours can spontaneously change. The default settings for the attribution of colours to numbers are not ideal either. There can be ambiguity as to whether the teal colour in the second graph should correspond to lower or higher values than the yellow colour without looking at the colour scales. Finally and most egregiously, Version 1 is infamous for not animating the matrix but rather presenting a static figure and requiring the user to close the figure window to allow the code to generate the next frame and so on. This would already be tedious and annoying for small grids with small timesteps but for larger cases it becomes too impractical for use. Version 1 was very much made to be a prototype for later versions of the sand pile model.

Moving to Version 2, there are a number of developments as shown in Figure 1.5.1:



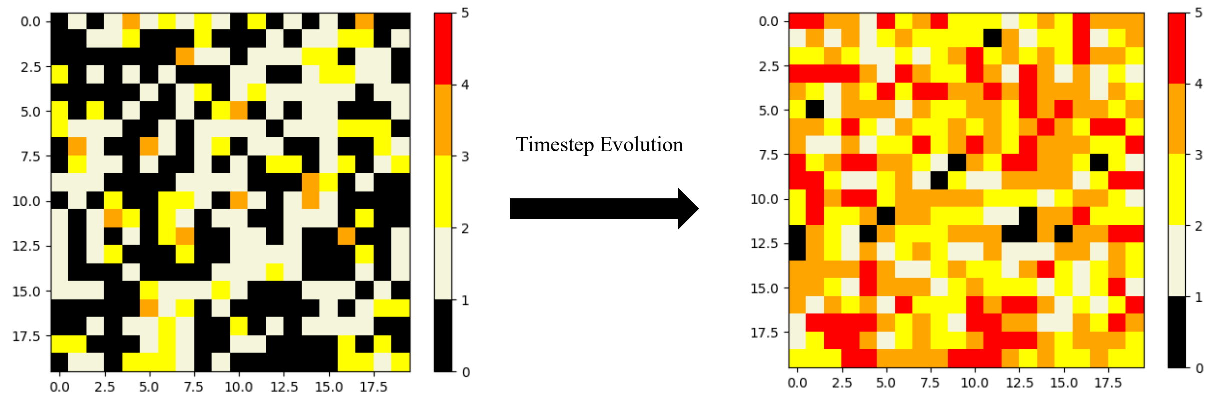
A 10x10 sand pile grid at the 50th timestep

Figure 1.5.1

A 10x10 sand pile grid at the 250th timestep

Firstly, there is the fixed colour palette. It no longer operates on a gradient of colours but rather on discrete colours that clearly correlate intensity to higher numbers. These operate on discrete and fixed values of the matrix, so yellow will always correspond to an entry value of two regardless of the largest number the matrix contains. It is very easy to tell which regions feature greater sand grain density because of this colour palette. Finally, the program actually creates an animation and does not require any input from the user except for establishing the conditions for the model itself (matrix dimension and timesteps). As established before, there were two approaches used to create an animation: the first involved playing the animation simultaneously as the code was running and the second approach was to wait until the code finished before playing the animation. Ultimately the second option was superior. With the former, the issue was that the delay between each frame would gradually increase as the timesteps progressed further, for topples were more likely to occur and likely induce further topples as a result of increased sand grains dropped into the system. Even without this factor, the animation window would frequently have to be closed since the method involved caused a noticeable and gradual slowdown of time staying in each frame of the animation which was fairly annoying and slightly reminiscent of the issues with Version 1. Furthermore, it meant that stopping the animation was tedious since one could not close the window to stop it because it would reappear until the program finished looping over each timestep. Because of these reasons, it became faster to instead make the animation appear after the code had already finished.

Moving to the gradient version which is displayed in Figure 1.5.2:



A 20x20 sand pile grid at the 300th timestep

Figure 1.5.2

A 20x20 sand pile grid at the 1000th timestep

It pretty much operates on the same exact animation principles as Version 2. Where it differs is in the underlying logic for the processes that gets translated to the animation. Version 2 operates on a basic BTW structure where a topple forces the entry to distribute an element to each of its neighbours. Version 3 instead operates by using gradients to determine where to distribute sand during a topple: meaning if there are neighbours of similar entry value to it, it won't distribute sand to those neighbours but instead to ones deficient in entry number size. The gradient version uses a gradient matrix to calculate where these sand distributions go for all topple sites simultaneously but as a result the code may take longer to perform. The advantage is a more realistic display of the avalanche effect of induced topples compared to Version 2.

Thus when comparing versions, it is clear the gradient version is the best iteration for sand pile modelling from a qualitative perspective of results output, and ties with Version 2 but is vastly superior to Version 1 for information display. It may be slower than Version 2 but makes up for it with a more realistic and appropriate approach to sand pile modelling. Either case is suitable for observing avalanche effects.

Speaking of avalanche effects, it is defined as when a topple induces further topples as a direct consequence. It has been discussed mechanically as to what that means, but visually it corresponds to red entries spawning from the toppling of other red entries; sparking a continuous chain reaction. An avalanche is more likely to occur in Version 2 than for the gradient version as a result of changes in the logic of how a topple can occur, but again the gradient model provides a more realistic depiction of such phenomena. Both versions make it very simple for an observer to track the development of an avalanche unlike Version 1 courtesy of abandoning nested for loops (as discussed around the introduction of Figure 1.2.1).

Wrapping up this section, the principles developed in these basic sand pile models are applied to the rather advanced application of traffic modelling along highways, of which the process of documenting its development and details is quite exhausting given its implementation challenges.

2: Application – Traffic Modelling

2.1: Motivations and Inspirations

When one thinks of applications for sand-pile modelling, one would not immediately jump to traffic modelling and instead perhaps model bushfires instead. As for me however, the dispersive properties exhibited in the sand-pile model for high densities of sand in one region show great potential for modelling traffic flow. It can be said that fundamentally, traffic is probabilistic. Sometimes drivers arrive in bulk at once, and other times one can actually hit the speed limit. While not one-to-one, it does lend itself well to the core idea of randomised generation of elements that can create avalanche effects: as one sand grain can create waves of topples, so too can a couple of cars create massive widespread congestion if appearing at the right position and time. In fact, the avalanche effects of the sand pile model also lend well to this application. With the sand pile model, a topple can propagate and unleash a series of consecutive topples across the plane and likewise with traffic, congestion at one point can with time extend to great lengths as one knows from experience. In this sense, they are similar.

As for why I chose this particular pathway as an application of sand pile modelling, part of it was the challenge. You may have noticed the previous sections were quite short and concise, and that was for a reason! Traffic modelling introduces many challenges (which will be covered soon) and thus more focus had to be placed on this section as a result. Another reason why is because of prior experience both in traffic modelling (though nowhere near as sophisticated as this time around) and actually experiencing traffic; where my appointments in the city would often require that I experience heavy doses of congestion and waiting while seeing barriers proclaiming "Save 20 minutes" for a year, across roadworks where only once have I seen anybody actually working. In a sense, I get to highlight how problematic this issue is for a project. Finally, I have for a while wanted to do some model building and this was my chance to make something that can colloquially be described as "very cool". To make this project manageable, it will be restricted to modelling traffic flow along a highway.

2.2: Differences and Challenges from Sand Pile Modelling

There are quite a large number of differences between the work that was done before and this application. For one, it cannot be assumed that the system is isotropic: it will behave differently based on where phenomena take place. To highlight why, it is as simple as reminding that if traffic congestion ends 500m along a highway and a driver is 750m along that same highway, that driver will not experience any effects from that dilemma. The same is not said for a driver 250m from where the congestion starts: that driver will experience the effect if they have not already. Moreover if traffic resumes, the cars further ahead will always move before the ones behind, they do not move simultaneously with each other. For that reason, a gradient matrix cannot be used to look up topple sites, for cars behind will be reactive to the cars in front who may very well change the conditions the cars further back will experience. A gradient matrix only works if the system reacts simultaneously to any event within it.

Another complication is the randomness of inserted material into the system. With the sand pile model, a grain of sand can be added one at a time anywhere across the system. For a traffic model, cars cannot just be placed in the middle of the road out of nowhere. Rather, cars can only enter from the main highway entrance, or through alternative entrances along the highway. Furthermore, the amount of cars entering at one time will always be random and will change with time of day (midnight vs peak hour traffic) so time-dependent probability distributions are required to appropriately determine how many cars can enter from one entrance in one timestep.

Removal of elements is similarly another concern; cars can only leave by reaching the end of a highway or by leaving through an alternative exit, not through any border of the matrix. This places significant restrictions on the piling up of cars compared to the piling up of sand.

Finally, the system will evolve even without a topple effect occurring. The elements of the system are always in motion, so unless there is congestion the cars are always moving across the matrix from one point to another regardless of any topples. Furthermore, traffic does not need to "topple" to disperse or contract, but can do so merely through probability and circumstance if one assigns probabilistic events to their speed for instance, or through assigning probabilities to lane switching. Evidently, the system has many different methods of evolving compared to the basic sand pile model which evolves either by obtaining a sand grain or by activating a topple.

Finally, the motion of cars will cause even greater headaches when one allows them to move in the opposite direction (i.e., the other half of the highway) which will be elaborated upon much later.

2.3: Computational Methods of Traffic Modelling

It is important to explain the underlying logic that the traffic model will operate on before looking at the development of the code. In essence, moving vertically in the matrix by one entry corresponds to roughly fifteen metres of distance covered, while moving horizontally across the matrix model corresponds to switching lanes. This is visualised in Figure 2.3.0. Each entry therefore allows multiple cars to be able to be occupy it at once (up to a maximal value of four), rather than each entry having only discrete values of zero (no car) and one (a car). In this sense, the entry then measures the density of cars within fifteen metres along a highway. Cars behave differently depending on how high the density is, with more cars corresponding to slower movement. As a rule of thumb for interpreting entry values, zero corresponds to no cars in the zone, one and two correspond to light traffic, three is high traffic, and four is heavy traffic.

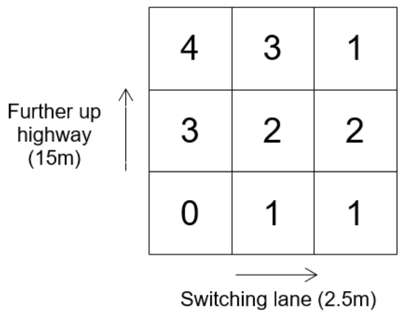


Figure 2.3.0: A demonstration of how distance changes when moving across the matrix.

As explained before, the system is not isotropic: cars further up ahead will generally not be dependent on entries behind them, but the same cannot be said for the inverse. To determine traffic flow, four gradients are measured for each entry which is shown in Figure 2.3.1: two horizontal gradients in opposite directions, and two vertical gradients in the same direction: a direct upper gradient (DUG) and second upper gradient (SUG). A weighted probability distribution is generated for whether the car will stay in its lane or switch lanes depending on the DUG and two horizontal gradients, with additional weighting being given for the car staying in its lane as cars will not generally switch lanes if they don't need to. The DUG and SUG determine how a car will evolve when travelling straight.

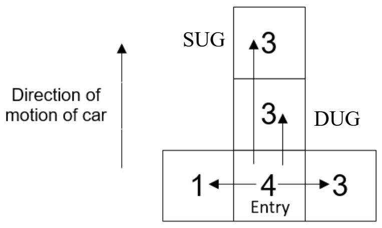


Figure 2.3.1: A diagram of the gradients each entry measures to determine movement.

The initial development of the model did not go very well. It began by taking the gradient version of the basic sand pile model and making tweaks to it. Traffic flows differently depending on the density of cars, so if statements had to be generated for each possible value an entry could take, which was wholly preserved in the final version. Using a gradient matrix was not a viable strategy, for the cars would still evolve regardless of any topple effect occurring so nested for loops had to be integrated once more to update cars in low density regions. Even worse, consider the calculation of the DUG and SUG. The gradient matrix is constructed before the evaluation of cars and topples each timestep, however if probabilistic processes such as cars switching lanes or slowing down were to take place, then the gradient matrix immediately became outdated. Finally, this model was made with the assumption that cars behind would be reactive to cars ahead, but by the time the code would iterate on the cars behind, the cars ahead would have already moved so the gradient matrix would again be supplying misinformation. The gradient matrix foundation had to thus be scrapped in favour of embracing nested for loops over the matrix much like in Version 1 of the basic sand pile model. Great efficiency was sacrificed for simply allowing the code to actually do its job properly. Additionally, an old problem is invited once more: recall in section 1.2 that it was beneficial to avoid iterating over a matrix with for loops due to bias in directions. Vertical bias is not an issue in this model as the system is not isotropic vertically, however it would be isotropic horizontally. Traffic can be more volatile on the outermost lanes where cars can enter and leave the system, whereas the innermost lanes would be the most static since there would be no exits or entrances to the highway; hence the horizontal iteration direction is chosen to be from the outermost to innermost lanes of the highway. Overall, this is an unfortunate consequence of the model, and it can be removed but the methods involved were not integrated into the model due to time constraints and will be discussed later. Figure 2.3.2 provides a visualisation of how the program would iterate across the highway horizontally.

Another thing that was creating significant headaches was the method of try and pass for the borders of the matrix. It was introduced in section 1.3 as a means of removing if statements and greatly boosting the efficiency of the code. In actuality it posed a problem: evaluating for example the left gradient of the left border of a matrix would be equivalent to measuring the gradient between the left border of the matrix with the right border, and this could not be filtered by those commands. Hence, if statements were brought back once again for border entries to reduce error, otherwise cars that were supposed to leave the highway would instead end up back at the beginning.

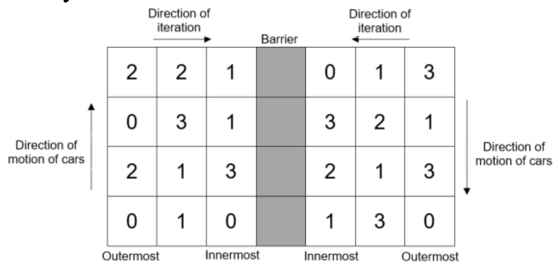


Figure 2.3.2: A diagram that shows the iteration direction of the code along both sides of the highway.

Regarding how car behaviour changes with entry value, it is important to explain why these gradients are so vital. Before evaluating the forward movement of cars in the entry, it decides whether any of the cars wish to switch lanes. It uses the horizontal gradients and the DUG to pull integers from a table. These integers act as weights which modify the weighted probability distribution of a car switching lanes or not. Generally, cars will prefer moving to regions of lower densities (particularly if the entry value is a three or four), but they also have a bias for remaining in their current lane. The program randomly determines whether each car in the entry will switch lanes to a horizontal neighbouring entry or not. To avoid cars switching lanes multiple times, a turn count is implemented which works as follows: if a car switches to an entry that was already iterated upon (an outer lane), an outer turn count increases by one. If a car switches lanes to a lane yet to be iterated upon (an inner lane), the inner turn count increases by one. The inner turn count means that when evaluating the next entry, the number of cars allowed to switch will reduce by how many switched into that lane. But for low density traffic cars don't slow down by switching, so they may still be allowed to move forwards. The outer turn count ensures that cars switching to an entry already iterated upon (an outer lane) can still move forward if the traffic density is low, but prevents the other cars in that entry from moving, for they already have in the same timestep. This ties back to the weakness of the model: it shows horizontal iteration bias, even if this mitigates it. All forward movement is then determined by the DUG and SUG, whose influence can vary with entry value. With all that said, there is now sufficient knowledge to explain the behaviours of each entry value.

Entry = 0

The trivial case, it means no cars reside there. It has interesting consequences for using gradients to determine traffic flow which will be covered shortly.

Entry = 1

One car resides within fifteen metres along the highway. This means the car will generally move quite quickly. If the DUG is equal to -1, it means the direct upper entry (DUE) is empty and the car can skip to the second upper entry (SUE) if there is room available. Figure 2.3.3 illustrates this process in detail. In the case where the SUG is equal to three (full capacity in the SUE), the car will only enter the DUE. The movement of the car is not affected by the turn counts when travelling forwards apart from being unable to switch lanes twice. Finally, cars tend to but are not restricted to staying in low density regions, meaning while not preferable they can spontaneously switch into high density regions of traffic.

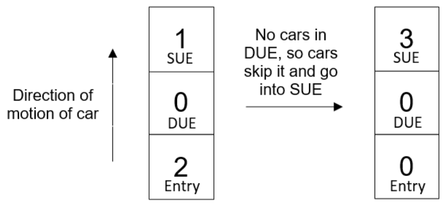

Direction of motion of car
1
SUE
0
DUE
2
Entry
No cars in DUE, so cars skip it and go into SUE
3
SUE
0
DUE
0
Entry

Figure 2.3.3: Entries of low value will skip the entry directly above them if they are empty to the second upper entry.

Entry = 2

Operates similarly to the case where the entry value equals one, except if the DUG/SUG equal -2, the DUE/SUE are empty respectively. If the DUG/SUG equal two, then the DUE/SUE are full. The primary difference is that if the DUE has value zero (empty entry) and the SUE has room for only one more car, then the two cars will disperse and one will enter the DUE and the other will enter the SUE when moving forwards. Lane switching remains mostly the same except cars are slightly more likely to switch to lanes of low density.

Entry = 3

Entry values of three correspond to high traffic conditions. When both the DUE and the SUE are empty (or the SUE has only one car), the three cars can either move altogether into the DUE, or one car moves into the SUE while the other two move into DUE. Each case has a 50% chance of occurring, to reflect the slow reaction times of drivers under high traffic conditions where it can be slow to disperse. This is unlike the previous entry values where the cars would move into the SUE if they could. Additionally, the turn counts play an effect on the flow of cars when they move forward. The effects of the DUG and the SUG remain, but the number of cars that can move ahead are reduced by the turn count. For example: if the DUG and the SUG allow three cars to move ahead but two cars switched into the current entry from the previously iterated entry, only 3 – 2 = 1 car(s) are permitted to move ahead by the inner turn count. If cars switch to an entry already iterated upon, the outer turn count only gives one car a 50% chance of moving ahead. The reason this was implemented was to highlight the effects of high traffic conditions: switching lanes disrupts the flow of traffic as cars have to stop to make room. Initially this was done for all possible entry values, but in low traffic conditions cars don't need to slow down to allow other cars to switch into their lanes so it was scrapped for entry values below three. Finally, cars are now likely to switch into lanes of lower density.

Entry = 4

This entry value represents peak traffic conditions, and operates quite differently to the other entries. If the DUE is empty and the SUE can afford at least one more car, the four cars can either; disperse (one car stays in the same entry, two travel to the DUE, one travels to the SUE), move altogether to the DUE, or not move at all. The probabilities of each action occurring are 50%, 40% and 10% respectively to account for slow reaction times to changing conditions under peak traffic. If the DUE is empty but the SUE is full, then all four cars will travel to the DUE to replicate the piling up effect of cars in peak traffic. If the DUE is not full but not empty either, only one car is permitted to enter the DUE. The turn counts work similarly for this case as for an entry value of three, except if a car moves into an already iterated entry, that car cannot move forward at all. The final property of this entry value is that cars will heavily favour switching to lanes of lower densities, even if it merely recreates the same density at a different entry. This entry value is the equivalent of the topple site of a basic sand pile model.

Now that the core mechanics of the model are illustrated, an explanation of how cars can enter the highway is warranted. An external program is used that supplies the locations of entrances along the highway. Each timestep, cars can spawn either from an entrance along the highway or from the beginning of the highway. The external program also houses a set of spawning matrices, and each entry in these matrices stores an integer value ranging from zero to four. Based on the timestep, a specific row is selected and a number is randomly chosen from that row, which determines how many cars spawn from that entry point in that timestep. By default, the first row corresponds to midnight time and consists mainly of zeroes, while another row corresponds to peak hour traffic and contains many fours. For example, if a value of one is chosen at an entry point, one car will spawn from that entrance into the highway. If the total number of cars in that entry exceed four, it is reduced back to four. Figure 2.3.4 visualises the process. Additionally, the user can select how many timesteps correspond to the same time row, and when the program exceeds that number of timesteps it switches to the next row in the spawning matrix. Both sides of the highway have their own separate spawning matrices, and side entrances also have their own matrices. This method of selecting how many cars should enter the highway was chosen as working with static probabilities for the whole simulation would not be representative of reality, for then the highway is either always congested or always running smoothly and neither on their own are interesting nor insightful.

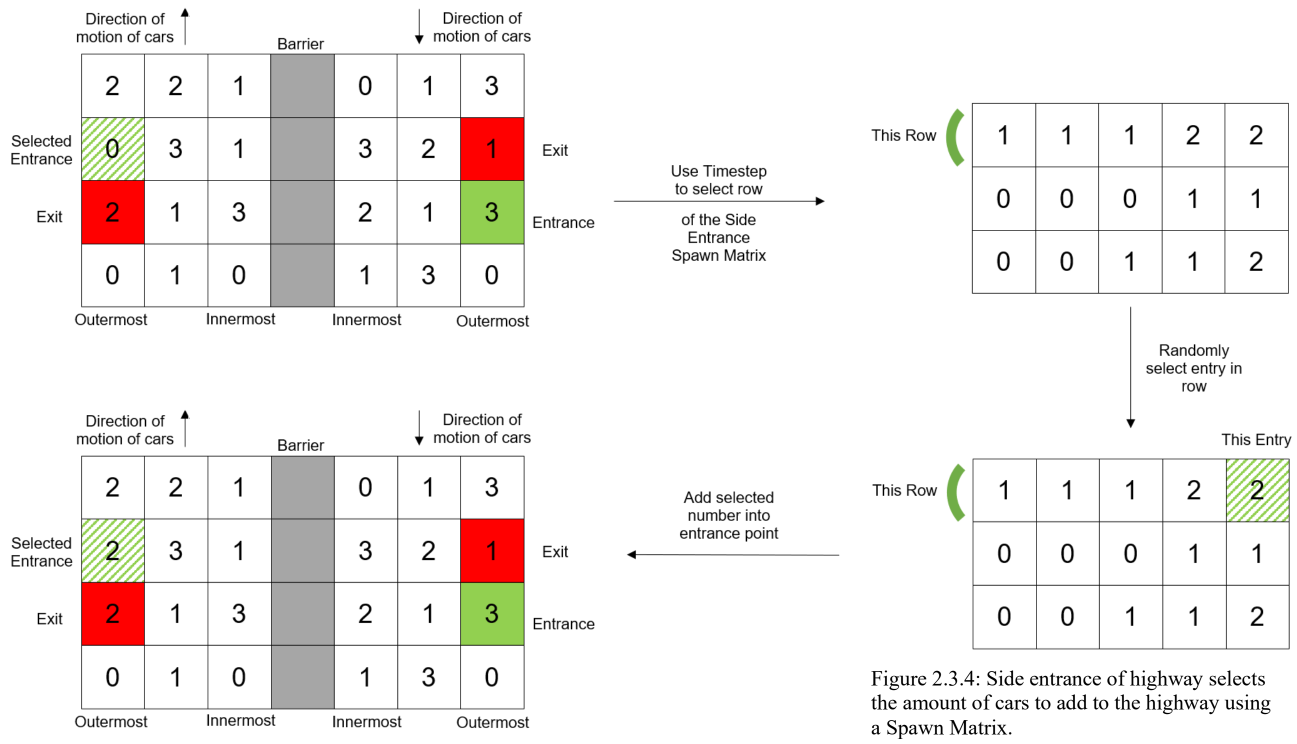

Direction of motion of cars
Barrier
Direction of motion of cars
Selected Entrance
Exit
Exit
Entrance
Outermost
Innermost
Innermost
Outermost
Use Timestep to select row
of the Side Entrance Spawn Matrix
This Row
Randomly select entry in row
This Entry
This Row
Add selected number into entrance point
Direction of motion of cars
Barrier
Direction of motion of cars
Selected Entrance
Exit
Exit
Entrance
Outermost
Innermost
Innermost
Outermost

Figure 2.3.4: Side entrance of highway selects the amount of cars to add to the highway using a Spawn Matrix.

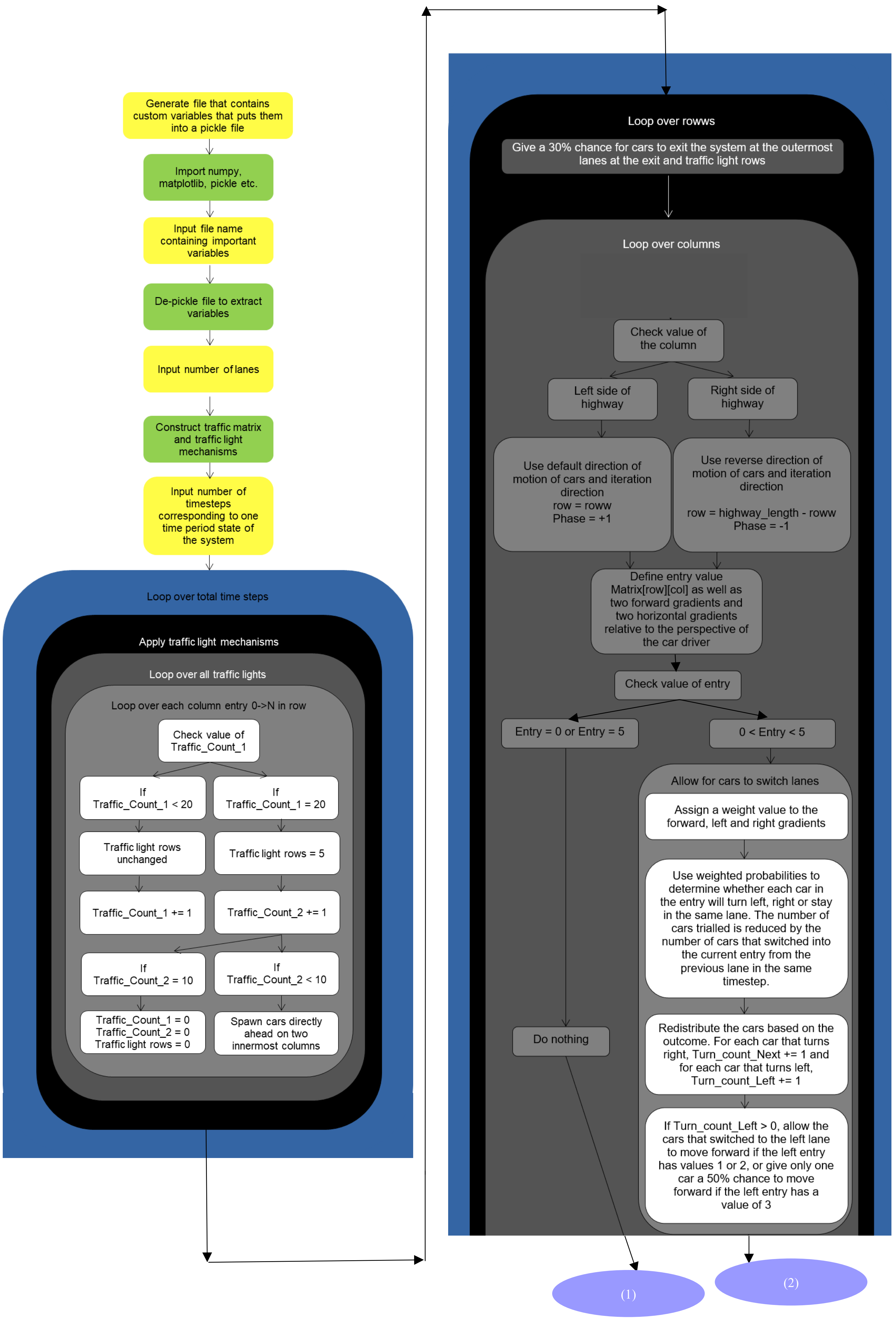
Regarding how cars exit the highway, it is a lot more simple to explain. If a car is travelling upwards/downwards through the top/bottom borders of the highway matrix respectively, they despawn and exit the highway. For the side exits however, their locations are also stored in an external program. At each new timestep the model will check if any cars are located at the exit points of the highway. Each car there rolls for a 30% chance of leaving the highway through that exit, and this occurs at the beginning of each timestep before cars can switch lanes in the highway. It is important to note that common sense dictates that exits are placed before entrances along a highway, otherwise there would be a clash between cars entering and cars trying to leave the highway. This is preserved in Figure 2.3.4.

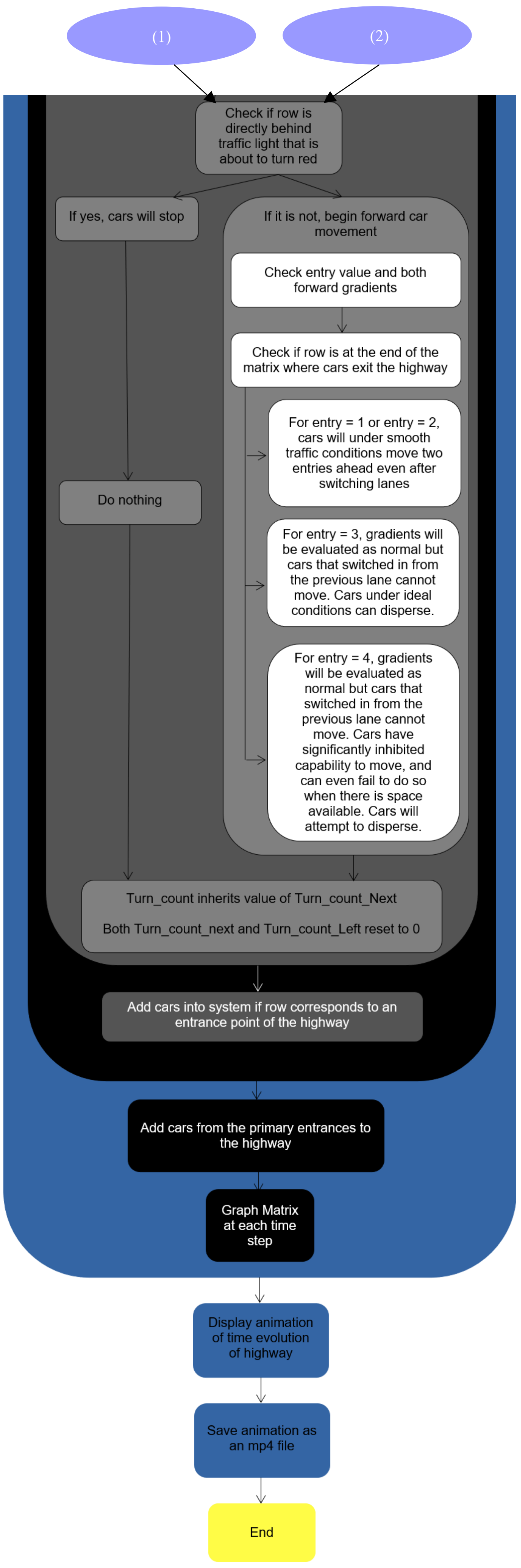
The second last device to be introduced to the highway model is the traffic light. The highway model is therefore extended to simulate not only the freeway (where there are no traffic lights but instead numerous entrances and exits), but also urban highways (where exits and entrances generally correspond to positions with traffic lights). The rows that contain traffic lights are stored in the same external program, which will be explained soon. The main program creates two arrays that contain separate traffic counts 1 and 2 (or TC1 and TC2) for each traffic light. TC1 keeps track of how long a specific traffic light turns green, and increases by one each timestep until it reaches a value of twenty where the traffic light turns red. TC2 then starts increasing by one each timestep until it reaches a value of ten. While TC2 increases, all entries in the row that the red traffic light resides in are set to a value of five (so no car may pass through them). Cars will then spawn in the two innermost lanes ahead of the red traffic light, using the side entrance spawning matrices. When TC2 reaches a value of ten: both traffic counts reset to zero, the entries in the traffic light row reset to a value of zero and the traffic light becomes green again, and so the process repeats. Traffic lights in reality do not switch colours at the exact same time, so each TC1 is initially randomised between zero and twenty so that when multiple traffic lights exist in the simulation, they too are not perfectly synchronous. TC2 for every traffic light however always starts at a value of zero. Another property is that when a TC1 reaches a value of 17, it begins a yellow traffic light effect where cars will not move forward if they are located just before the traffic light. This also prevents cars from accidentally being deleted from the simulation. Finally, rows with a traffic light also function as an as an exit point of the highway, with the same probabilities too. Unlike normal exit points, they effect both the outer and innermost lanes of the highway.

The final core ingredient to the model is merely adding the other side of the highway: where cars move in the opposite direction. The logic behind this facet of the simulation is simple to incorporate in theory, but very tedious in reality. Cars going in both directions obey Figure 2.3.1, but problems arise with the direction of iteration along the matrix. When cars only moved in one direction, the vertical iteration direction of the traffic grid was opposite to the motion of the cars, and this must be preserved for cars travelling in the other direction too. Otherwise, when iterating on an entry in the traffic matrix, the code pushes the cars further along the highway to the next row(s) to be iterated upon. When the code goes over to the next row(s) and finds the same cars there, it then allows those cars to move further up ahead again, and this process will continue to repeat until they either hit a red traffic light or reach the other side of the highway where they despawn. This would take place in one timestep, so cars can essentially teleport from one side of the highway to the other instantly. Hence the vertical iteration direction of the matrix needs to be inverse to the motion of the cars to prevent this. In addition, the horizontal iteration direction needs to be reversed for more subtle reasons: as a result of the presence of the horizontal bias in the evaluation of traffic, cars can behave differently when switching to one lane compared to another. This was distinguished between two cases; cars switching towards the outermost lanes, and cars switching towards the innermost lanes of the highway (Figure 2.3.2 visualises what is meant by innermost and outermost lanes). This must be preserved for cars travelling in the opposite direction, for it would mean that this horizontal bias would behave consistently between both car trajectories. That way, if one half of the highway features abnormal piling up of cars in the outermost lane as a result of this design flaw, then the cars travelling in the other direction will experience the same flaw in their outermost lane and not for instance in their innermost lane. This simulation has flaws, but it seems reasonable to at least keep them consistent across the board. There is a method to resolve this issue but there was not enough time to address it.

As a bonus feature, user compatibility and customisation was added to the program. The use of an external program that stores arrays and spawning matrices was repeatedly referenced in earlier sections, and this was why. The code allows for a user to import their own spawn matrices and locations for entrances, exits and traffic lights but at the cost of requiring a separate program to store these assets. It was impractical to force the user to manually input a spawn matrix each time the simulation code would run, however complications arise in allowing multiple viable external programs to contain that information rather than just one with a specific file name. To generalise the code to apply to any viable external program, it had to unpickle data from files created by those programs. A pickle is the process of saving the assets of a program (variables, matrices, etc.) in a separate file that can then be extracted into a different python script. These external programs would pickle the matrices and arrays, and the simulation code would unpickle and use those same assets. Now the parameters can be freely customised.

In concluding the computational methods used to construct this simulation, it is convenient (arguably necessary) to provide a flowchart that visualises the internal structure of the program. It spans two pages and had to be split into three components just to be adequately readable.


Generate file that contains custom variables that puts them into a pickle file
Import numpy, matplotlib, pickle etc.
Input file name containing important variables
De-pickle file to extract variables
Input number of lanes
Construct traffic matrix and traffic light mechanisms
Input number of timesteps corresponding to one time period state of the system
Loop over total time steps
Apply traffic light mechanisms
Loop over all traffic lights
Loop over each column entry 0->N in row
Check value of Traffic_Count_1
If Traffic_Count_1 < 20
If Traffic_Count_1 = 20
Traffic light rows unchanged
Traffic light rows = 5
Traffic_Count_1 += 1
Traffic_Count_2 += 1
If Traffic_Count_2 = 10
If Traffic_Count_2 < 10
Traffic_Count_1 = 0
Traffic_Count_2 = 0
Traffic light rows = 0
Spawn cars directly ahead on two innermost columns
Loop over rowws
Give a 30% chance for cars to exit the system at the outermost lanes at the exit and traffic light rows
Loop over columns
Check value of the column
Left side of highway
Right side of highway
Use default direction of motion of cars and iteration direction
row = roww
Phase = +1
Use reverse direction of motion of cars and iteration direction
row = highway_length - roww
Phase = -1
Define entry value Matrix[row][col] as well as two forward gradients and two horizontal gradients relative to the perspective of the car driver
Check value of entry
Entry = 0 or Entry = 5
0 < Entry < 5
Allow for cars to switch lanes
Assign a weight value to the forward, left and right gradients
Use weighted probabilities to determine whether each car in the entry will turn left, right or stay in the same lane. The number of cars trialled is reduced by the number of cars that switched into the current entry from the same previous lane in the same timestep.
Redistribute the cars based on the outcome. For each car that turns right, Turn_count_Next += 1 and for each car that turns left, Turn_count_Left += 1
If Turn_count_Left > 0, allow the cars that switched to the left lane to move forward if the left entry has values 1 or 2, or give only one car a 50% chance to move forward if the left entry has a value of 3
Do nothing
(1)
(2)



(1)
(2)
Check if row is directly behind traffic light that is about to turn red
If yes, cars will stop
If it is not, begin forward car movement
Check entry value and both forward gradients
Check if row is at the end of the matrix where cars exit the highway
For entry = 1 or entry = 2, cars will under smooth traffic conditions move two entries ahead even after switching lanes
For entry = 3, gradients will be evaluated as normal but cars that switched in from the previous lane cannot move. Cars under ideal conditions can disperse.
For entry = 4, gradients will be evaluated as normal but cars that switched in from the previous lane cannot move. Cars have significantly inhibited capability to move, and can even fail to do so when there is space available. Cars will attempt to disperse.
Do nothing
Turn_count inherits value of Turn_count_Next
Both Turn_count_next and Turn_count_Left reset to 0
Add cars into system if row corresponds to an entrance point of the highway
Add cars from the primary entrances to the highway
Graph Matrix at each time step
Display animation of time evolution of highway
Save animation as an mp4 file
End

2.4: Results and Output

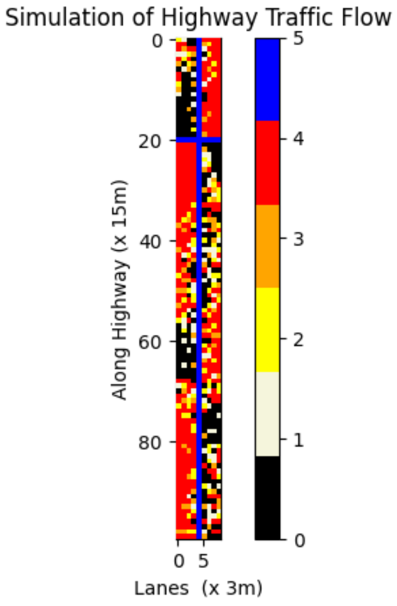


Figure 2.4.0: A snapshot of a simulation of traffic flow along a highway.

In allowing the code to run with parameters specified by an external script, the animation is of the form featured in Figure 2.4.0. Note that ffmpeg is required to save the animation. The cars on the left half of the highway travel upwards while the cars on the right half travel downwards. The colour bar to the right of the animation portrays the colours assigned to each entry value. This colour scheme was adopted from the humble beginnings of this project: the basic sand pile model. The blue colour however is new and corresponds to an entry value of five, where it represents two features of the model. The first is that the vertical streak of blue in the middle of the highway represents the barriers that distinguish the two sides of the highway. The second feature is shown in the horizontal blue streak at the twentieth row in Figure 2.4.0. The horizontal streaks only appear if a traffic light is red and situated in that row. This makes it clear to the user if a traffic light is green or red to make their effects on traffic flow more easily observable. Another characteristic of the animations are the main and axis titles, with the latter illustrating the scale of one unit of traversal in their respective directions in metres. Travelling upwards by one unit corresponds to fifteen metres traversed while travelling horizontally by one unit corresponds to travelling three metres.

Before discussing the results obtained in running the simulation, it is vital to explain the parameters used. The standard settings consist of having four lanes on each side of the highway, the length of the highway being one hundred units, and ten timesteps being assigned to every row in the spawn matrices. The default spawn matrices were used, with low traffic conditions set around midnight to 5:00am, peak traffic spawns occurring around 7:00am and 7:00pm, and the rest of the available times containing medium to high traffic spawn probabilities. The first row of the spawn matrices corresponded to midnight time, and ran on a 24-hour timeframe with spawn probabilities adjusting every 15 minutes in simulation (not real-world) time. The main and side entrances of the highway each have their own spawn matrices, but are shared between both sides of the highway. The side entrances are located at the 49th and 88th rows of the left side of the highway, and at the 89th and 50th rows of the right side. The side exits were positioned at the 89th and 50th rows of the left side, while being at the 88th and 49th rows of the right side. The traffic rows were positioned at the 20th and 70th rows of the highway. The animation was supplied with the submission of this report.

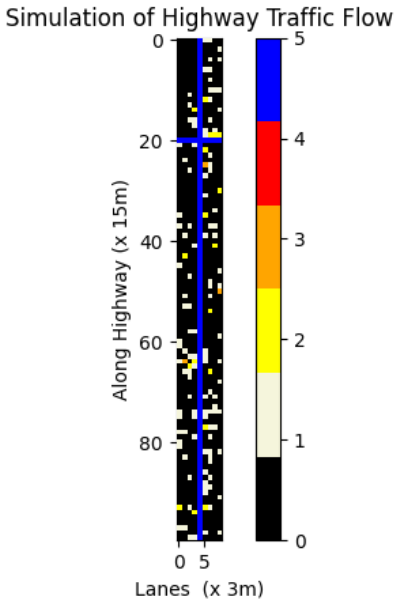


Figure 2.4.1: A snapshot of traffic flow along a highway past midnight.

The beginning of the simulation is fairly predictable, and is shown in Figure 2.4.1. Because the matrix was initially empty, all visible cars spawned from the side and main entrances to the highway. While it may seem like an issue that no cars were initially scattered across the highway, at midnight zero/one car(s) spawn each timestep at an entry point into the highway, so the traffic conditions are tame enough that the highway will generally be quite empty. It would therefore only require a relatively small number of timesteps to restore traffic conditions across the highway to being at expected levels. Moving on, the upper traffic light has turned red and created small amounts of congestion for the right side of the highway, and a small number of entries extend into values of two and even three as a result, so traffic is never completely smooth at this time.

When the morning rush of traffic (roughly 7:00am) hits, it is clear that many cars are entering the system at once. This is illustrated in Figure 2.4.2 on the next page. The combination of the surge of cars entering the highway and the presence of the traffic lights forces congestion to be widespread across the highway. The effects of the traffic lights are quite visible in Figure 2.4.0: with the drastic piling up of cars occurring behind them. Figure 2.4.3 shows the traffic conditions at a later time in the day when less cars are entering the highway, where unfortunately the morning flood of cars causes the accumulated congestion to never fade. After peak hour begins at roughly 6:30pm, the conditions of Figure 2.4.2 repeat and when it reaches midnight again, the end result is Figure 2.4.4. Congestion has noticeably reduced, but remnants of peak hour traffic still remain despite expectations, which will be discussed in the next section.

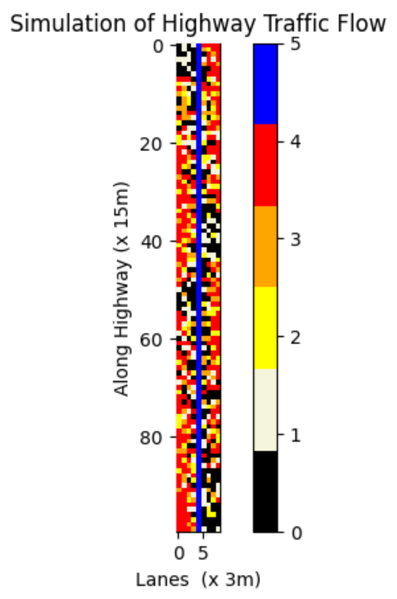


Figure 2.4.2: A snapshot of a simulation of traffic flow at the start of the morning rush.

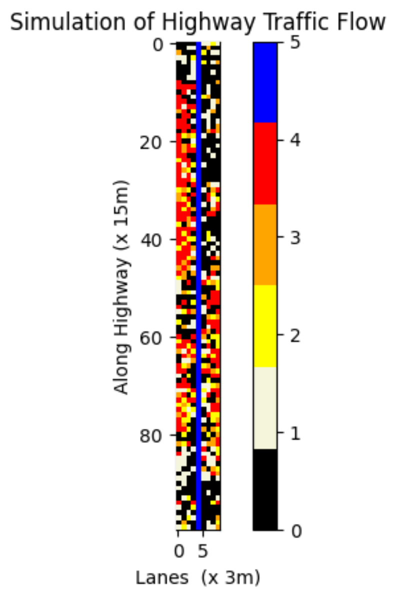


Figure 2.4.3: A snapshot of a simulation of traffic flow during off-peak time.

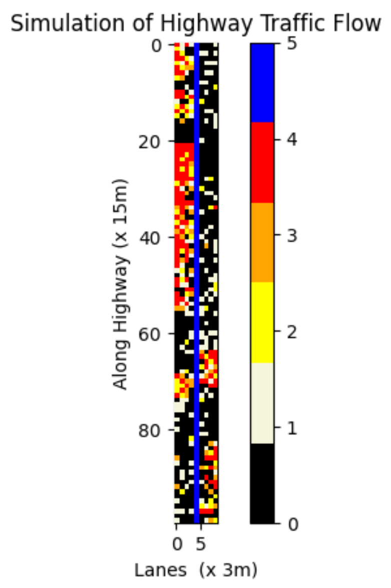


Figure 2.4.4: A snapshot of a simulation of traffic flow along a highway when returning to midnight.

There are two interesting traits of this simulation: the recurring density patterns and the distribution of densities among the lanes. If one observes Figures 2.4.2 and 2.4.3 carefully, there are oscillations in traffic density along the highway: regions with significant amounts of red and orange are contrasted with regions of black and beige. This simulation suggests that when traffic congestion occurs, the high densities of slow-moving cars are not preserved along the entire highway but rather come in waves. A possible explanation for this is the red traffic lights cause cars to pile up significantly (as displayed in Figure 2.4.0), while a small amount of cars enter the system in front of the traffic light. Additionally, cars experiencing peak densities have a high probability of dispersing, with one car travelling further ahead while the cars behind continue to pile up. These two properties of the simulation pose a sensible explanation to the phenomena occurring. As for the distribution of density of cars across the lanes, Figures 2.4.2 and 2.4.3 demonstrate that the outermost lanes seem to be the most likely to be at full capacity while the innermost lanes seem to be the most likely to be empty. This can be explained by the fact that the side entrances to the highway occur at the outermost lanes, so cars are frequently entering the highway from there and thus congestion is constantly building from those positions. Overall, this simulation displays properties that seem to be consistent with reality and expectations. In real life, traffic congestion tends to occur in waves rather than stretch uniformly for kilometres across a highway, and cars tend to congregate around the exits and entrances to highways under peak hour traffic conditions purely from the sheer number of them present.

2.5: Potential Areas of Improvement

This section will focus on pre-existing aspects of the simulation that could use further development. There are several features of the code that could be improved, as evidenced by the existence of traffic congestion at midnight in Figure 2.4.4 when the simulation finishes, which should not occur. Beyond merely adjusting probabilities for various stochastic processes, it would be worthwhile to add separate probability tables for cars switching lanes and staying in their lane based on their entry values and gradients. Currently both events share the same probability table, but different weighting factors are applied to prevent cars from constantly switching. This addition could force cars in high-density regions to move into lower-density regions more consistently, and likewise mean cars would generally remain in their lane in smooth traffic conditions rather than constantly switch back and forth as the current simulation does. On the topic of switching lanes, the current method for managing this facet of the program could use a rework. As it stands, there is a horizontal bias across the highway which as explained in section 2.3 should not be present. To remove this, it is suggested that before iterating over each column in a selected row of the matrix, the calculations for switching lanes are carried out for all cars in that row simultaneously. Doing this would prevent the asymmetry arising from cars switching into outer lanes compared to switching into inner lanes, and reduce runtime as the outer turn count would no longer need to be distinguished from the inner turn count, and thus the specialised outer turn count calculations could be altogether removed. Unfortunately, there was not enough time available to apply this rework to the simulation.

3: Conclusion

In summary, the basic sand pile model was developed for square matrices. Three distinct versions were created: the first functioned as more of an introduction to the expectations of creating a BTW model, the second was a fully realised BTW model, and the third version operated similarly to the second but used gradients to determine the redistribution of sand from topple sites. The second and third versions reliably demonstrated avalanche effects that occurred as a result of topples inducing further topples.

The basic sand pile model was then applied to the context of traffic modelling along a highway, where significant changes had to be made to facilitate its implementation. The addition of cars being able to move on their own, topples being defined in terms of probabilistic processes, traffic lights, and more caused the demand in programming proficiency to skyrocket compared to the basic sand pile models. The end result was a simulation that could by default model traffic along a highway in a 24-hour timeframe while accounting for characteristics of standard driving behaviour and evolving conditions such as the peak hour rush. The simulation produced interesting results, most notably that congestion would never fully disappear throughout the day as a result of the initial morning flood of cars, and that congestion would appear in waves across the highway rather than being a uniform presence throughout the roads.

Regarding the possible ways to expand the simulation, the simplest would be to install separate probability tables for cars staying in their lane and cars switching lanes to fix some behavioural issues featured in low and peak traffic conditions. Additionally, a total rework of the lane switching mechanic would be the most vital change for the code as it would rid the simulation of unneeded biases and processes featured in the evaluation of traffic. In terms of new features, the most obvious would be to add car crashes, which were originally scheduled to be incorporated into the simulation but were scrapped due to time constraints. Another natural addition would be different vehicle types: for instance trucks would take up the space of two cars but move slower than average, speedsters would generally move faster than the average car but be prone to crashing, police cars would behave as normal cars until in the vicinity of a speedster where they would speed up and pursue them, and much more. Another feature could be giving each car a destination such that they would be assigned an exit that they would need to reach, which would have interesting repercussions for peak hour traffic flow. Finally, being able to adjust the geometry of the highway would be very interesting. Taking a banked curve for instance, cars in the innermost lanes of the curve would move faster than those in the outermost lanes. All in all, the strength of this model lies in that the possibilities for expanding its features are practically endless due to its basic but rigorous methods of evaluating traffic flow.

4: Bibliography

Primary References

1. BTW Model:
   - P. Bak, C. Tang and K. Wiesenfel, “Self-organised criticality: an explanation of the 1/f noise”, Phys. Rev. Lett. 59, pg. 381-384 (1987) 10.1103/PhysRevLett.59.381.
   - P. Bak, C. Tang and K. Wiesenfel, “Self-organised criticality”, Phys. Rev. A. 38, pg. 364-374 (1988) 10.1103/PhysRevA.38.364.
2. Official numpy documentation: https://numpy.org/doc/stable/index.html
3. Official matplotlib documentation: https://matplotlib.org/stable/index.html
4. Official python documentation for “random” library: https://docs.python.org/3/library/random.html

Secondary References

1. Appending Numpy arrays: https://www.tutorialspoint.com/numpy/numpy_append.htm
2. Rounding float numbers: https://www.freecodecamp.org/news/how-to-round-numbers-up-or-down-in-python/
3. Using random.choice: https://pynative.com/python-random-choice/
4. Animating using matplotlib: https://www.youtube.com/watch?v=7RgoHTMbp4A
5. Using remainder operation: https://www.educba.com/python-remainder-operator/
6. Selecting weighted randomised values: https://pynative.com/python-weighted-random-choices-with-probability/
7. Guide to installing ffmpeg: https://www.youtube.com/watch?v=r1AtmY-RMyQ
8. Implementing pickles: https://www.digitalocean.com/community/tutorials/python-pickle-example

FAQ References

1. Obtaining index from random numpy array element: https://stackoverflow.com/questions/18794390/how-to-get-the-index-of-numpy-random-choice-python
2. Using np.argwhere: https://stackoverflow.com/questions/44296310/get-indices-of-elements-that-are-greater-than-a-threshold-in-2d-numpy-array
3. Plotting 2D Matrix: https://stackoverflow.com/questions/42116671/how-to-plot-a-2d-matrix-in-python-with-colorbar-like-imagesc-in-matlab
4. Matrix Animation:
   - https://stackoverflow.com/questions/17541449/displaying-numpy-matrix-as-video
   - https://stackoverflow.com/questions/34975972/how-can-i-make-a-video-from-array-of-images-in-matplotlib
5. Try and pass techniques:
   - https://stackoverflow.com/questions/19522990/catch-exception-and-continue-try-block-in-python
   - https://stackoverflow.com/questions/19081515/python-try-except-pass-on-multi-line-try-statements
6. Numpy array errors with indexing: https://stackoverflow.com/questions/46902367/numpy-array-typeerror-only-integer-scalar-arrays-can-be-converted-to-a-scalar-i
7. Adding feature for code to stop if anomalies arise: https://stackoverflow.com/questions/543309/programmatically-stop-execution-of-python-script
8. Checking if variable is integer: https://stackoverflow.com/questions/3501382/checking-whether-a-variable-is-an-integer-or-not

Special thanks goes to Balu Sreedhar and Roger Rassool for assisting with both the ideation and implementation of various aspects of the models that allowed the simulations to be possible.