Summary Sheet

If you have ever been on an aircraft, it is plane to see how slow boarding and disembarking is. For many this is insignificant, but for an airline company saving even a couple of minutes for each flight's boarding and disembarking will result in huge savings when considering the tens of thousands of airports and flights that occur each time. For an industry still struggling from the collapse of the tourism industry due to COVID-19, optimal and robust boarding and disembarking methods must be found.

To achieve this we developed two models, one for each of boarding and disembarking. As boarding and disembarking planes is an inherently stochastic process, we created a computational simulation over a pure mathematical model. Thus, we could better account for variable human behaviours and scenarios, giving a much more accurate distribution of data. Whilst many models already exist for this purpose, a key point of difference of our model is a greater consideration to several aspects of human behaviour. Namely, disobedience of boarding instructions, and travelling in groups.

We first modelled the Narrow Body Aircraft, simulating different boarding and disembarking methods using a Monte Carlo method. To create different boarding methods, we generated a randomized queue of passengers in the order that the boarding method prescribes (accounting for disobedient people) which could then be simulated boarding. Over many simulations, we could obtain an accurate average for the total time taken, allowing us to determine the most optimal method (least time taken). We also proposed two additional methods and ran them through the same simulations.

To simulate disembarking, we gave all seated passengers a priority value. Disembarking was carried out by moving passengers towards the exit at different rates dependent on their priority level. By altering the priority values we could carry out different disembarking methods and account for disobedience.

Both models implemented real-world data for factors such as moving speeds. This was to ensure the highest accuracy of our resulting times. We comprehensively analysed the results of these simulations, determining the effect of altering variables such as the number of people who disobey instructions, and varying numbers of carry-on baggage.

We adapted our models to two other passenger aircraft, the Flying Wing and the Two-Entrance Two-Aisle, and applied the most optimal boarding and disembarking methods used on the Narrow Body plane. Furthermore, we considered the effect of a reduced capacity of the passenger aircraft, a relevant deliberation in the age of COVID-19.

Overall, it was found that for boarding, one of our own proposed methods – boarding in the order of window, middle and aisle seats with the allowance of groups to board together – was on the whole the most optimal over the three aircraft. The optimal disembarking method was one in which the plane was unloaded from the back of the craft to the front.

Contents

1 Introduction

1.1 Background

As society becomes increasingly globalised, the importance of air travel grows. Flight numbers before the COVID-19 pandemic were at an all-time high, and they have doubled in the past 20 years. Following a temporary disruption due to COVID-19, this trend appears ready to continue it's steady upwards climb[1]. This has the consequence that small optimisation changes can result in enormous savings for both airline companies, passengers, and airports in terms of usable time wasted. Some of the biggest bottlenecks for plane turnaround are boarding and disembarking efficiency - that is, the way that passengers are loaded to and unloaded from planes[2]. There exist a variety of methods for these processes, each with varying theoretical and practical efficacies. As such, this report presents our developed model and simulates different onboarding and embarking methods for various aircraft models.

1.2 Problem Restatement

To ascertain the efficiency of different systems, we will develop two models with allowances for practical considerations that can be adapted to a variety of conditions.

1. Develop a plane boarding model and disembarking model which allows us to test the efficiency of different boarding/disembarking methods on a narrow-body plane
2. Adapt the models to test on different aircraft types (i.e. Flying Wing and Two-Entrance Two-Aisle) and also the effects of limited capacity flights due to COVID-19
3. Write a one-page letter to an airline executive that explains our results and its benefits to their airline

1.3 Basic Assumptions

Our initial model uses a few basic assumptions. The aircraft is to be divided into cells which one person can occupy at a time. The aisle space between rows and each seat is represented by one cell.

- Only one person can comfortably walk in an aisle cell
  Justification: Although aisle width varies by aircraft, a reasonable estimate is 0.50m wide[3]. On average, men have longer shoulder width than women, at 0.41m wide[4] and passengers are often carrying luggage which increases their width requirement. Thus, it is reasonable to assume that only one person can walk down the aisle at a time, with passengers both being laden with bags, respecting personal space, and potentially being weary of close contact due to infection risks. As such, when a passenger is loading their carry-on luggage into an overhead bin, the aisle is also blocked.

- Seated passengers block passengers who wish to sit further down in the same row
  Justification: The passenger cannot leap over the seated passenger. Not only is this valid from a social etiquette perspective, but in the provided aircraft designs, legroom looks to be minimal so it is physically unfeasible too.

- When a seat passenger leaves a row to make room for an incoming passenger, they are momentarily able to inhabit the same aisle cell
  Justification: As the passenger will want to reach their seat, they will not mind temporarily having reduced room as they move into their seat cell.

- Time to walk one aisle cell is constant
  Justification: This time was obtained by analysing a sample of $n = 10$ YouTube videos of people walking down aisles on flights, by counting the number of frames elapsed when each individual walks one aisle cell, and the playback details of the YouTube videos (typically either 60 or 30 frames per second - these are listed in the references). Using this, we can determine that the time to move one cell down the aisle is given by $1.05s$.

1.4 Variables and Factors

Several variables were used in our model to account for real-life phenomena. Some of these will be expanded on in later sections.

A **bag coefficient** was used to give a weighted probability of each passenger having carry-on luggage that they would want to stow in an overhead locker.

Another variable was the **number of groups**. Passenger populations are not homogenous; often they contain inseparable groups such as families of varying sizes. Members of these groups were seated adjacently in the same row and entered the plane in adjacent cells too. Upon entering the plane, it was assumed that groups would be in an order that would minimize blockage when getting into seats (i.e. in the order window, middle, aisle). This is reasonable as groups would want to minimize their own inconvenience and could communicate with each other to align themselves in this order. This factor has an appreciable effect on different boarding methods and was rarely investigated with any depth in any of the papers found in our literature review.

A **disobedience coefficient** was introduced to model the common scenario of passengers not following instructions. In these cases, a passenger (or group) would enter the plane in a different boarding category than ordered, which could be caused by ignorance, impatience or lateness. This, much like the **number of groups**, was rarely considered in an in-depth manner in the existing literature but would still significantly affect boarding times.

2 Narrow-Body Boarding

For both our models, we simulated the entire boarding/disembarking process. Keeping track of time during this simulation, we could calculate total boarding/disembarking time. Python 3.9 was used for this simulation.

2.1 Boarding Model Situation

To model boarding, we designed an algorithm that would see all passengers make their way to their assigned seat. Once on the plane after waiting in the boarding queue, passengers would follow a rigid set of rules, and variation would naturally occur due to variation in input: passengers had randomly generated differing numbers of baggage, and orders in which they entered the plane. Different boarding methods would be accounted for in the order of which passengers in prioritized seats entered the plane.A simplified process of the model as experienced by a passenger is best represented in the flow chart in Fig 2.1. This logic is easily followed and provides a robust algorithm that passengers can follow.

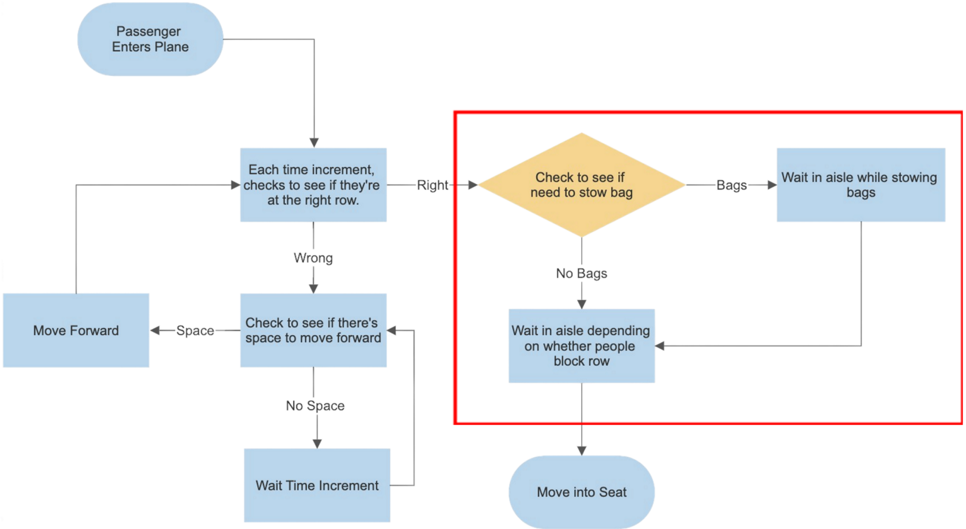


Figure 2.1: The logic behind passenger movement in the narrow-body aircraft

In the model this is simulated for all passengers simultaneously, as any passenger in the aisle could be at any step at any time. This is done by repetitively iterating down the aisle, starting from the passenger furthest from the entrance. Their state is determined, and an action done accordingly. Since it is assumed that there is a steady flow rate into the plane, if the first position in the aisle is ever empty, then the next passenger in the boarding queue occupies this space – 'passenger enters plane' in the flow chart. A key part of this simulation is the concept of an **internal clock**. Each passenger has this attribute, which counts down the real time (e.g., 1 sec) until they can complete an action. For example, the time to progress one cell forward is constant. The section of the flow chart enclosed in red is implemented in the simulation by calculating the total time that these actions would take and increasing the passenger's internal clock until this time is achieved, whereupon they can undertake their action. A visualisation tool was used on the code, allowing us to generate real-time visualisations of the simulations (see Fig 2.2, and the code in Appendix NUMBERHERE).

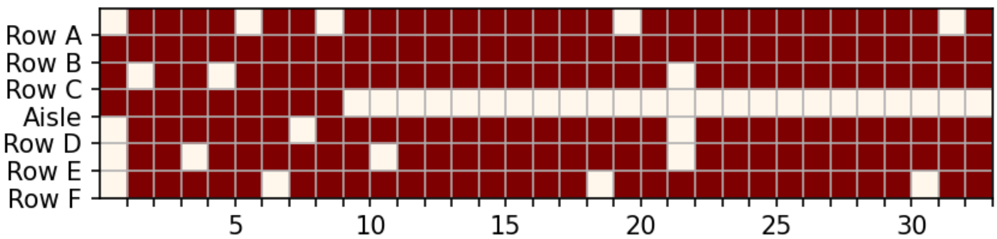


Figure 2.2: Visualisation tool in use on the narrow aircraft. Note that the aisle is currently blocked by a passenger in row 9.

In the following sections, we derive how these times are calculated.

2.1.1 Carry-on Baggage Delay

In airplanes it is commonplace that passengers load their carry-on luggage into the overhead bins. The aisle is blocked for the duration of this process. To account for this, the following piecewise function was developed to model the time that each passenger blocks the aisle while loading carry-on bags (which impedes the flow of passengers down the aisle).

Variable	Description
T_{bags}	Time that the main aisle is blocked due to carry-on luggage loading
n_{bags}	Number of carry-on bags to be stored in the overhead bins
n_{bins}	Number of carry-on bags present in overhead bins before storing
n_{max}	Maximum number of bags overhead bins can hold
C_0, C_1, C_2	Scaling constants depending on the value of n_{max}

$$T_{bags}(n_{bags}, n_{bins}, n_{max}) = \begin{cases} 0 & \text{if } n_{bags} = 0 \\ \dfrac{C_0}{1 - C_1 n_{bins}/n_{max}} & \text{if } n_{bags} = 1 \\ \dfrac{C_0}{1 - C_1 n_{bins}/n_{max}} + \dfrac{C_2}{1 - (n_{bins}+1)/n_{max}} & \text{if } n_{bags} = 2 \end{cases} \tag{1}$$

The model only considers $n_{bags} \in \{0, 1, 2\}$ since it is assumed that the maximum number of carry-on items that each passenger is permitted to have is $n_{bags} = 2$. Many airlines, including Air New Zealand[5], impose this maximum (even for business class passengers). The benefit of this equation is in its generality; its many parameters allow for precise calibration to produce more accurate results, especially for different aircraft models. For the purposes of modelling the narrow plane, we assumed each row of three had an overhead bin with capacity $n_{max} = 6$ since each passenger in the row could carry at most $n_{bags} = 2$. This is assuming not all the stowed items are full size suitcases: some carry-ons are likely to be smaller items such as handbags/tote bags. The passengers will be able to fit more of these into an overhead bin, thus the larger capacity. Then, taking $n_{max} = 6$, the values of C_0, C_1, C_2 were calibrated to be 4, 0.8, and 2.25 respectively. This yields the following equation, which was implemented into our model.

$$T_{bags}(n_{bags}, n_{bins}, 6) = \begin{cases} 0 & \text{if } n_{bags} = 0 \\ \dfrac{4}{1 - 0.8 n_{bins}/6} & \text{if } n_{bags} = 1 \\ \dfrac{4}{1 - 0.8 n_{bins}/6} + \dfrac{2.25}{1 - (n_{bins}+1)/6} & \text{if } n_{bags} = 2 \end{cases} \tag{2}$$

The function is piecewise to easily account for the varying number of bags that each passenger carries. Passengers carrying no bags do not take time to stow, while those stowing two bags take longer than those stowing one bag (thus the added term). Another consideration is that the function is designed to increase when there is less space in the overhead bin (i.e., when n_{bins}/n_{max} is large) as passengers will have to find space and squeeze their bags in, increasing aisle blockage time. For instance, if a passenger has one bag and there are already 1/6 bags in the overhead bin, then $T_{bags}(1, 1, 6) = 4.6$. However, if the compartment is almost full with 5/6 bags, then $T_{bags}(1, 5, 6) = 12$ as the passenger will have to locate a space and squeeze their carry-on in.

2.1.2 Shifting Seats Delay

Another large source of aisle blockage arises from the common situation where a passenger tries to reach their seat in a row but is blocked by a seated passenger. Before the passenger can reach their seat, the seated individual must stand up and move out to the aisle to allow the passenger to reach their seat, before sliding back. This process is lengthy and will impede the flow of passengers down the main aisle. This is furthermore complicated by the fact that there are many variations on this scenario, with different seated passenger positions and passenger seat goals, which will have appreciably different delay times. To model the additional time needed for these different shuffles, Eq. 3 was derived.

Variable	**Description**
$T_{shuffle}$	Time that the main aisle is blocked
t_{up}	Time taken for a seated passenger to stand up
t_s	Time taken for a passenger to travel the width of a seat
f	The index of the furthest seat that blocks the passenger's seat
n_s	The number of seated passengers that block the passenger's seat

Let the seats be indexed such that the aisle seat has index 1 and the index of each consecutive seat increases until the window seat. Since we are only concerned with total time that the aisle is blocked, only the time that passengers are occupying the aisle needs to be kept track of. First, the person seated furthest from the aisle stands and moves into the aisle ($t_{up} + ft_s$). Then the passenger moves into the row (t_s), and finally the previously seated passengers move back into the row ($n_s t_s$).

$$\begin{aligned} T_{shuffle}(f, n_s) &= t_{up} + ft_s + t_s + n_s t_s \\ &= t_{up} + t_s(f + 1 + n_s) \end{aligned} \tag{3}$$

Following this derivation, we state that the equation makes the following assumptions:

- The seated passengers notice the passenger once they are standing next to the row
- All the required seated passengers stand up at the same time and begin to exit the row
- That two people can inhabit the aisle cell adjacent to the row (assumed earlier)
- Once the passenger has entered the row, the previously seated passengers begin moving back into the aisle, following right behind the passenger in the correct order

These assumptions are sufficiently realistic to generate results which closely model reality.

2.2 Boarding Queue Generation

A queue of passengers with assigned seats was generated to move into the aisle. By altering the order of the passengers in this queue, we could simulate different boarding methods. For example, we could place everyone in the queue in order of aft, middle, front. Within these sub-sections of the queue, the order was randomized each trial to further increase realism. At this point, we also assigned each passenger a discrete number of baggage, either 0, 1, or 2. This was done by utilizing a weighted probability. Overall, we implemented algorithms to create boarding queues for all the required boarding methods, as well as several others. However, to increase realism of the model, we added additional variation within these.

2.2.1 Disobedience Coefficient

Undoubtedly, there will be passengers who do not follow the rules of whichever boarding method is in place. This is due to two main reasons: impatience (boarding before they are called), and lateness (being late to their boarding time). These passengers are rarely accounted for in the literature, yet they have an appreciable effect on boarding times. To include this in our model, we introduced the **disobedience coefficient**, ψ, the probability of any passenger in the queue to not follow the desired boarding method. For instance, in a sectional boarding method, a passenger sitting in the aft section of the plane would have a ψ chance of boarding with a different group (and given that they do, a 50% chance for either group). Initially this was fixed at $\psi = 0.3$; online studies found that 30% of passengers are late for their flights , and we thought that this was a reasonable number that would be impatient as well.

2.2.2 Groups of People

Another important consideration in the model is the existence of groups of people that board together. Families, couples, and the like are present in high concentration on flights and are often seated together. Importantly and as discussed previously, they board together and enter the queue in the way in which they would enter seat rows, decreasing total boarding time. To account for this in our model, when a passenger in queue is generated, there is a weighted probability that they will be in a group of 1, 2 or 3. Groups of 1 are simply regular passengers. Groups of 2 or 3 are adjacent in the boarding queue and are seated in adjacent cells. Groups of 4 or larger were excluded since the aircraft only allowed a maximum of 3 to sit together in a row, effectively meaning a group above 3 can be split into two groups. Initially, the weighted probabilities of a passenger being in a group of 1, 2 or 3 was set at (20,80,10).

We also considered the effect of the disobedience coefficient on groups. We initially considered a group to be disobedient if any members of the group of size n were disobedient. However, as $(1 - \psi)$ is the probability that a passenger is obedient, then $(1 - \psi)^n$ is the probability that the entire group is obedient. Hence, $1 - (1 - \psi)^n$ is the probability that the group would be disobedient. For a ψ value of 0.3, this would create a disobedience probability of 0.51 for groups of 2 and 0.657 for groups of 3. We thought that this was unrealistically high, and instead determined that the disobedience probability would be ψ for the entire group.

2.2.3 Bag Coefficient

A key stochastic variable in this model is the number of carry-on bags that any given passenger will stow in the overhead lockers. Just as in real plane boarding, this is clearly prone to variation. To account for this, we introduced another 3-tuple in the code to give a weighted probability of a passenger stowing either 0, 1 or 2 bags. Unfortunately, there was a lack of available data on average passenger bag count online. As such, further analysis of the previous YouTube videos allowed us to tentatively obtain an estimate of (20,80,10). However, in the sensitivity analyses later this value was changed appropriately, allowing us to determine the validity of this initial assumption.

2.3 Modelled Results for Provided Boarding Methods

The three provided methods for boarding were random boarding, boarding by section, and boarding by seat. It was assumed that boarding by seat would make no allowances for groups of people. However, the other methods were modelled using groups.

2.3.1 Random Boarding

At first glance, the method of random boarding seems crude and inefficient. However, simulations run on our model reveal that the random method is reasonably effective. It took on average 689.4 seconds to finish boarding the plane, with a 5th percentile of 626.7s and a 95th percentile of 755.7s. This means that 90% of the values fall in this range of 129 seconds.

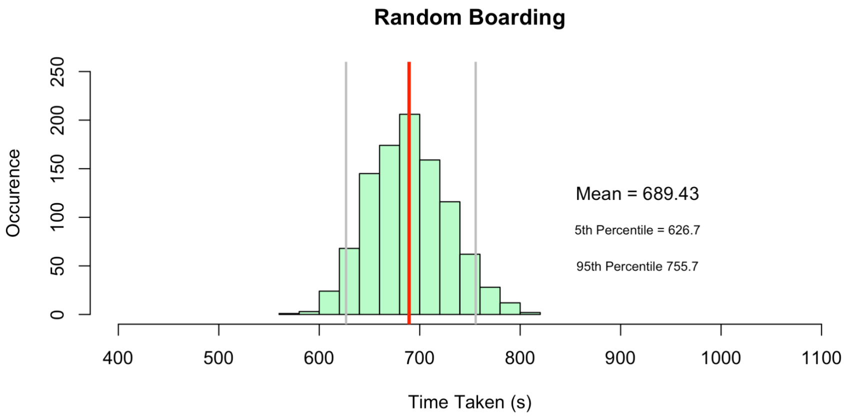


Figure 2.3: Monte Carlo simulation graph of the random boarding method

2.3.2 Boarding by Section

The second supplied method was to board the plane in sections. Boarding by aft (rows 23-33), middle (12-22) and front (rows 1-11) sections in varying order produced different results in our model. A set of results for all possible variations can be seen in the bar chart in Fig 6.1, but we discuss only the most and least optimal methods.

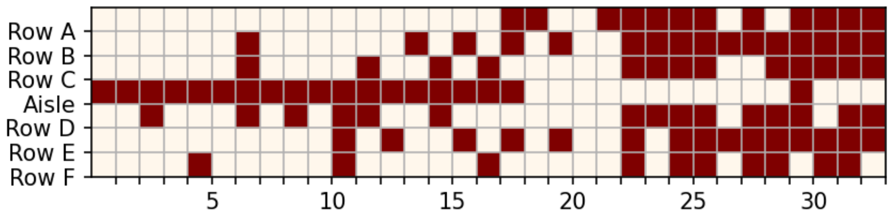


Figure 2.4: Visualised boarding by section starting with the aft. Note the disobedient passengers who have already seated themselves in the front and middle sections.

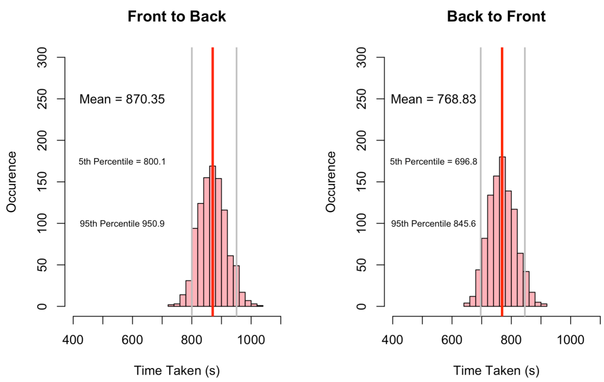


Figure 2.5: Monte Carlo simulation graphs of boarding front, middle, aft and aft, middle front

After running 10,000 trials, we found that the most optimal order of boarding was aft, middle, front (see breakdown in Appendix A). The mean time taken to fill up the narrow body airplane was **768.8 seconds**, with 90% of the times falling between 696.8s and 845.6s (spread of 148.8s). In comparison to this, it took on average **870.4 seconds** to board using the front, middle, aft method, with 90% of the times between 800.1s and 950.9s (spread of 150.8 seconds). This difference can be explained by considering Fig 2.6. On the left visualisation, the back fills first and so there is room to queue in the aisle, while on the right when the front is boarded first, the queue extends outside of the plane. Interestingly, this common method for boarding the plane is actually significantly slower than a random boarding order. However, the ability to simplistically split boarding into groups of people is valuable for airline companies, as it provides structure as to who should line up when. In the random boarding method, everyone is called to line up at once. This may potentially cause large queues and waste passengers time queuing in a long line.

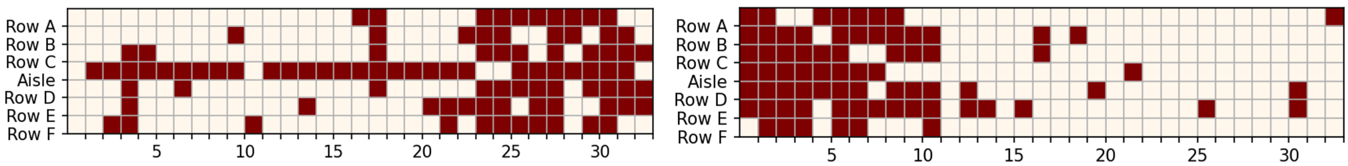


Figure 2.6: Visualisation of boarding by section, with AMF on the left and FMA on the right. Note the disobedience passengers sitting in the incorrect sections.

2.3.3 Boarding by Seat (WMA/WilMA)

The plane can also be boarded by seat type. This method allows all passengers with a window seat to board first, then middle, and finally aisle seats. Initially it seems like an ideal boarding method as it is relatively fast, with a mean boarding time 519.1 seconds. It is consistent too, with 90% of the values within 85 seconds of each other (5th percentile 479.1s, 95th percentile 564.1s). Not only this, but it is also straightforward to implement, with 3 easily definable groups of passengers. However, it splits groups. This is effectively unworkable in practice due to the separation of groups, particularly in the case of children and elderly.

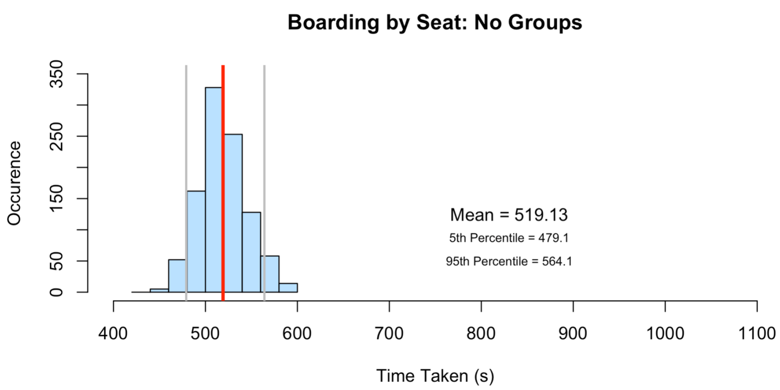


Figure 2.7: Monte Carlo simulation graph of boarding by seat without groups

2.3.4 Sensitivity Analysis of Provided Boarding Methods

We now perform a sensitivity analysis on the provided boarding methods.

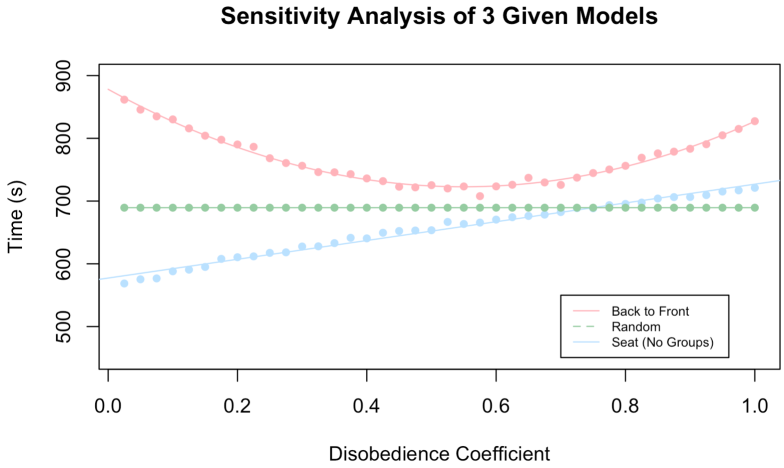


Figure 2.8: Sensitivity analysis of the disobedience coefficient on the interval $0 \leq \psi \leq 1$

Fig 2.8 shows the impact of changing the disobedience coefficient on the time taken to board, for the three given models. The effect of changing the disobedience coefficient for the section boarding was most interesting. As the number of people not following the prescribed method increased, the boarding method trended towards random. This meant the time taken decreased as random boarding is faster than section boarding. At a disobedience coefficient of $\psi = 0.5$, the boarding method is effectively random, thus the times are equivalent. However, as more people decide not to board with their prescribed group, the time starts to increase again. This is due to the boarding becoming 'ordered' again by section, which is slower than a random boarding method. This behaviour from the boarding by section method is ideal for airline companies, as a realistic extent of disobedience will help their boarding times. The random boarding method is completely insensitive to changes in disobedience, as there are no rules to disobey. The boarding by seat method without groups is the fastest boarding method provided, but it is also the method most impacted by changes in disobedience. This is potentially undesirable behaviour in a boarding method for airline companies, however under all reasonable values of the disobedience coefficient, boarding by seat is the fastest boarding method.

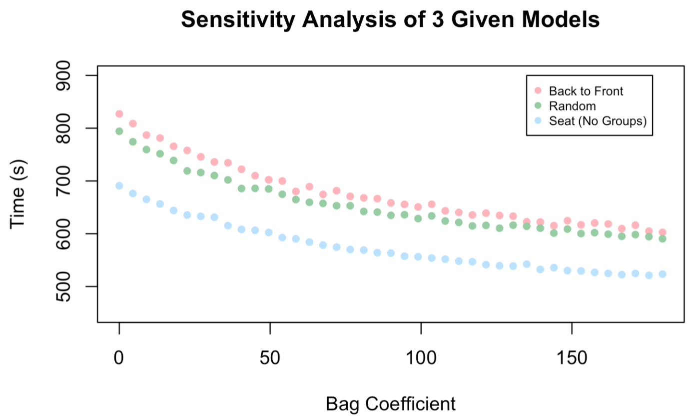


Figure 2.9: Sensitivity analysis of provided boarding methods by scaling a part of the bag coefficient.

Changing the bag coefficient changes the number of people without bags. The higher the coefficient, the higher the number of people without bags. The relevant time relating to bag numbers is the time spent in the aisle stowing. As such, only the number of bags stowed is pertinent to this model. Therefore, our variation of the bag coefficient (integers from 0 to 180) is effective at describing the impact of all plausible variations in bag numbers and bag stowage on the time take to board an aircraft. From this analysis, we found that the three recommended methods are of equal sensitivity to variations in the bag coefficient. This is shown by the identical shape of the curves.

2.4 Modelled Results for Other Boarding Methods

2.4.1 Modified Steffen Method

The Steffen method is a plane boarding method proposed by Jason Steffen in 2008 which is suggested to be the method that produces the optimal plane boarding time[6]. However, this method is highly theoretical. It relies on the unrealistic assumption that passengers are efficient and highly organised. Instead, we present the modified version of the Steffen method which has a slightly larger grounding in reality. This method boards even numbered rows on the right hand side, then even rows on the left, then odd rows on the right, to odd rows on the left. This was almost the fastest boarding method we tested, with a mean time to board of 647.05 seconds. The 5th percentile was 595.3 seconds, and the 95th percentile was 696.5 seconds (a spread of 101.2 seconds).

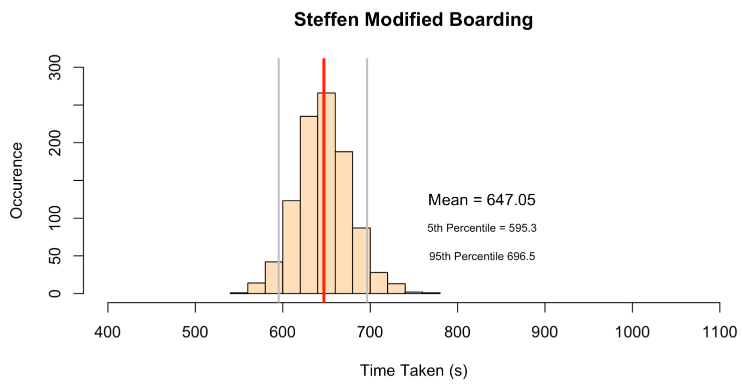


Figure 2.10: Monte Carlo simulation graph of the modified Steffen method

2.4.2 Prioritised Groups

In this method, passengers are classified as having one of two classes of walking speeds: normal and slow. This removes the need for our initial assumption that walking speed is relatively constant and allows us to test the validity of this assumption. Many airlines allow prioritised groups such as families with young children, disabled and elderly people to board first. The passengers in these prioritised groups are classified as having slow walking speed. We run this method through our model to determine its efficacy.

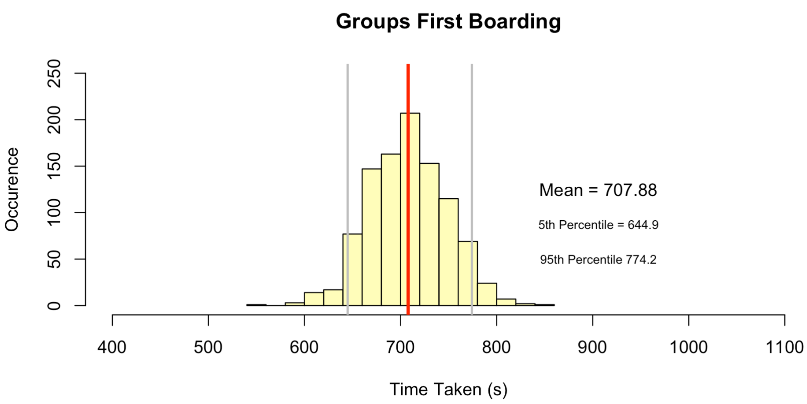


Figure 2.11: Monte Carlo simulation graph of the prioritised group boarding method

2.4.3 Modified Boarding by Seats (WMA)

As mentioned, the WMA has some serious drawbacks, particularly in regard to the splitting of groups. To overcome this, we devised a modified WMA method, which is one of our additional boarding methods. In this seating method, window seats are boarded first. However, if someone with a window seat is also part of a group, that whole group will board. The same thing occurs for middle seats and aisle seats. This avoids the problem of splitting groups while maintaining some of the efficiency of the WMA method.

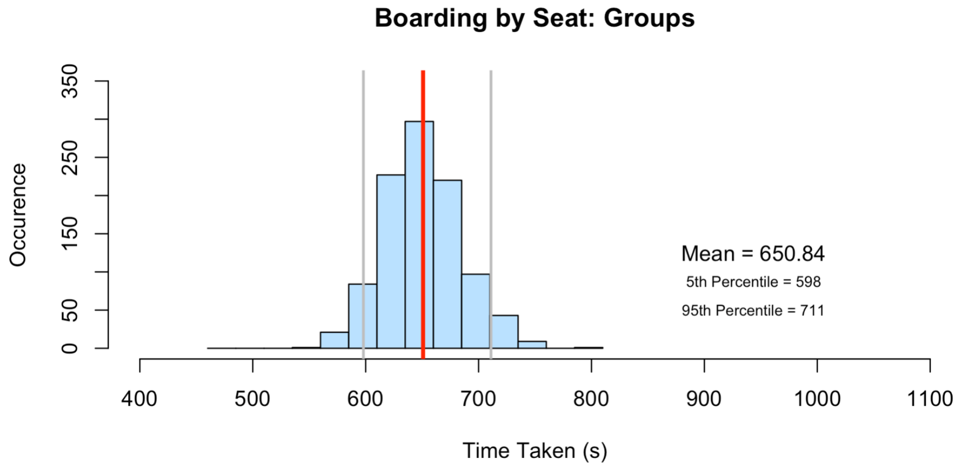


Figure 2.12: Monte Carlo simulation graph of the modified boarding by seats (WMA) method

The mean boarding time we obtained from this method was 650.84 seconds, with a 5th percentile of 598s and a 95th percentile of 711s (spread of 113 seconds). This adjusted method is relatively novel and hasn't seen much discussion in literature. However, its unique combination of practical and theoretical efficiency makes it an attractive proposition.

2.5 Optimal Boarding Method

After analysis of the previous five methods, we conclude that the modified WMA method is the best. The mean time to board the narrow body plane after 1,000 trials is 650.84 seconds. It should be noted that this isn't the optimal time that was achieved; the modified Steffen took only 647.05 seconds, and WMA without groups took 519.13s. This data is summarised in Fig 2.13. However, the modified WMA is significantly more practical to implement than both. The modified Steffen requires an unrealistic degree of coordination from random passengers and WMA without groups has the unrealistic assumption of splitting families and other groups apart. The modified WMA method allows for groups and can be easily implemented by airlines (by just calling seat letters to board, including family groups). It is also less sensitive to changes in the disobedience coefficient than alternative methods, such the Steffen modified. Although the time to board is initially slightly faster in the Steffen modified, as the disobedience coefficient increases, the time to board from the Steffen method increases faster than the time to board from the modified WMA. This is advantageous, as it means there is likely less variation in this modified WMA model in comparison to similarly fast boarding methods, allowing airline companies to better predict the boarding times.

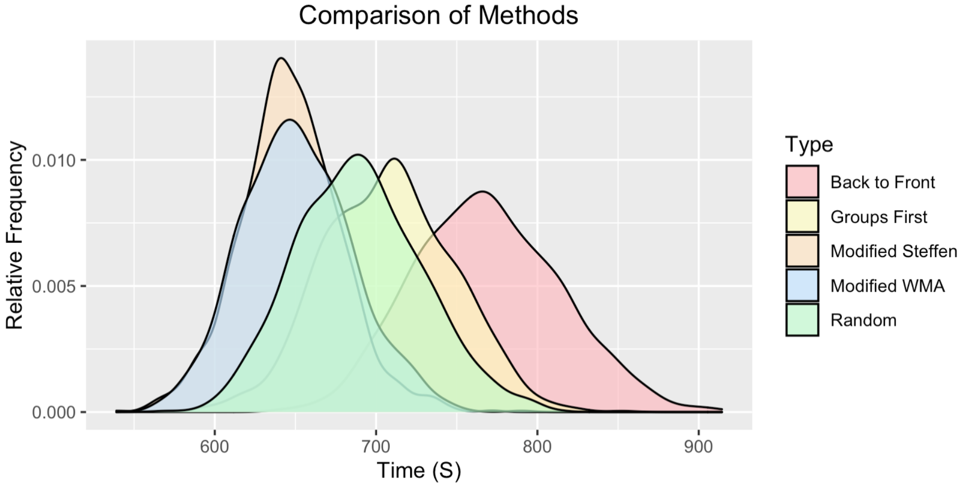


Figure 2.13: Comparison of Monte Carlo simulation graphs of different boarding models featured in previous sections

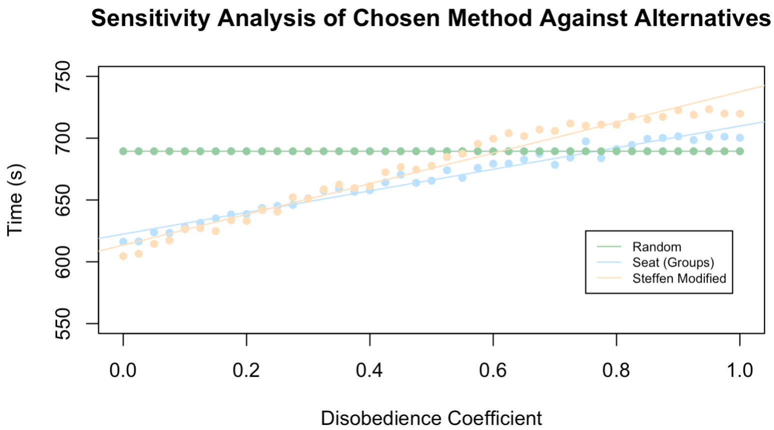


Figure 2.14: Sensitivity analysis of chosen methods for disobedience coefficient

3 Narrow-Body Disembarking

Having run simulations on our model under different boarding methods, we now turn our attention to the problem of disembarking. When exiting a plane, people typically move towards the nearest exit whenever space becomes available. A simulation of this is the basis of our disembarking model. By modelling individual interactions, such as what happens when two people come into the same space, we were able to ensure that our model was true as possible to a real disembarking.

3.1 Generation of Priority Map

The disembarking model runs through the generation of a priority map. Each person/group is assigned a priority value, representing how much they want to leave the plane. This is realistic since some people are desperate to leave and others being happy to sit on the plane until the rush dies down. This value is used when there is a passenger interaction. The priority values of each passenger that can move into the square are compared, and the passenger with highest priority is given the right of way. This map can also be manipulated to get different disembarking methods. By giving the highest priority to passengers we want to leave first, we can manipulate the order of who leaves first to find an optimal disembarking method. As such, different methods call for different priority maps. The creation of the priority maps begins with the creation of an ideal priority map. In this map everyone would be assigned values such that they'd disembark in the desired fashion. Fig 3.1 shows an ideal priority map for disembarking by row, back to front, in the narrow body aircraft.

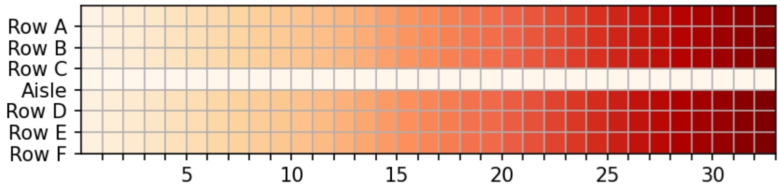


Figure 3.1: Ideal priority heatmap of back to front disembarking

In practice it is highly unlikely that everyone would follow a perfect disembarking model and therefore a disobedience coefficient was implemented, similar to the boarding model. The value of the disobedience coefficient was increased from 0.3 (in the boarding model) to 0.4 in this disembarking model. This choice was based on the fact that people are more likely to be tired, and may just want to leave the plane as soon as possible following a long flight. There is no feasible way to obtain data for this particular coefficient, and to investigate the effect this coefficient has on boarding times we performed a sensitivity analysis, varying the disobedience coefficient. Like in the boarding model, the disobedience coefficient describes the chance that a particular person won't follow their prescribed disembarking method. These disobedient people are then randomly assigned a new priority value ranging from 1 to the maximum possible priority value which varies depending on method. An implementation of this on the previously given priority map can be seen in Fig 3.2.

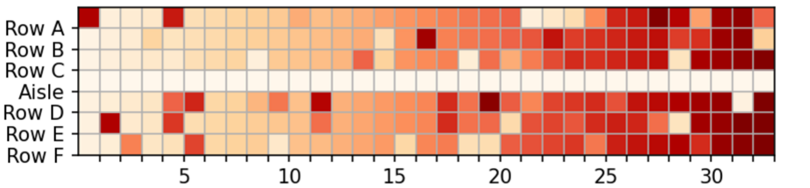


Figure 3.2: Introduction of disobedience coefficient to the ideal heatmap in Fig 3.1

As in the boarding method, we accounted for the fact that many people travel in groups that cannot be split. To implement this in the model, the priority of a group of size n is set to the mean of each member's priority in that group like so: $P_{group} = \frac{1}{n}\sum_{i=1}^{n} P_i$. The effect of this can be seen in Fig 3.3. Note the group in row 32 (seats ABC).

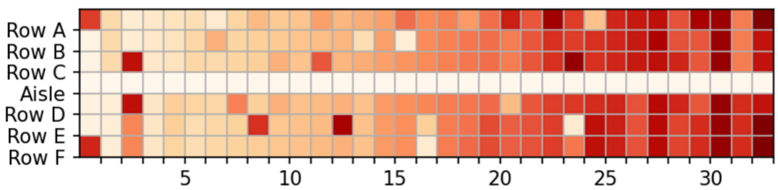


Figure 3.3: Introduction of groups to the heatmap in Fig 3.3

3.2 Logic of Disembarking

The diagram to the right shows the logic of the disembarking. By looping through the unoccupied aisle spots, and moving individuals into them, we can simulate the whole moving out. We considered the movement in and out of aisles as well.

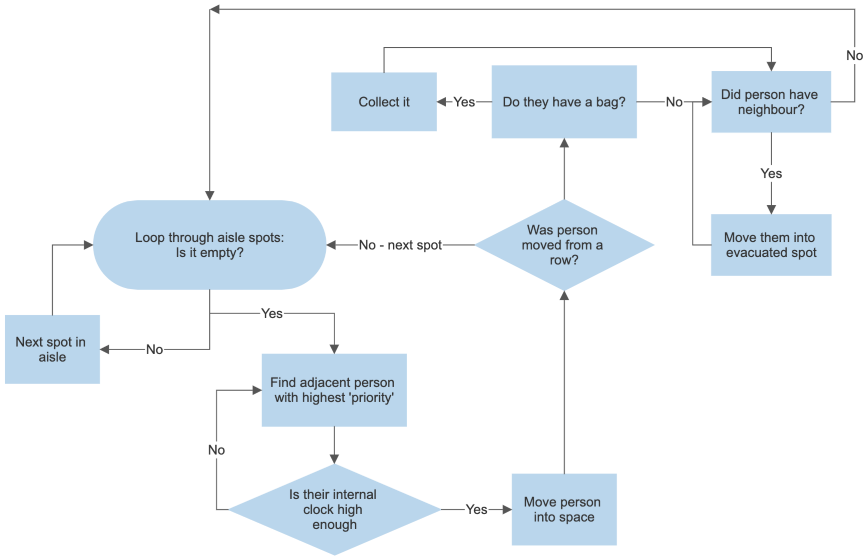


Figure 3.4: A flow diagram of the disembarking algorithm from a passenger's perspective

3.3 Time to Unstow Bags

Just as stowing bags during boarding blocks the aisle, the act of unstowing bags during disembarking blocks the aisle too. The following formula is a variant of Eq 1 that simply changes $n'_{bins} = n_{bins} - 2$. This is done to avoid division by zero, since many overhead bins will have $n_{bags} = 6$ as they are full. Eq 1 accounts for the number of bags already in bins – it takes longer difficult to remove a bag out of a packed luggage bin than an empty one. Note this n'_{bins} is simply labelled n_{bins} in Eq 1.

$$T_{bags}(n_{bags}, n_{bins}, 6) = \begin{cases} 0 & \text{if } n_{bags} = 0 \\ \dfrac{4}{1 - 0.8(n_{bins} - 1)/6} & \text{if } n_{bags} = 1 \\ \dfrac{4}{1 - 0.8(n_{bins} - 2)/6} + \dfrac{2.25}{1 - (n_{bins} - 1)/6} & \text{if } n_{bags} = 2 \end{cases} \tag{4}$$

3.4 Optimal Disembarking Method

The optimal disembarking method for the narrow body aircraft was found to be disembarking from back to front by row. This was initially surprising. However further analysis suggested it to be the quickest due to it having the greatest aisle flow out of all methods. The rate of free aisle flow hindered by retrieving baggage determines the rate people can enter the aisles and hence leave the plane. Back to front results in the greatest aisle flow due to people feeding into the aisles from the back of the plane. Should they need to retrieve a bag, they **a)** hold very few people up as they are at near the end of the queue, and therefore hold very few people up and **b)** by them stopping, they allow people in front of them flow into the queue meaning no gaps are left open.

This is opposite to the 'front to back' boarding method which is employed by most airlines and is the slowest disembarking method. This is because when someone enters the aisle from the front

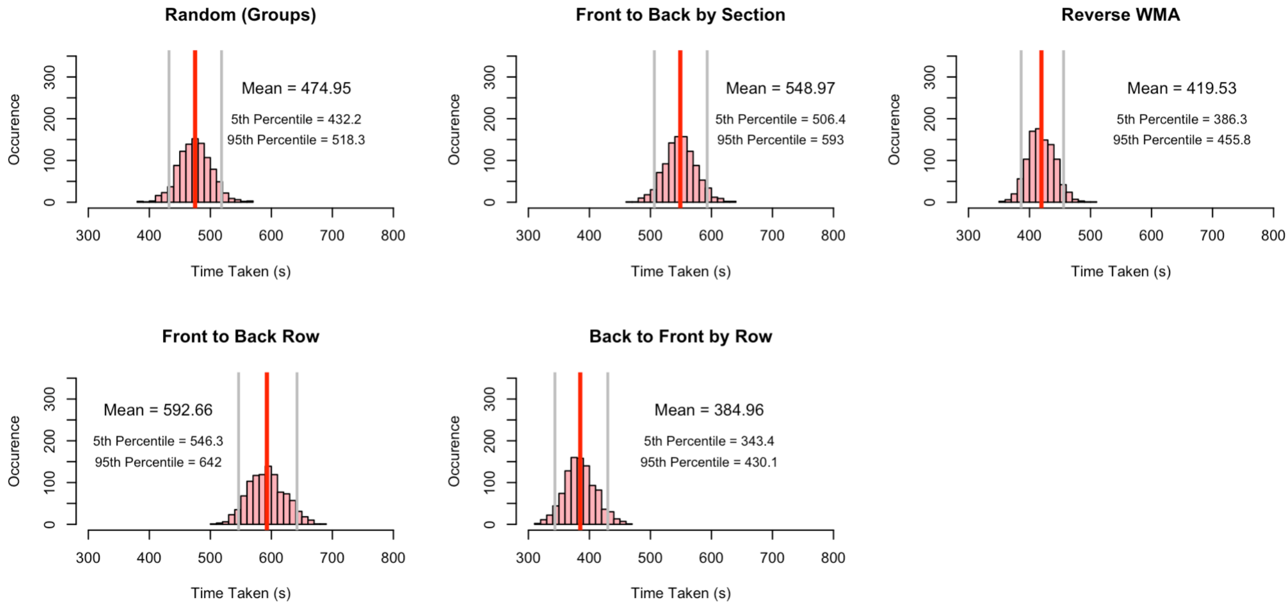


Figure 3.5: Monte Carlo simulation graphs of various disembarking methods (1k trials)

of the aircraft and retrieve their bag they hold the whole queue up whilst not allowing anyone in front of them to enter the queue as there is no one else in front.This back to front system would rely on a 'right of way' approach to disembarking the plane, where people at the back have the highest priority. When there is a space that two people could move into, the passenger at the front would have to give way to the passenger coming from behind them.

Other notable disembarking methods were the Reversed WMA which grants priority to aisle-seat passengers followed by middle then window seat. The Reversed WMA produces slower time than Back to Front as more people enter the queue near the front of the aircraft and thus block the queue as they retrieve bags. Reversed WMA is also impractical to implement as it requires a large degree of coordination in comparison to the relatively simple Back to Front method which is a reverse of the commonly used Front to Back Disembarking.

Disobedience Coefficient	**Reverse WMA**	**Back to Front**	**Random**
0	393	231	474.3
0.2	421.7	355	474.3
0.4	438.2	404.2	476.1
0.6	459.4	435.7	474.9
0.8	470.2	451	474.2
1.0	474.9	474.1	474.7

Table 1: Sensitivity analysis of disembarking methods by scaling the disobedience coefficient

This table shows that as disobedience increases, the time taken to disembark decreases. At no disobedience, we get fast disembarking times for reverse Wilma and back to front and at the maximum disobedience we see the times being similar to a random boarding time. Importantly this table also reveals that these models are very sensitive to disobedience especially back to front between 0 – 0.4. It is important to note that despite back to front disobedience sensitive nature it still remains the quickest at the assumed disobedience coefficent of 0.4. This also suggests the importance of airlines employing methods to increase obedience when disembarking as a 20% reduction in disobedience could cause up two minutes in extreme cases.

People Not Retrieving Bags	Reverse WMA	Back to Front	Random
0.2	411.9	374.7	468.2
0.4	352.7	325.3	352.7
0.6	289.9	261.0	309.8
0.8	234.1	215.7	235.6
1.0	200.1	200.9	200.0

Table 2: Sensitivity analysis of disembarking methods by changing people not retrieving bags

Table 2 shows a steady trend where the boarding time decreases and tends towards a constant time of 200s as the people not retrieving bags increases. This trend is important for two reasons. Firstly, it shows that if airlines could reduce the amount of bags carried it would result in much faster disembarking times, to the point it would no longer matter which disembarking method was employed. This is because less bags mean the aisle is blocked for a reduced amount of time. Even a minor increase in people not taking bags, for example from 40% to 60%, would result in a drastic reduction in disembarking time of 30s. This could be achieved by encouraging passengers to retrieve their bag in the period between when the plane lands and the disembarking process begins thus increasing the amount of people not retrieving during the disembarking.

4 Extension of Model to Other Aircraft

4.1 Flying Wing Aircraft

4.1.1 Flying Wing Boarding

The Flying Wing Aircraft has a revolutionary seating plan with an additional 3 aisles and 18 seats across, but only 14 rows. To account for this, we built upon the core algorithm of the narrow body in which passengers walk down the aisle, by simulating all 4 aisles at once, with an additional aisle connecting all of these at the top from the entrance. We initially considered simulating only one aisle and simply quartering the flow rate into the aisle. However, this is not realistic as the top aisle can still be blocked – for example, consider the case where a passenger is stowing their luggage in row 1 of the first aisle, whilst a passenger behind them waits to get into this aisle. Keeping with the assumption that only one passenger can fit into an aisle, such a scenario would block passengers from accessing all other aisles, increasing total boarding time. Thus, we must simulate all aisles boarding at once. Furthermore, although the number of aisles in this plane may cause confusion about where to go, we assumed that this would already be accounted for by the presence of flight attendants, causing no passengers to walk down the wrong aisle.The extended algorithm as experienced by a passenger is represented in the flow chart. A visualization of this model nearing completion is also displayed. Note: the top aisle is not included in this visualisation.

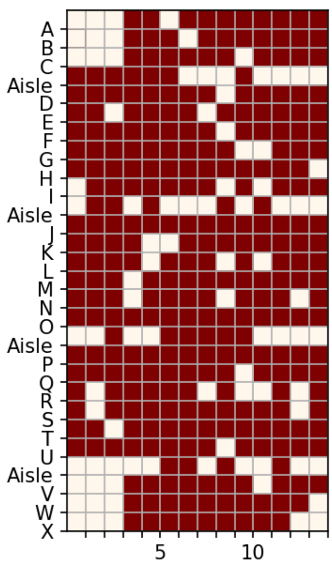


Figure 4.1: Flying Wing model

Different boarding models can be applied to the flying wing aircraft to different effect. Random and sectional boarding are relatively easily implemented, and both would be theoretically and practically effective. However, our optimal boarding method for the narrow body aircraft, the WMA method, is now rendered impractical to implement. When considering a seat block between two aisles, where 'A', 'M', and 'W' represent aisle, middle, and window seats respectively. Translating into rows six seats wide, you get the pattern A—M—W—W—M—A. It would be impractical for passengers to judge whether their seat is designated as A, M or W, even without incorporating groups.

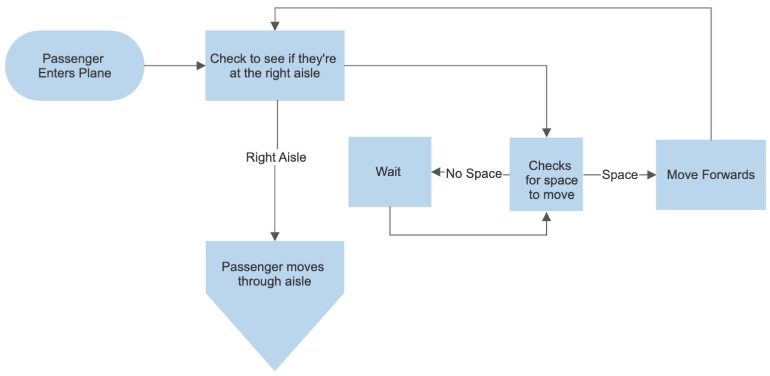


Figure 4.2: Flow diagram of passenger movement logic for the flying wing aircraft

Another potential boarding method we could adapt to a wide wing aircraft would be the modified Steffen method. However, given the established impracticality of the modified Steffen method on the narrow body, this would be even less realistic to expect passengers to follow it when the aircraft is boasting multiple aisles. Thus, we disregarded this method for this plane type. This left us with two viable boarding methods for the wing plane. These are shown in Fig 4.3, along with adjusted WMA times. The mean result times from these boarding methods were 593.2 seconds for the back to front time, 558.9 seconds for the random boarding, and 546.1 seconds for the adjusted WMA boarding time.Despite this, the optimal boarding method is the section boarding, from back to front. Despite having the lowest times, the impracticality of other solutions makes it the most attractive. The closest in terms of overall effectiveness would be random boarding. However, the organisational issues of trying to queue all the passengers in a random order with resulting in excessively large queues would more than out weigh the megre 34.3 second boarding time advantage.

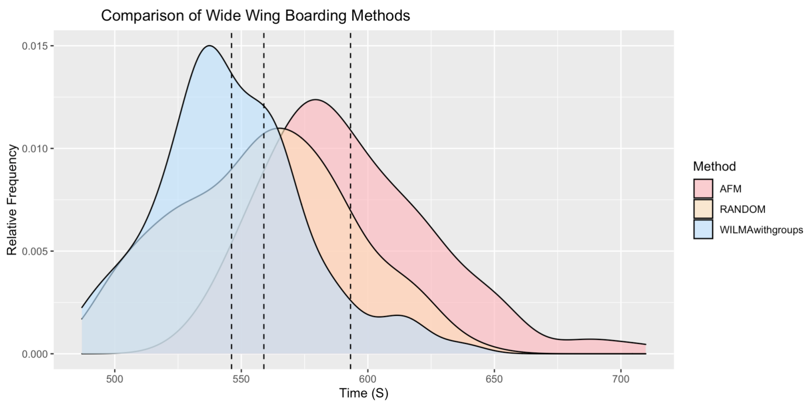
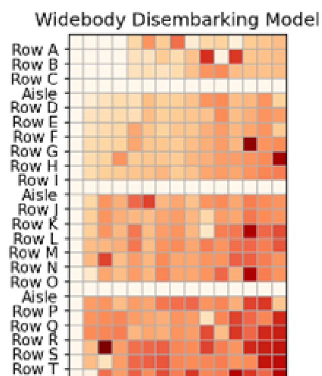


Figure 4.3: Comparison of distribution of times for different methods on the Flying Wing aircraft. Note that the mean lines for AFM, random and WMA are located at 546.1, 558.9 and 593.2 respectively.

4.1.2 Flying Wing Disembarking

The core logic of the priority disembarking model remains the same when adapted to the Flying Wing Aircraft. The plane is broken into four sub-aisles that passengers are able to move into, dependent on the priority logic. Additionally a leaving aisle has been added that runs by all of the sub aisles. Passengers are moved into and along this leaving aisle to the exit through the use of the priority logic.



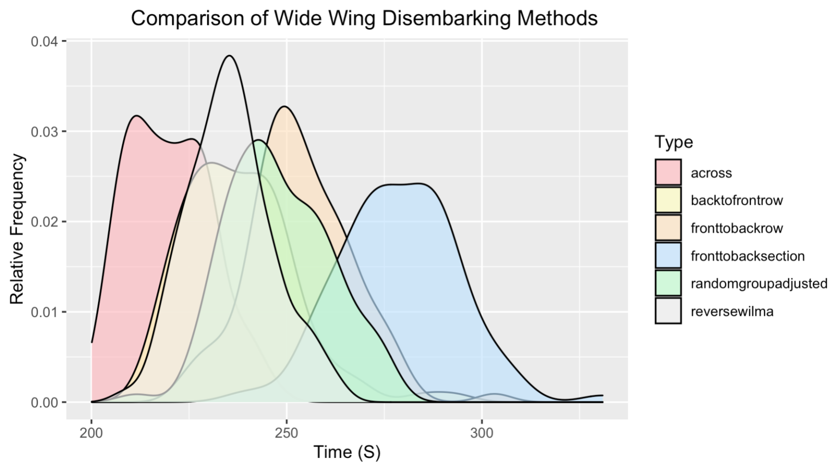


Figure 4.5: Comparison of disembarking methods on the flying wing aircraft

Similarly to the narrow body disembarking, models that prioritise aisle flow will be the fastest in the flying wing aircraft. The most successful model was the across method in Fig 4.4. The across method increased the priority assigned to passengers the further they are from the door, with respect to row and seat. This is in comparison to the 'front to back' disembarking method that only increases priority across the row. The across method of distributing priority resulted in the quickest disembarking times as it systematically emptied the aircraft from the bottom right to top left of the diagram (with the exit in the top left). This allowed most people to enter the aisle when they are few people behind them (as the people behind have already left). This means there are minimal aisle blockages when passengers retrieve bags, and passengers in front of the blockage will be able to move into the aisle. The across method is quite practical as it just requires for people to wait for the person behind to leave, or the person behind to retrieve a bag hence allowing them to leave. Similarly to other disembarking models, this is a form of 'right of way system', where people furthest from the door have right of way.

4.2 Two-Entrance Two-Aisle Aircraft

4.2.1 Two-Entrance Two-Aisle Boarding

The two-entrance two-aisle aircraft adds the complexity of multiple entrances, as well as a first-class section. However, we made the following assumptions:

- The first-class section would board first, as is standard across airlines. Any late passengers would interfere negligibly with the rest of the boarding process, as they do not walk down the same aisles as the rest of the passengers
- Rows 12 to 26 (and first-class) would board from the front entrance, whereas rows 27 to 47 would board from the back
- All passengers would board from the correct entrance. In a similar reasoning to everybody walking down the right aisle, we assumed that a plane of this size – and especially one with first class – would have sufficient flight attendants to ensure that this did not happen.

Given these assumptions, a valid simplification can be made to the model: the total boarding time would simply be the time taken to board first class, added to the greatest boarding time of the two sections (seats accessed from entrance 1 and seats accessed from entrance 2) of the plane. Boarding methods in first class are unreasonable to implement in real life given the smaller number of seats, but also the easier access to seats due to greater space. Thus, this time was calculated using the random boarding method with increased speeds for walking and stowing luggage. The average was found to beWe again tested this plane with unstructured (random) and sectional methods and our proposed method of Wilma with groups (in this case, the middle seat does not exist). Once again, Wilma with groups was found to be the most effective method.

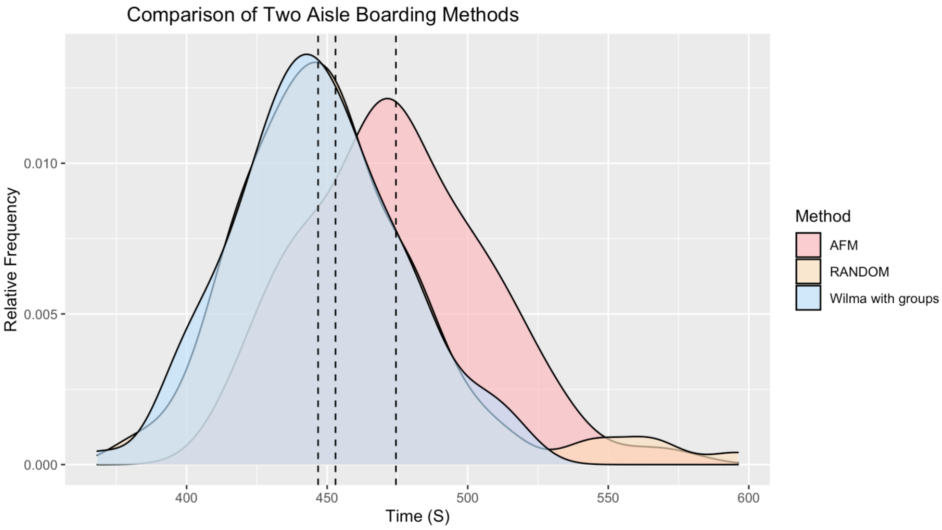


Figure 4.6: Comparison of boarding methods on the two-entrance two-aisle aircraft

4.2.2 Two-Entrance Two-Aisle Disembarking

The assumptions made in the boarding model of Two Entrance Two Aisle can be carried across to the disembarking model. Consequently, simulation were run on both halves of the plane for disembarking and the times were added to the time taken to board first class. Although disembarking occurs on two aisles in this model, aisle flow is still valued and therefore models that increase aisle flow such as back to front is still fastest with a mean time of $180s$.

5 Pandemic Capacity Decrease

The COVID-19 pandemic has introduced many additional barriers to air travel. Notably, the passenger capacity of aircrafts is forced to decrease to help combat the spread of the disease. We test and present the effect on embarking and disembarking the three aircraft types when passenger capacity is limited to 70%, 50% and 30%. We ran 1000 test cases for all aircraft at all capacity levels with the three most optimal methods, the averages for which can be found in the appendix.

5.1 Boarding

An important consideration is that it is not simply random what tickets are not for sale on an aircraft with reduced capacity. Instead, they are chosen to maximise social distancing. For capacity c each row with numbers of seats s, we allowed a maximum of $\lceil c \times s \rceil$ seats to be filled in that row (where $\lceil x \rceil$ is the ceiling function), and reduced passengers randomly from there on until the number of passengers reached c. However, groups are still allowed to sit with each other. An analysis of the data would suggest that for both the Narrow Body and Flying Wing aircraft, the Wilma with groups method remains preferable up until a capacity of 50%. At this value its efficiency is only marginal. However, beyond this it becomes optimal to board by section (aft then middle then front). This can be explained in practice, as at lower numbers of passengers, there is lesser chance of someone blocking the way to a seat – the main problem with sectional boarding. Therefore, without this problem, filling up from the back allows the most passengers to enter the queue at once, resulting in being more optimal. For the Two Entrance Two Aisle aircraft, sectional boarding also quickly becomes the most This method has the added benefit of splitting passengers into boarding groups that also sit together, meaning that although contact cannot be completely avoided, it is minimized to be with the same people. If the pandemic is at the point that capacities of 50% or below need to be enforced, then this is a valuable aspect. Therefore, we would recommend this method for capacities of 50% and below on all aircraft. For capacities of 70% and 100%, the original recommended method remains the most optimal (this is still sectional for the Two Entrance Two Aisle aircraft).

5.2 Disembarking

To model reduced capacity of disembarking, we assumed a similar dispersal of passengers as in boarding. For all three aircraft, back to front remained the quickest method of disembarking no matter the capacity. Unfortunately, this does not preserve the social distancing between differing sections of the plane as achieved by boarding. However, at low capacities, disembarking times between different methods became exceedingly quick (under 2 minutes) and closer together. It would be no huge cost to the aircraft to favour a slower method at these capacities. Thus, at low capacities which aim to contain Covid, we would recommend a front to back method. Although this has not been modelled, by extrapolating data for modelled methods, it is clear that this would still be done in a tight timeframe.

6 Evaluation of Models

6.1 Strengths

- Adaptable. Not only can our models be used for many different boarding and disembarking methods, but our models can be adapted to wide range of plane shapes and sizes, that could consist of multiple aisles or entrances, with relative ease.

- Comprehensive. Our models take into account a wide range of factors affecting boarding and disembarking times, such as people moving past each other within rows, or time taken to stow and retrieve luggage. These times are calculated using real life data, ensuring the highest accuracy.

- Realistic. Many online models may have the strengths above, but fail to account for many common behaviours, such as people disobeying boarding instructions and passengers travelling together in groups.

6.2 Limitations

- Memory intensive. Due to boarding/disembarking being a stochastic process, a large number of test cases are needed to obtain an accurate average for any given scenario. Our model is bulky.

- Large number of assumptions made. For ease of simulation, we assumed many things, ranging from a constant walking speed down the aisle, or that passengers would always sit in the correct seat or walk down the correct aisle to their seat. In reality, this will not always be the case. To improve our model, we could include these factors in our simulation.

Letter to Executive

Dear Airline Executive,

Through analysis of boarding and disembarking methods, our team has developed a model that has allowed us to determine the optimal boarding and disembarking methods for a variety of different aircraft, with a number of different restrictions.

One consideration when deciding boarding methods is the impact that it will have on family groups. It is vital that we avoid splitting these groups when boarding to ensure all passengers have a positive travelling experience with you. Another factor that affects loading times is the number of passengers that don't follow boarding methods.

We found that three main factors that contribute to the time take to board an aircraft. The first of these is the walking speed of the passengers. However, there is not much that can realistically be done about this. Secondly, there is the time taken to stow overhead luggage. While passengers are doing this, they are blocking the aisle. Another aisle blockage comes passengers try to get to seats that are blocked by other passengers in same row.

These impact of these aisle blockages can be minimised by the chosen boarding method, and also by several different techniques. These include ensuring that people follow the boarding method (potentially through regulation from air stewards), and also by making more easily accessible overhead storage, to minimise time spent retrieving stowed luggage.

That being said, method choice is an easy was to immediately speed up passenger boarding/disembarking. In a standard narrow body aircraft, passengers should be boarded with the adjusted WMA method (window seats board first, followed by middle seats and then aisle seats, but groups board together), and should disembark giving the right of way to passengers coming from the back. Both methods minimise aisle blockages, and allow optimal aisle flow.

In a wide wing aircraft, the optimal boarding method is by section (back to front). This fastest method that can be practically implemented. To disembark, we recommend the 'across' method, similar the method for a narrow aircraft, where passengers furthest from the door get right of way. For the 2 aisle, 2 entrance aircraft, the recommended methods are the same as the narrow body: adjust WMA for boarding and back to front for disembarking. We hope these recommendations and explanations will aid you in running your airline, and look forward to your feedback.

7 Appendices

References

[1] E. Mazareanu. *Number of flights performed by the global airline industry from 2004 to 2022.* 2022. URL: https://www.statista.com/statistics/564769/airline-industry-number-of-flights/ (visited on 03/28/2022).

[2] Suzanne Hiemstra-Van Mastrigt, Richard Ottens, and Peter Vink. "Identifying bottlenecks and designing ideas and solutions for improving aircraft passengers' experience during boarding and disembarking". In: *Applied Ergonomics* 77 (2019), pp. 16–21.

[3] Highskyflying. *How Wide Are Airplane Aisles?* 2022. URL: https://www.highskyflying.com/how-wide-are-airplane-aisles/ (visited on 03/29/2022).

[4] Kathryn Watson. *What's an Average Shoulder Width?* 2018. URL: https://www.healthline.com/health/average-shoulder-width (visited on 03/30/2022).

[5] Air New Zealand. *Carry-on Bags.* URL: https://www.airnewzealand.co.nz/carry-on-baggage (visited on 03/29/2022).

[6] Jason H. Steffen. "Optimal boarding method for airline passengers". In: *Journal of Air Transport Management* 14.3 (May 2008), pp. 146–150. DOI: 10.1016/j.jairtraman.2008.03.003. URL: https://doi.org/10.1016%2Fj.jairtraman.2008.03.003.

YouTube videos used for analysis throughout the report
https://www.youtube.com/watch?v=SnAYtU3nc0g
https://www.youtube.com/watch?v=F4Dt2-kc5Mw
https://www.youtube.com/watch?v=OCCUgcb4LvE
https://www.youtube.com/watch?v=V3RXT0D0CnM
https://www.youtube.com/watch?v=jv4y4ir07Nc
https://www.youtube.com/watch?v=AjSFc3yLA9s
https://www.youtube.com/watch?v=a6oBGKEdYxc
https://www.youtube.com/watch?v=perToEokKR0
https://www.youtube.com/watch?v=godZCdvG8CI
https://www.youtube.com/watch?v=mrcQL5Od-Fg

Appendix A

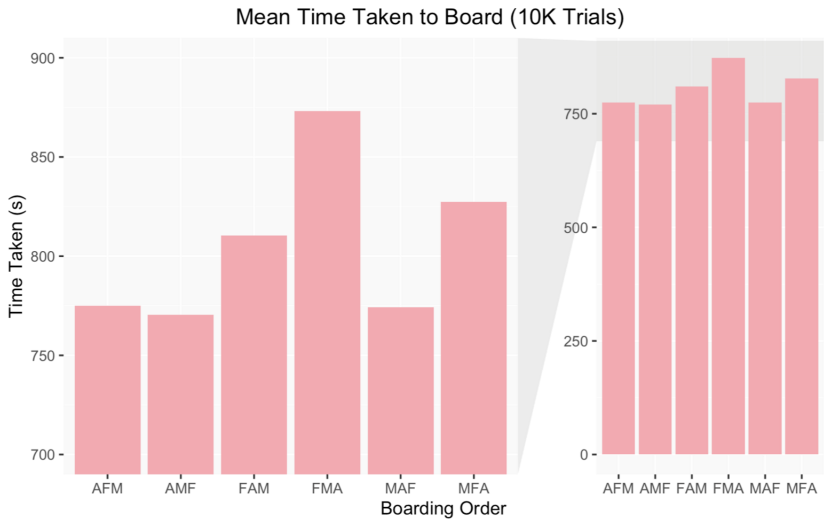


Figure 7.1: Bar chart of mean boarding times for different boarding by section combinations. The labels are in order of section boarding (e.g. AFM means aft first, front second, middle last. Note too that the chart on the left is an enlarged section of the chart on the right.

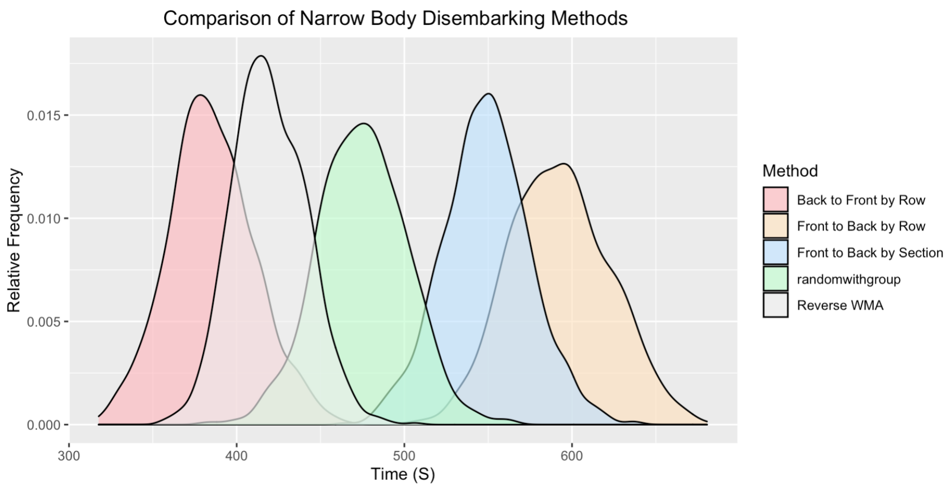


Figure 7.2: Comparison of disembarking methods for narrow body aircraft

Appendix B

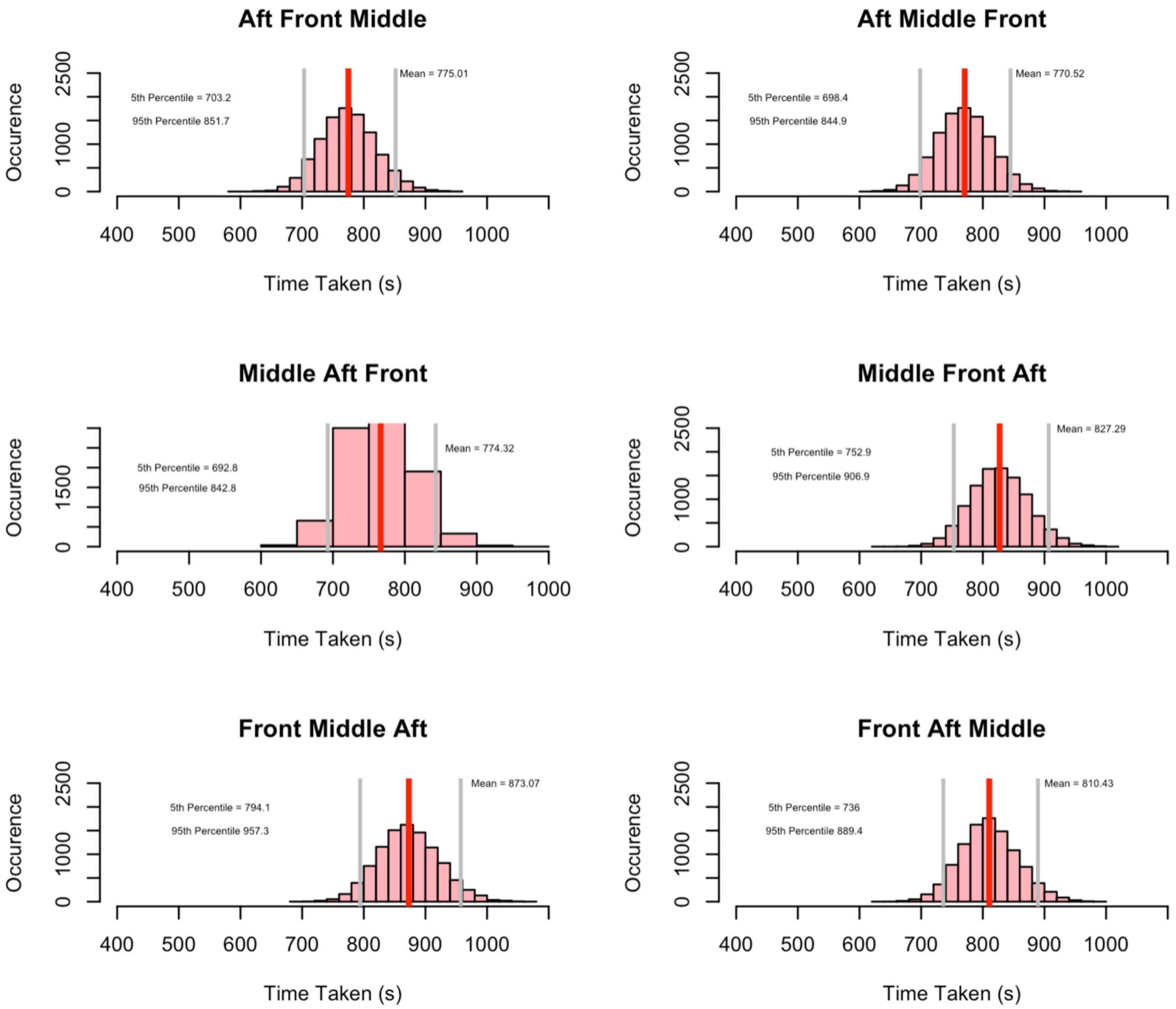


Figure 7.3: Monte Carlo simulation graphs of boarding by sections method in different orders. For instance, Aft Front Middle means first the aft is boarded, then front, and finally the middle.

Appendix C

			Narrow Body	Time taken depending on different boarding methods (s)
Capacity of plane	Wilma with groups	Random Boarding	Section boarding	
100%	644	700	764	
70%	545	550	600	
50%	301	311	302	
30%	194	202	180	

Flying Wing	Time taken depending on different boarding methods (s)		
Capacity of plane	Wilma with groups	Random Boarding	Section boarding
100%	493	552	575
70%	332	364	383
50%	237	263	254
30%	189	187	177

2E2A			
Capacity of plane	Wilma with groups	Random Boarding	Section boarding
100%	453	456	454
70%	319	302	292
50%	240	238	222
30%	179	177	152

COVID TALES DISEMBARKING

Narrow Body	Time taken depending on different boarding methods (s)		
Capacity of plane	Back to front	Reverse Wilma	Random
100%	384	419	474
70%	275	302	308
50%	128	224	220
30%	106	139	132

Flying Wing	Time taken depending on different boarding methods (s)		
Capacity of plane	Across (modified Back to Front)	Reverse Wilma	Random
100%	221	239	248
70%	167	160	178
50%	111	125	124
30%	95	97	106

2E2A	Time taken depending on different boarding methods (s)		
Capacity of plane	Back to front	Reverse Wilma	Random
100%	180	198	209
70%	142	168	192
50%	104	154	142
30%	87	92	104

Appendix D

Code used in R for statistical analysis

```
# INSTALL AND LOAD PACKAGES ################################

library(datasets)  # Load/unload base packages manually

################# Graph Function ###################
make_histogram <- function(x,colour, title,labelpos,meanpos,breaks1) {
  quantile5 <- c(quantile(x, probs = 0.05))
  quantile95 <- c(quantile(x, probs = 0.95))
  mean <- mean(x)
  hist(x,
```

```
        xlim=c(400,1100),
        ylim=c(0,350),
        col=colour,
        breaks = breaks1,
        xlab="Time Taken (s)",
        ylab="Occurence",
        main=title)
  # Add lines for mean, 5%, and 95%
  abline(v = mean,
         col = "red",
         lwd = 3)
  abline(v = c(quantile5, quantile95),
         col = "grey",
         lwd = 2)
  # Add labels
  text(x = mean*meanpos,
       y = 125,
       paste("Mean =", round(mean,2)),
       col = "black",
       cex = 1)
  text(x = mean*labelpos,
       y = 87.5,
       paste("5th Percentile =", round(quantile5,1)),
       col = "black",
       cex = 0.7)
  text(x = mean*labelpos,
       y = 50,
       paste("95th Percentile =", round(quantile95, 1)),
       col = "black",
       cex = 0.7)
}
dev.off()
#function(x,colour, title,labelpos,meanpos)
###########Finding best and worst boarding by sections ############
par(mfrow = c(3,2))
make_histogram(sectionsdata$afm, "#ffb3ba",
               "Aft Front Middle",
               0.65, 1.18 )
make_histogram(sectionsdata$amf, "#ffb3ba",
               "Aft Middle Front",
               0.65, 1.18 )
make_histogram(sectionsdata$maf, "#ffb3ba",
               "Middle Aft Front",
               0.65, 1.18 )
make_histogram(sectionsdata$mfa, "#ffb3ba",
               "Middle Front Aft",
               0.65, 1.18 )
make_histogram(sectionsdata$fma, "#ffb3ba",
               "Front Middle Aft",
               0.65, 1.18 )
make_histogram(sectionsdata$fam, "#ffb3ba",
               "Front Aft Middle",
               0.65, 1.18 )
fma<-mean(sectionsdata$fma)
#Compare with groups to without gorups
dev.off()
#function(x,colour, title,labelpos,meanpos)
#Compare four boarding methods: Random, Seats, Best Section, Worst Section
par(mfrow = c(1,2))
breaks2<-rep(10,1)
```

```
make_histogram(dataogmethod$Random, "#baffc9",
               "Random Boarding",
               1.33, 1.33 )
make_histogram(dataogmethod$Seat, "#bae1ff",
               "Boarding by Seat: Groups",
               1.45, 1.45, c
   (460,485,510,535,560,585,610,635,660,685,710,735,760,785,810) )
make_histogram(dataogmethod$fma, "#ffb3ba",
               "Front to Back",
               0.65, 0.65 )
make_histogram(dataogmethod$amf, "#ffb3ba",
               "Back to Front",
               0.7, 0.7 )
make_histogram(dataogmethod$Seats.No.Groups, "#bae1ff",
               "Boarding by Seat: No Groups",
               1.6, 1.6, c(420,440,460,480,500,520,540,560,580,600))
dev.off()
make_histogram(groupsfirst$prioritize_groups_boarding, "#ffffba",
               "Groups First Boarding",
               1.28, 1.28 )
make_histogram(modifiedsteffen$modsteffen, "#ffdfba",
               "Steffen Modified Boarding",
               1.3, 1.3 )
########################Overlay  histograms###########################
library(ggplot2)
library(plyr)
#mean
mu <- ddply(dataogmethod.transp, "Type", summarise, grp.mean=mean(Time))
# Basic density
#ggplot(dataogmethod.transp, aes(x=Time, fill=Type)) +
  #geom_density(color="darkblue", fill="lightblue")
# Use semi-transparent fill
p <-  ggplot(dataogmethod.transp, aes(x = Time, fill = Type)) +
        geom_density(alpha=0.6) +
      #theme_bw() +
      #geom_vline(data=mu, aes(xintercept=grp.mean, color="black"),
                 #linetype="dashed") +
      labs(title="                             Comparison of Methods",
           x="Time (S)", y = "Relative Frequency")

# Add mean lines
p+scale_fill_manual(values=c("#ffb3ba", "#ffffba","#ffdfba","#bae1ff","#baffc9"))
p
############Provided Methods Sensitivity Analysis#############################
dev.off()
attach(bagsensitivity)
linearModel <- lm(SectionBTF ~ NBC, data=sensitivityanalysis)
summary(linearModel)
lm(formula = Seats.No.Group ~ NBC)
sensitivityanalysis$NBC2 <- sensitivityanalysis$NBC^2
quadraticModel <- lm(SectionBTF ~ NBC + NBC2, data=sensitivityanalysis)
summary(quadraticModel)
#create sequence of hour values
NBCValues <- seq(0, 1, 0.01)
#create list of predicted time values using quadratic model
timePredict <- predict(quadraticModel,list(NBC=NBCValues, NBC2=NBCValues^2))
plot(bagsensitivity$bagcoef, bagsensitivity$section, pch = 19, cex = 0.75,
     col = "#ffb3ba", xlab = "Bag Coefficient",
     ylab = "Time (s)",
     ylim=c(450,900),
```

```r
     main = "Sensitivity Analysis of 3 Given Models")
#lines(NBCValues, timePredict, col='#ffb3ba')
abline(lm(bagsensitivity$section ~ bagsensitivity$bagcoef), col = "#ffb3ba")
points(bagsensitivity$bagcoef, bagsensitivity$random, pch = 19, cex = 0.75, col =
    "#95cca1")
abline(lm(bagsensitivity$random ~ bagsensitivity$bagcoef), col = "#95cca1")
#lines(bagsensitivity$bagcoef, bagsensitivity$wma, col="#bae1ff")
points(bagsensitivity$bagcoef, bagsensitivity$wma, pch = 19, cex = 0.75, col = "#
   bae1ff")
abline(lm(bagsensitivity$wma ~ bagsensitivity$bagcoef), col = "#bae1ff")
legend(0.7, 550, legend=c("Back to Front", "Random", "Seat (No Groups)"), col=c("
   #ffb3ba", "#95cca1", "#bae1ff"), lty=1:2, cex=0.65)
####################Suggested Methods Sensitivity############################
dev.off()
attach(seat_with_groups1)
plot(NBC, SeatwithGroups, pch = 19, cex = 0.75,
     col = "#bae1ff", xlab = "Disobedience Coefficient",
     ylab = "Time (s)",
     ylim=c(550,750),
     main = "Sensitivity Analysis of Chosen Method Against Random")
abline(lm(SeatwithGroups ~ NBC), col = "#bae1ff")
lines(NBC, Random, col="#95cca1")
points(NBC, Random, pch = 19, cex = 0.75, col = "#95cca1")

legend(0.75, 625, legend=c("Random", "Seat (Groups)"), col=c("#95cca1", "#bae1ff"
   ), lty=1, cex=0.65)
########################################################
barplot((Barplot_section_order$Mean),
        main="Mean Time Taken to Board (10k Trials)",
        names.arg=c("AFM", "AMF", "MAF","MFA","FMA","FAM"),
        ylim=c(700,900),
        xpd=FALSE,
        col="#ffb3ba",
        ylab = "Time Taken to Board (s)",
        xlab = "Boarding Method")
############################################
install.packages("ggforce")
library("ggforce")
attach(Barplot_section_order)
ggplot(Barplot_section_order, aes(Order, Mean)) +             # ggplot2 facet_
   zoom plot
  geom_bar(stat = "identity", fill = "#f2aab1") +
  ggtitle("                            Mean Time Taken to Board (10K Trials)")+
  labs(y= "Time Taken (s)", x = "Boarding Order")+
  theme(panel.background = element_rect("#f9f9f9"))+
  facet_zoom(ylim = c(700, 900), show.area=TRUE)
##########################################################################
```

Appendix E

```python

from audioop import reverse
import graphlib
from lib2to3.pgen2.token import NUMBER
import random
from string import ascii_letters
import math

```

```
#Comment out if not using
import matplotlib.pyplot as plt
import numpy
import matplotlib.colors as colors
import matplotlib as mpl
from matplotlib.colors import Colormap, LinearSegmentedColormap, ListedColormap
#Constants for refrences
PRIORITY = 0
INTERNAL_COCK = 1
HAS_LUGGAGE = 2

# all measured in standard units (m,s,m/s etc)
AVERAGE_WALKING_SPEED = 0.8
AVERAGE_SEAT_PITCH = 0.78
TIME_TO_MOVE = AVERAGE_SEAT_PITCH / AVERAGE_WALKING_SPEED
TIME_TO_MOVE_PAST_SEAT = 2

#Priority system
priority_weightings = []
highest_priority_assigned = 0
#Things to change
BAG_COEFFICIENT = (20,80,10)
NAUGHTY_BOY_COEFFICIENT = 0.3
THANOS_SNAP_COEFFICENT = 0.5
# proportions of group sizes
SINGLE_PRINGLE_COEFFICIENT = 70
COUPLES_COEFFIENCT = 20
THREESOME_COEFFICIENT = 10



# General setup shotput all seats in
NUM_ROWS = 33
NUM_SEATS = 6
AISLE_INDEX = 3
#Wide body shot
WIDE_WING_SEATS = 28
WIDE_WING_ROWS = 15
TWO_SEATS = 9
TWO_ROWS = 42
TWO_A_ROWS= 18
TWO_B_ROWS = 21
GAP_SIZE = 3




#Normal render
def intalize_render():

    global highest_priority_assigned
    #Absolute mess of code
    image = []
    for i in range(NUM_SEATS+1):
        subimage = []
        for k in range(NUM_ROWS):
            if k % 2 == 0:
                subimage.append(0.5)
            else:
                subimage.append(1.5)
```

```
        image.append(subimage)


    fig,ax = plt.subplots(1,1)

    cmap = mpl.cm.OrRd
    norm = mpl.colors.Normalize(vmin=-1, vmax=highest_priority_assigned)



    image = numpy.array(image)
    im = ax.imshow(image, cmap=cmap, norm = norm)

    ax.set_yticks(numpy.arange(0.5, NUM_SEATS+1.5, 1).tolist(), minor=False)
    ax.yaxis.grid(True, which='major')
    ax.set_yticklabels(['Row A','Row B','Row C','Aisle','Row D','Row E','Row F'])
    ax.set_ylim(top=-0.5)

    ax.set_xticks(numpy.arange(0.5, NUM_ROWS+.5, 1).tolist(), minor=False)
    ax.xaxis.grid(True, which='major')
    xticklist = []
    #Create list of numbers between
    for i in range(NUM_ROWS):
        if ((i+1) % 5 == 0) and (i != 0):
            xticklist.append(str(i+1))
        else:
            xticklist.append('')

    ax.set_xticklabels(xticklist)
    ax.set_xlim(left=-0.5)

    return im,fig
def update_render(seat_plan):

    visualizer = []

    for i,column in enumerate(seat_plan):
        visualizer.append([])
        for seats in column:

            visualizer[i].append(seats[PRIORITY])





    im.set_data(visualizer)
    fig.canvas.draw_idle()
    plt.pause(1)
def intalize_render_two_thing():

    global highest_priority_assigned

    #Absolute mess of code
    image = []
    for i in range(TWO_SEATS):
        subimage = []
        for k in range(TWO_ROWS):
            if k % 2 == 0:
```

```
                subimage.append(0.5)
            else:
                subimage.append(1.5)

        image.append(subimage)


    fig,ax = plt.subplots(1,1)

    cmap = mpl.cm.OrRd
    norm = mpl.colors.Normalize(vmin=-1, vmax=highest_priority_assigned)



    image = numpy.array(image)
    im = ax.imshow(image, cmap=cmap, norm = norm)

    ax.set_yticks(numpy.arange(0.5, TWO_SEATS+0.5, 1).tolist(), minor=False)
    ax.yaxis.grid(True, which='major')
    ax.set_yticklabels(['Row A','Row B','Aisle','Row C','Row D','Row E','Aisle','
    Row F','Row G'])
    ax.set_ylim(top=-0.5)
    ax.set_title('Two Doors Two Aisles Disembarking Model')
    ax.set_xticks(numpy.arange(0.5, TWO_ROWS+.5, 1).tolist(), minor=False)
    ax.xaxis.grid(True, which='major')
    xticklist = []
    #Create list of numbers between
    for i in range(TWO_ROWS):
        if ((i+1) % 5 == 0) and (i != 0):
            xticklist.append(str(i+1))
        else:
            xticklist.append('')

    ax.set_xticklabels(xticklist)
    ax.set_xlim(left=-0.5)

    return im,fig
def  intalize_render_widebody():

    global highest_priority_assigned

    #Absolute mess of code
    image = []
    for i in range(WIDE_WING_SEATS):
        subimage = []
        for k in range(WIDE_WING_ROWS):
            if k % 2 == 0:
                subimage.append(0.5)
            else:
                subimage.append(1.5)

        image.append(subimage)


    fig,ax = plt.subplots(1,1)

    cmap = mpl.cm.OrRd
    norm = mpl.colors.Normalize(vmin=-1, vmax=highest_priority_assigned)


```

```python

    image = numpy.array(image)
    im = ax.imshow(image, cmap=cmap, norm = norm)

    ax.set_yticks(numpy.arange(0.5, WIDE_WING_SEATS+0.5, 1).tolist(), minor=False
    )
    ax.yaxis.grid(True, which='major')
    ax.set_yticklabels(['Row A','Row B','Row C','Aisle','Row D','Row E','Row F','
    Row G','Row H','Row I','Aisle','Row J','Row K','Row L','Row M','Row N','Row O'
    ,'Aisle','Row P','Row Q','Row R','Row S','Row T','Row U','Aisle','Row V','Row
    W','Row X'])
    ax.set_ylim(top=-0.5)
    ax.set_title('Widebody Disembarking Model')
    ax.set_xticks(numpy.arange(0.5, WIDE_WING_ROWS+.5, 1).tolist(), minor=False)
    ax.xaxis.grid(True, which='major')
    xticklist = []
    #Create list of numbers between
    for i in range(WIDE_WING_ROWS):
        if ((i+1) % 5 == 0) and (i != 0):
            xticklist.append(str(i+1))
        else:
            xticklist.append('')

    ax.set_xticklabels(xticklist)
    ax.set_xlim(left=-0.5)

    return im,fig

#General shot to setup
def generate_priorties(highest_priority_assigned):


    weights = list(range(1, highest_priority_assigned+1))

    return(weights)
def group_size():
    return random.choices([1,2,3], weights=(SINGLE_PRINGLE_COEFFICIENT,
    COUPLES_COEFFIENCT,THREESOME_COEFFICIENT), k=1)[0]
def assign_luggage():
    return random.choices([0,1,2], weights=BAG_COEFFICIENT, k=1)[0]
def bag_shit():
    global seating_plan
    #Intalize bag amounts
    lockers = [[0,0] for i in range(NUM_ROWS)]
    for row in range(NUM_ROWS):
        for seat in range(NUM_SEATS+1):

            if seat < 3:
                lockers[row][0] += seating_plan[seat][row][HAS_LUGGAGE]

            elif seat > 3:
                lockers[row][1] += seating_plan[seat][row][HAS_LUGGAGE]
    return lockers
#Widebody
def bag_shit_wide():
    global seating_plan
    #Intalize bag amounts

    lockers = [[ [0,0] for _ in range(WIDE_WING_ROWS-1)] for _ in range(4)]

```

```
    for row in range(1,WIDE_WING_ROWS):

        for seat in range(1,WIDE_WING_SEATS):
            sublocker = math.floor((seat)/7)
            if seating_plan[seat][row][PRIORITY] != -1:
                if seat % 7 < 3:
                    lockers[sublocker][row-1][0] += seating_plan[seat][row][
    HAS_LUGGAGE]

                elif seat % 7 > 3:
                    lockers[sublocker][row-1][1] += seating_plan[seat][row][
    HAS_LUGGAGE]






    return lockers
#Modifitying shot
def group_shit():
        for row in range(NUM_ROWS):
            #Resets var
            current_group_size = 0
            current_group_priorty = []
            current_group_people_added = 0
            for seat in range(NUM_SEATS+1):
                #make sure we not in aisles
                if seat != AISLE_INDEX:
                    #If not currently generating create a new group
                    if current_group_people_added == 0:
                        current_group_size = current_group_people_added =
    group_size()
                        if current_group_people_added == 1:
                            current_group_people_added = 0
                        else:
                            current_group_priorty.append(seating_plan[seat][row][
    PRIORITY])
                            current_group_people_added -=1


                    else: # Currently generating a group
                        current_group_priorty.append(seating_plan[seat][row][
    PRIORITY])
                        current_group_people_added -=1

                        #If all people added to group
                        if current_group_people_added == 0:
                            #Loop back through people and send priority to
    average


                            gone_through_aisles = 0
                            for i in range(current_group_size):
                                #Go back through and adjust priority

                                #If gone through aisles add another
                                if (seat - i) == AISLE_INDEX:
                                    gone_through_aisles = 1
```

```
                                    seating_plan[seat-(i+gone_through_aisles)][row][
    PRIORITY] = round(sum(current_group_priorty) / len(current_group_priorty))
def naughty_people():

        #Generate total amount of naughty boys
        naughty_bois = math.ceil(NUM_ROWS*NUM_SEATS*NAUGHTY_BOY_COEFFICIENT)

        for i in range(naughty_bois):
            numbers = list(range(0, NUM_SEATS+1))

            numbers.remove(3)

            seat = random.choice(numbers)
            row = random.randrange(NUM_ROWS)

            seating_plan[seat][row][PRIORITY] = random.randrange(1,
    highest_priority_assigned+1)
def thanos_snap():
    for seat in range(NUM_SEATS+1):
        for row in range(NUM_ROWS):
            if THANOS_SNAP_COEFFICENT > random.random():
                seating_plan[seat][row] = [-1,0]
#Narrow body boarding
def reverse_wilma():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = 3

    seating_plan = [[ [3,0,assign_luggage()] for _ in range(NUM_ROWS)] for _ in
    range(NUM_SEATS + 1)]
    seating_plan[0]= [[1,0,assign_luggage()]  for _ in range(NUM_ROWS)]
    seating_plan[6]= [[1,0,assign_luggage()]  for _ in range(NUM_ROWS)]
    seating_plan[1]= [[2,0,assign_luggage()]  for _ in range(NUM_ROWS)]
    seating_plan[5]= [[2,0,assign_luggage()]  for _ in range(NUM_ROWS)]
    seating_plan[AISLE_INDEX]= [[-1,0]  for _ in range(NUM_ROWS)]

    naughty_people()
    group_shit()
def random_deboard():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = 10

    seating_plan = [[ [random.randrange(1,10),0,assign_luggage()] for _ in range(
    NUM_ROWS)] for _ in range(NUM_SEATS + 1)]
    seating_plan[AISLE_INDEX]= [[-1,0]  for _ in range(NUM_ROWS)]

    group_shit()
def sections():

    global seating_plan

    global highest_priority_assigned

    highest_priority_assigned = 3
    fjuk = []
    for i in range(NUM_SEATS+1):
```

```
        aisles = []
        for k in range(0,11):
            aisles.append([3,0,assign_luggage()])
        for k in range(11,22):
            aisles.append([2,0,assign_luggage()])
        for k in range(22,NUM_ROWS):
            aisles.append([1,0,assign_luggage()])
        fjuk.append(aisles)
    seating_plan = fjuk
    seating_plan[AISLE_INDEX]= [[-1,0]  for _ in range(NUM_ROWS)]
    naughty_people()
    group_shit()
def back_to_front():
    global seating_plan
    global highest_priority_assigned


    #Create empty seating plan
    seating_plan = [[ [-1,0,assign_luggage()] for _ in range(NUM_ROWS)] for _ in range(NUM_SEATS + 1)]

    highest_priority_assigned = 0

    for row in range(NUM_ROWS):
        for seat in range(NUM_SEATS+1):
            #Increment priority


            if seat == 0 or seat == 3:
                highest_priority_assigned += 1
            seating_plan[seat][row] = ([highest_priority_assigned,0, assign_luggage()])

    seating_plan[AISLE_INDEX]= [[-1,0]  for _ in range(NUM_ROWS)]
    #naughty_people()
    #group_shit()
def generate_front_to_back():
    global seating_plan
    global highest_priority_assigned


    #Create empty seating plan
    seating_plan = [[ [-1,0,assign_luggage()] for _ in range(NUM_ROWS)] for _ in range(NUM_SEATS + 1)]

    highest_priority_assigned = 0

    for row in reversed(range(NUM_ROWS)):
        for seat in range(NUM_SEATS+1):
            #Increment priority


            if seat == 0 or seat == 3:
                highest_priority_assigned += 1
            seating_plan[seat][row] = ([highest_priority_assigned,0, assign_luggage()])

    seating_plan[AISLE_INDEX]= [[-1,0]  for _ in range(NUM_ROWS)]


```

```
    naughty_people()
    group_shit()

def group_shit_two():
    for row in range(4,TWO_ROWS):
        #Resets var
        current_group_size = 0
        current_group_priorty = []
        current_group_people_added = 0



        for seat in range(0,9):

            #make sure we not in aisles
            if (seat != 2 or seat !=4) and seating_plan[seat][row][PRIORITY] != -1:
                #If not currently generating create a new group
                if current_group_people_added == 0:
                    current_group_size = current_group_people_added = group_size()
                    if current_group_people_added == 1:
                        current_group_people_added = 0
                    else:
                        current_group_priorty.append(seating_plan[seat][row][PRIORITY])
                        current_group_people_added -=1


                else: # Currently generating a group
                    current_group_priorty.append(seating_plan[seat][row][PRIORITY])
                    current_group_people_added -=1

                    #If all people added to group
                    if current_group_people_added == 0:
                        #Loop back through people and send priority to average


                        gone_through_aisles = 0
                        for i in range(current_group_size):
                            #Go back through and adjust priority

                            #If gone through aisles add another
                            if (seat - i) == 2 or (seat - i) == 6:
                                gone_through_aisles = 1

                            seating_plan[(seat-(i+gone_through_aisles))][row][PRIORITY] = round(sum(current_group_priorty) / len(current_group_priorty))
#Wide body boarding
def clear_aisles_widebody():
    global seating_plan


    #Clear aisles
    for i in range(WIDE_WING_SEATS):
        if i % 7 == 3 and i != WIDE_WING_SEATS:
            seating_plan[i]= [[-1,0]  for _ in range(WIDE_WING_ROWS)]
    #Clear front aisles
    for k in range(WIDE_WING_SEATS):
```

```
        seating_plan[k][0] = [-1,0]

    #Clear extra 9 on both sides
    for k in range(1, 4):
        for j in range(0,3):
           seating_plan[j][k] = [-1,0]

    #Clear extra 9 on both sides
    for k in range(1, 4):
        for j in range(WIDE_WING_SEATS-3,WIDE_WING_SEATS):
           seating_plan[j][k] = [-1,0]
def naughty_people_wide():

        #Generate total amount of naughty boys
        naughty_bois = math.ceil(((WIDE_WING_ROWS-1)*(WIDE_WING_SEATS-4)-18)*NAUGHTY_BOY_COEFFICIENT)

        for i in range(naughty_bois):

            while True:
                seat = random.randrange(WIDE_WING_SEATS)
                row = random.randrange(WIDE_WING_ROWS)

                if seating_plan[seat][row][PRIORITY] != -1:
                    seating_plan[seat][row][PRIORITY] = random.randrange(1, highest_priority_assigned+1)
                    break
def group_shit_wide():
        for row in range(WIDE_WING_ROWS):
            #Resets var
            current_group_size = 0
            current_group_priorty = []
            current_group_people_added = 0


            for current_aisle in range(0, WIDE_WING_SEATS,7):
                for seat in range(0,7):

                    #make sure we not in aisles
                    if seat != 3 and seating_plan[current_aisle+seat][row][PRIORITY] != -1:
                        #If not currently generating create a new group
                        if current_group_people_added == 0:
                            current_group_size = current_group_people_added = group_size()
                            if current_group_people_added == 1:
                                current_group_people_added = 0
                            else:
                                current_group_priorty.append(seating_plan[current_aisle+seat][row][PRIORITY])
                                current_group_people_added -=1


                        else: # Currently generating a group
                            current_group_priorty.append(seating_plan[current_aisle+seat][row][PRIORITY])
                            current_group_people_added -=1

                            #If all people added to group
                            if current_group_people_added == 0:
```

```
                                #Loop back through people and send priority to average


                                gone_through_aisles = 0
                                for i in range(current_group_size):
                                    #Go back through and adjust priority

                                    #If gone through aisles add another
                                    if (seat - i) == 3:
                                        gone_through_aisles = 1

                                    seating_plan[(seat-(i+gone_through_aisles))+current_aisle][row][PRIORITY] = round(sum(current_group_priorty) / len(current_group_priorty))
def reverse_wilma_widebody():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = 3

    #Make an empty
    seating_plan = [[ [-1,0,0] for _ in range(WIDE_WING_ROWS)] for _ in range(WIDE_WING_SEATS)]

    for i in range(WIDE_WING_SEATS):
        if i in [2,4,9,11,16,18,23,25]:
            seating_plan[i]= [[3,0,assign_luggage()]  for _ in range(WIDE_WING_ROWS)]
        elif i in [1,5,8,12,15,19,22,26]:
            seating_plan[i]= [[2,0,assign_luggage()]  for _ in range(WIDE_WING_ROWS)]
        elif i in [0,6,7,13,14,20,21,27]:
            seating_plan[i]= [[1,0,assign_luggage()]  for _ in range(WIDE_WING_ROWS)]


    clear_aisles_widebody()
    naughty_people_wide()
    group_shit_wide()
def random_deboard_widebody():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = 10

    seating_plan = [[ [ random.randrange(1,10),0,assign_luggage()] for _ in range(WIDE_WING_ROWS)] for _ in range(WIDE_WING_SEATS)]

    clear_aisles_widebody()
    group_shit_wide()
def sections_widebody():

    global seating_plan

    global highest_priority_assigned

    highest_priority_assigned = 3
    fjuk = []
    for i in range(WIDE_WING_SEATS):
```

```
        aisles = []
        for k in range(1,8):
            aisles.append([3,0,assign_luggage()])
        for k in range(8,12):
            aisles.append([2,0,assign_luggage()])
        for k in range(12,WIDE_WING_ROWS+1):
            aisles.append([1,0,assign_luggage()])
        fjuk.append(aisles)
    seating_plan = fjuk



    clear_aisles_widebody()

    naughty_people_wide()
    group_shit_wide()
def back_to_front_widebody():
    global seating_plan
    global highest_priority_assigned


    #Create empty seating plan
    seating_plan = [[ [k+j,0,assign_luggage()] for k in range(WIDE_WING_ROWS)]
    for j in range(WIDE_WING_SEATS)]

    highest_priority_assigned = WIDE_WING_SEATS+WIDE_WING_ROWS

    clear_aisles_widebody()

    naughty_people_wide()
    group_shit_wide()
def across_widebody():
    global seating_plan
    global highest_priority_assigned


    #Create empty seating plan
    seating_plan = [[ [k,0,assign_luggage()] for k in range(WIDE_WING_ROWS)] for
    j in range(WIDE_WING_SEATS)]

    highest_priority_assigned = WIDE_WING_ROWS

    clear_aisles_widebody()

    naughty_people_wide()
    group_shit_wide()
def naughty_people_two ():

        global seating_plan
        global highest_priority_assigned

        #Generate total amount of naughty boys
        naughty_bois = math.ceil(((TWO_SEATS-2)*(TWO_A_ROWS+TWO_B_ROWS)+18)*
    NAUGHTY_BOY_COEFFICIENT)

        for i in range(naughty_bois):

            while True:
                seat = random.randrange(TWO_SEATS)
                row = random.randrange(TWO_ROWS)
```

```
                if seating_plan[seat][row][PRIORITY] != -1:
                    seating_plan[seat][row][PRIORITY] = random.randrange(1, highest_priority_assigned)
                    break
def generate_front_to_back_widebody ():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = WIDE_WING_SEATS+WIDE_WING_ROWS
    #Create empty seating plan
    seating_plan = [[ [highest_priority_assigned-(k+j),0,assign_luggage()] for k in range(WIDE_WING_ROWS)] for j in range(WIDE_WING_SEATS)]



    clear_aisles_widebody()

    naughty_people_wide()
    group_shit_wide()


def bag_shit_Two():
    global seating_plan
    #Intalize bag amounts
    lockers = [[[0,0] for _ in range(TWO_A_ROWS)],[[0,0] for _ in range(TWO_B_ROWS)]]

    #A first
    for row in range(TWO_A_ROWS):

        for seat in range(TWO_SEATS):

            if seating_plan[seat][row][PRIORITY] != -1:
                if seat <=3:
                    lockers[0][row][0] += seating_plan[seat][row][HAS_LUGGAGE]

                elif seat > 3:
                    lockers[0][row][1] += seating_plan[seat][row][HAS_LUGGAGE]
    #B second
    for row in range(TWO_B_ROWS):

        for seat in range(TWO_SEATS):

            if seating_plan[seat][row+GAP_SIZE+TWO_A_ROWS][PRIORITY] != -1:
                if seat <=3:
                    lockers[1][row][0] += seating_plan[seat][row+GAP_SIZE+TWO_A_ROWS][HAS_LUGGAGE]

                elif seat > 3:
                    lockers[1][row][1] += seating_plan[seat][row+GAP_SIZE+TWO_A_ROWS][HAS_LUGGAGE]






    return lockers
def two_first_class():
```

```
    global seating_plan
    global highest_priority_assigned

    #Generate first class
    seating_plan[0][0][0] = highest_priority_assigned
    seating_plan[0][1][0] = highest_priority_assigned
    seating_plan[0][2][0] = highest_priority_assigned
    seating_plan[1][0][0] = highest_priority_assigned
    seating_plan[1][1][0] = highest_priority_assigned
    seating_plan[1][2][0] = highest_priority_assigned
    seating_plan[3][0][0] = highest_priority_assigned
    seating_plan[3][1][0] = highest_priority_assigned
    seating_plan[3][2][0] = highest_priority_assigned
    seating_plan[5][0][0] = highest_priority_assigned
    seating_plan[5][1][0] = highest_priority_assigned
    seating_plan[5][2][0] = highest_priority_assigned
    seating_plan[7][0][0] = highest_priority_assigned
    seating_plan[7][1][0] = highest_priority_assigned
    seating_plan[7][2][0] = highest_priority_assigned
    seating_plan[8][0][0] = highest_priority_assigned
    seating_plan[8][1][0] = highest_priority_assigned
    seating_plan[8][2][0] = highest_priority_assigned
 #Two
def two_cleanup():
    global seating_plan
    global highest_priority_assigned



    #Clear aisles
    seating_plan[2]= [[-1,0]  for _ in range(TWO_ROWS)]
    seating_plan[6]= [[-1,0]  for _ in range(TWO_ROWS)]
    #Tidy up first class
    seating_plan[4][0] = [-1,0]
    seating_plan[4][1] = [-1,0]
    seating_plan[4][2] = [-1,0]


    #Clear queues out
    for k in range(TWO_SEATS):
        for j in range(TWO_ROWS):
            if j in [3, 18,19,20,42]:
                seating_plan[k][j] = [-1,0]

def two_random():
    global seating_plan
    global highest_priority_assigned

    seating_plan = [[ [random.randrange(1,10),0,assign_luggage()] for _ in range(
   TWO_ROWS)] for _ in range(TWO_SEATS)]

    highest_priority_assigned = 10
    two_first_class()
    two_cleanup()
    naughty_people_two()
    group_shit_two()

def two_back_to_front():
    global seating_plan
    global highest_priority_assigned
```

```
    highest_priority_assigned = 0
    #Generate an empty plane
    seating_plan = [[ [0,0,0] for _ in range(TWO_ROWS)] for _ in range(TWO_SEATS)
   ]
    for _ in range(TWO_SEATS):
        for k in range(4,TWO_A_ROWS):

            seating_plan[_][k] = [k,0,assign_luggage()]

    for _ in range(TWO_SEATS):
        for k in (range(TWO_B_ROWS)):

            seating_plan[_][(TWO_B_ROWS-k)+(TWO_A_ROWS+GAP_SIZE-1)] = [k,0,
   assign_luggage()]

    highest_priority_assigned = TWO_B_ROWS

    two_first_class()
    two_cleanup()
    naughty_people_two()
    group_shit_two()

def two_front_to_back():
    global seating_plan
    global highest_priority_assigned
    highest_priority_assigned = 0
    #Generate an empty plane
    seating_plan = [[ [0,0,0] for _ in range(TWO_ROWS)] for _ in range(TWO_SEATS)
   ]
    for _ in range(TWO_SEATS):
        for k in range(4,TWO_A_ROWS):

            seating_plan[_][TWO_A_ROWS-k] = [k,0,assign_luggage()]

    for _ in range(TWO_SEATS):
        for k in (range(TWO_B_ROWS)):

            seating_plan[_][(k)+(TWO_A_ROWS+GAP_SIZE)] = [k,0,assign_luggage()]

    highest_priority_assigned = TWO_B_ROWS

    two_first_class()
    two_cleanup()
    naughty_people_two()
    group_shit_two()

def two_reverse_wilma_widebody():
    global seating_plan
    global highest_priority_assigned

    highest_priority_assigned = 2

    #Make an empty
    seating_plan = [[ [-1,0,0] for _ in range(TWO_ROWS)] for _ in range(TWO_SEATS
   )]

    for i in range(TWO_SEATS):
        if i in [1,3,5,7]:
            seating_plan[i]= [[2,0,assign_luggage()]  for _ in range(TWO_ROWS)]
        elif i in [0,4,8]:
```

```
            seating_plan[i]= [[1,0,assign_luggage()]  for _ in range(TWO_ROWS)]

    two_first_class()
    two_cleanup()
    naughty_people_two()
    group_shit_two()


def two_reverse_sections_360():
    global seating_plan

    global highest_priority_assigned

    highest_priority_assigned = 3
    fjuk = []
    for i in range(TWO_SEATS):
        aisles = []
        for k in range(0,8):
            aisles.append([3,0,assign_luggage()])
        for k in range(8,13):
            aisles.append([2,0,assign_luggage()])
        for k in range(13,21):
            aisles.append([1,0,assign_luggage()])
        for k in range(21,28):
            aisles.append([1,0,assign_luggage()])
        for k in range(28,36):
            aisles.append([2,0,assign_luggage()])
        for k in range(36,TWO_ROWS):
            aisles.append([3,0,assign_luggage()])
        fjuk.append(aisles)
    seating_plan = fjuk

    two_first_class()
    two_cleanup()
    naughty_people_two()
    group_shit_two()







#Logic
def check_locker_space_wide(luggage_number, current_row, seat, lockers):
    # if passenger has no baggage
    if luggage_number == 0:
        return 0

    sublocker = math.floor((seat)/7)

    if seat % 7 < 3:
        nbins = lockers[sublocker][current_row-1][0]
        lockers[sublocker][current_row-1][0]  -= luggage_number

    elif seat % 7 > 3:
        nbins = lockers[sublocker][current_row-1][1]
        lockers[sublocker][current_row-1][1]  -= luggage_number

    # derivations in writeup
```

```
    if luggage_number == 1:
        t = (4)/(1-(0.8*((nbins-2)))/6)
    if luggage_number == 2:
        t = (4)/(1-(0.8*((nbins-2)))/6) + (2.25)/(1-((nbins-2))/6)

    return t
def check_locker_space(luggage_number, current_row, down, lockers):
    # if passenger has no baggage
    if luggage_number == 0:
        return 0

    # if on right side of aisle
    if down==True:

        nbins = lockers[NUM_ROWS-current_row-1][1]
        lockers[NUM_ROWS-current_row-1][1] -= luggage_number
    else:

        nbins = lockers[NUM_ROWS-current_row-1][0]
        lockers[NUM_ROWS-current_row-1][0] -= luggage_number
    if luggage_number == 1:
        t = (4)/(1-(0.8*((nbins-2)))/6)
    if luggage_number == 2:
        t = (4)/(1-(0.8*((nbins-2)))/6) + (2.25)/(1-((nbins-2))/6)

    return t
def check_locker_space_Two(luggage_number, current_row, seat, lockers,sectionA):

    if sectionA:
        thingy = 0
    else:
        thingy = 1

    # if passenger has no baggage
    if luggage_number == 0:
        return 0

    # if on right side of aisle
    if seat > 3:

        nbins = lockers[thingy][current_row][1]
        lockers[thingy][current_row][1] -= luggage_number
    else:

        nbins = lockers[thingy][current_row][0]
        lockers[thingy][current_row][0] -= luggage_number
    if luggage_number == 1:
        t = (4)/(1-(0.8*((nbins-2)))/6)
    if luggage_number == 2:
        t = (4)/(1-(0.8*((nbins-2)))/6) + (2.25)/(1-((nbins-2))/6)

    return t
def move_up(seat,row):

    if seating_plan[seat-1][row][0] == -1:
        seating_plan[seat-1][row] = seating_plan[seat][row]
        seating_plan[seat-1][row][1] = 0
        seating_plan[seat][row] = [-1,0]
def move_down(seat,row):

```

```
    if seating_plan[seat+1][row][0] == -1:
        seating_plan[seat+1][row] = seating_plan[seat][row]
        seating_plan[seat+1][row][1] = 0
        seating_plan[seat][row] = [-1,0]
def aisle_take_above(row,hello,world):
    #Move person from above into aisle
    seating_plan[AISLE_INDEX][row] = seating_plan[AISLE_INDEX-1][row]
    seating_plan[AISLE_INDEX-1][row] = [-1,0]

    #Luggage
    seating_plan[AISLE_INDEX][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[AISLE_INDEX][row][HAS_LUGGAGE], row, True, lockers
   )
def aisle_take_below(row,frick,me):
    #Move person from below into aisle
    seating_plan[AISLE_INDEX][row] = seating_plan[AISLE_INDEX+1][row]
    seating_plan[AISLE_INDEX+1][row] = [-1,0]


    seating_plan[AISLE_INDEX][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[AISLE_INDEX][row][HAS_LUGGAGE], row, False,
   lockers)
def aisle_take_above_wide(row,current_aisles,uguil):
    #Move person from above into aisle
    #Luggage
    seating_plan[current_aisles-1][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[current_aisles-1][row][HAS_LUGGAGE], row,
   current_aisles-1, lockers)

    seating_plan[current_aisles][row] = seating_plan[current_aisles-1][row]
    seating_plan[current_aisles-1][row] = [-1,0]
def aisle_take_above_Two(row,current_aisles,SectionA):
    #Move person from above into aisle
    #Luggage
    if row != 3:
        seating_plan[current_aisles-1][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[current_aisles-1][row][HAS_LUGGAGE], row-(GAP_SIZE
   +TWO_A_ROWS), current_aisles-1, lockers,SectionA)

    seating_plan[current_aisles][row] = seating_plan[current_aisles-1][row]
    seating_plan[current_aisles-1][row] = [-1,0]

def aisle_take_below_Two(row,current_aisles,sectionA):
    #Move person from below into aisle
    if row != 3:
        seating_plan[current_aisles+1][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[current_aisles+1][row][HAS_LUGGAGE], row-(GAP_SIZE
   +TWO_A_ROWS), current_aisles+1, lockers,sectionA)

    seating_plan[current_aisles][row] = seating_plan[current_aisles+1][row]
    seating_plan[current_aisles+1][row] = [-1,0]

def aisle_take_below_wide(row,current_aisles,aszgasdhasdh):
    #Move person from below into aisle
    if row != 0:
        seating_plan[current_aisles+1][row][INTERNAL_COCK] = -locker_shit_type[
   boarding_type](seating_plan[current_aisles+1][row][HAS_LUGGAGE], row,
   current_aisles+1, lockers)

```

```
    seating_plan[current_aisles][row] = seating_plan[current_aisles+1][row]
    seating_plan[current_aisles+1][row] = [-1,0]





def aisle_take_left(row,current_aisles,whythefucknot):

    #Move person from right into aisle
    seating_plan[current_aisles][row] = seating_plan[current_aisles][row-1]
    seating_plan[current_aisles][row-1] = [-1,0]
    seating_plan[current_aisles][row][INTERNAL_COCK] = 0
def aisle_take_right(row, idonot, careanymore):

    #Move person from right into aisle
    seating_plan[AISLE_INDEX][row] = seating_plan[AISLE_INDEX][row+1]
    seating_plan[AISLE_INDEX][row+1] = [-1,0]
    seating_plan[AISLE_INDEX][row][INTERNAL_COCK] = 0

def aisle_take_right_wide(row,current_aisles,whythefucknot):

    #Move person from right into aisle
    seating_plan[current_aisles][row] = seating_plan[current_aisles][row+1]
    seating_plan[current_aisles][row+1] = [-1,0]
    seating_plan[current_aisles][row][INTERNAL_COCK] = 0

#While loop
def off_the_plane(generation_method,text):
    global im,fig
    #isual shot

    global seating_plan
    global priority_weightings
    global lockers
    test_cases = []




    for i in range(N_TEST_CASES):
        generation_method()
        total_time=0
        left_plane = 0
        priority_weightings = generate_priorties(highest_priority_assigned)
        lockers = bag_shit_type[boarding_type]()
        if VISUALIZER:
            im,fig = render_type[boarding_type]()

        while True:
            if boarding_type == TWO:
                #Exit square
                if seating_plan[2][41][PRIORITY] != -1 and seating_plan[2][41][
    INTERNAL_COCK] >= TIME_TO_MOVE:
                    #Empty square
                    seating_plan[2][41] = [-1,0]
                    left_plane += 1

                if seating_plan[6][41][PRIORITY] != -1 and seating_plan[6][41][
    INTERNAL_COCK] >= TIME_TO_MOVE:
```

```
                #Empty square
                seating_plan[6][41] = [-1,0]
                left_plane += 1
            if seating_plan[TWO_SEATS-1][3][PRIORITY] != -1 and seating_plan[TWO_SEATS-1][3][INTERNAL_COCK] >= TIME_TO_MOVE:
                #Empty square
                seating_plan[TWO_SEATS-1][3] = [-1,0]
                left_plane += 1


            #End code shot
            if left_plane == 249:
                test_cases.append(total_time)
                break

            for current_aisle in (range(TWO_SEATS)):
                if seating_plan[current_aisle][3][PRIORITY] == -1:

                        priorities = [0,0,0,0]
                        possible_moves = [aisle_take_above_Two, aisle_take_below_Two, aisle_take_right_wide,aisle_take_left ]
                        total_move_possibilites = 0

                        #Get things to check
                        is_person_above_moving = seating_plan[current_aisle-1][3][PRIORITY] != -1 and seating_plan[current_aisle-1][3][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT

                        if is_person_above_moving:
                            priorities[0] = priority_weightings[seating_plan[current_aisle-1][3][PRIORITY]-1]
                            total_move_possibilites +=1
                        if current_aisle != 8:
                            is_person_below_moving = False #seating_plan[current_aisle+1][3][PRIORITY] != -1 and seating_plan[current_aisle+1][3][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT
                        else: is_person_below_moving = 0
                        if is_person_below_moving:

                            priorities[1] = 0 #priority_weightings[seating_plan[current_aisle+1][3][PRIORITY]-1]
                            total_move_possibilites +=1
                        #Prevent indexing error
                        if True: #row+1+ec != TWO_ROWS:
                            is_person_right_moving = seating_plan[current_aisle][3+1][PRIORITY] != -1 and seating_plan[current_aisle][3+1][INTERNAL_COCK] >= TIME_TO_MOVE
                        else:
                            is_person_right_moving = 0

                        if is_person_right_moving and (current_aisle == 2 or current_aisle == 6):

                            priorities[2] = priority_weightings[seating_plan[current_aisle][3+1][PRIORITY]-1]
                            total_move_possibilites +=1
                    #ewginsdaogvnadsklbvasj nwklsfnwdsf
                        if True: #row+1+ec != TWO_ROWS:
                            is_person_left_moving = seating_plan[current_aisle][3-1][PRIORITY] != -1 and seating_plan[current_aisle][3-1][
```

```
INTERNAL_COCK] >= TIME_TO_MOVE
                            else:
                                is_person_right_moving = 0

                            if is_person_left_moving and (current_aisle == 2 or
current_aisle == 6) :
                                priorities[3] = priority_weightings[seating_plan[
current_aisle][3-1][PRIORITY]-1]
                                total_move_possibilites +=1

                            #Decide who moves above and below
                            if total_move_possibilites > 0:
                                #Reset time
                                seating_plan[current_aisle][3][INTERNAL_COCK] = 0

                                #frick knows what is happening here but it works
so it stays
                                move = numpy.argwhere(priorities == numpy.amax(
priorities))
                                possible_moves[(random.choice(move))[0]](3,
current_aisle,False)

                for current_aisle in range(2,TWO_SEATS,4):

                    for row in reversed(range(0,TWO_B_ROWS)):

                        #extra constant
                        ec = GAP_SIZE+TWO_A_ROWS
                        #Check if aisles place is empty
                        if seating_plan[current_aisle][row+ec][PRIORITY] == -1:

                            priorities = [0,0,0]
                            possible_moves = [aisle_take_above_Two,
aisle_take_below_Two, aisle_take_left ]
                            total_move_possibilites = 0

                            #Get things to check
                            is_person_above_moving = seating_plan[current_aisle
-1][row+ec][PRIORITY] != -1 and seating_plan[current_aisle-1][row+ec][
INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT

                            if is_person_above_moving:
                                priorities[0] = priority_weightings[seating_plan[
current_aisle-1][row+ec][PRIORITY]-1]
                                total_move_possibilites +=1

                            is_person_below_moving = seating_plan[current_aisle
+1][row+ec][PRIORITY] != -1 and seating_plan[current_aisle+1][row+ec][
INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT
                            if is_person_below_moving:

                                priorities[1] = priority_weightings[seating_plan[
current_aisle+1][row+ec][PRIORITY]-1]
                                total_move_possibilites +=1
                            #Prevent indexing error
                            if True: #row+1+ec != TWO_ROWS:
                                is_person_right_moving = seating_plan[
current_aisle][row-1+ec][PRIORITY] != -1 and seating_plan[current_aisle][row
-1+ec][INTERNAL_COCK] >= TIME_TO_MOVE
                            else:
```

```
                                is_person_right_moving = 0

                            if is_person_right_moving:

                                priorities[2] = priority_weightings[seating_plan[current_aisle][row+ec-1][PRIORITY]-1]
                                total_move_possibilites +=1

                            #Decide who moves above and below
                            if total_move_possibilites > 0:
                                #Reset time
                                seating_plan[current_aisle][row+ec][INTERNAL_COCK] = 0

                                #frick knows what is happening here but it works so it stays
                                move = numpy.argwhere(priorities == numpy.amax(priorities))
                                possible_moves[(random.choice(move))[0]](row+ec,current_aisle,False)


                for row in (range(0,TWO_A_ROWS)):
                    if row != 3:
                        #extra constant

                        #Check if aisles place is empty
                        if seating_plan[current_aisle][row][PRIORITY] == -1:

                            priorities = [0,0,0,0]
                            possible_moves = [aisle_take_above_Two, aisle_take_below_Two, aisle_take_right_wide,aisle_take_left ]
                            total_move_possibilites = 0

                            #Get things to check
                            is_person_above_moving = seating_plan[current_aisle-1][row][PRIORITY] != -1 and seating_plan[current_aisle-1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT

                            if is_person_above_moving:
                                priorities[0] = priority_weightings[seating_plan[current_aisle-1][row][PRIORITY]-1]
                                total_move_possibilites +=1

                            is_person_below_moving = seating_plan[current_aisle+1][row][PRIORITY] != -1 and seating_plan[current_aisle+1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT
                            if is_person_below_moving:

                                priorities[1] = priority_weightings[seating_plan[current_aisle+1][row][PRIORITY]-1]
                                total_move_possibilites +=1
                            #Prevent indexing error
                            if True: #row+1+ec != TWO_ROWS:
                                is_person_right_moving = seating_plan[current_aisle][row+1][PRIORITY] != -1 and seating_plan[current_aisle][row+1][INTERNAL_COCK] >= TIME_TO_MOVE
                            else:
                                is_person_right_moving = 0

```

```
                            if is_person_right_moving:

                                priorities[2] = priority_weightings[seating_plan[current_aisle][row+1][PRIORITY]-1]
                                total_move_possibilites +=1
                        #ewginsdaogvnadsklbvasj nwklsfnwdsf
                            if True: #row+1+ec != TWO_ROWS:
                                is_person_left_moving = seating_plan[current_aisle][row-1][PRIORITY] != -1 and seating_plan[current_aisle][row-1][INTERNAL_COCK] >= TIME_TO_MOVE
                            else:
                                is_person_right_moving = 0

                            if is_person_left_moving and row == 3:
                                priorities[3] = priority_weightings[seating_plan[current_aisle][row-1][PRIORITY]-1]
                                total_move_possibilites +=1

                            #Decide who moves above and below
                            if total_move_possibilites > 0:
                                #Reset time
                                seating_plan[current_aisle][row][INTERNAL_COCK] = 0

                                #frick knows what is happening here but it works so it stays
                                move = numpy.argwhere(priorities == numpy.amax(priorities))
                                possible_moves[(random.choice(move))[0]](row,current_aisle,False)

                #Move towards aisle
                for row in range(TWO_ROWS):
                    if row!= 3:
                        if seating_plan[0][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT:
                            move_down(0,row)
                        if seating_plan[TWO_SEATS-1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT and seating_plan[TWO_SEATS-1][row][PRIORITY] != -1:
                            move_up(TWO_SEATS-1,row)
                        if seating_plan[4][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT:
                            if random.randint(0,1) and seating_plan[3][row][PRIORITY] == -1:
                                move_up(4,row)
                            elif seating_plan[5][row][PRIORITY] == -1:
                                move_down(4,row)





            for i in range(TWO_SEATS):
                for k in range(TWO_ROWS):
                    seating_plan[i][k][INTERNAL_COCK] += TIME_STEP
```

```



            elif boarding_type == WIDEBODY:
                #Exit square
                if seating_plan[0][0][PRIORITY] != -1 and seating_plan[0][0][
    INTERNAL_COCK] >= TIME_TO_MOVE:
                    #Empty square
                    seating_plan[0][0] = [-1,0]

                    #End code shot
                    left_plane += 1
                    if left_plane == (WIDE_WING_ROWS-1)*(WIDE_WING_SEATS-4)-18:
                        test_cases.append(total_time)

                        break

                #Down queue do not touch 2am code
                for seat in range(WIDE_WING_SEATS):

                    #if can move something in
                    if seating_plan[seat][0][PRIORITY] == -1:

                        priorities = [0,0]
                        possible_moves = [ aisle_take_below_wide,
    aisle_take_right_wide ]
                        total_move_possibilites = 0

                        #Check shot
                        is_person_below_moving = seating_plan[seat][1][PRIORITY]
    != -1 and seating_plan[seat][1][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT and
    seat in [3,10,17,24]
                        if is_person_below_moving:

                            priorities[1] = priority_weightings[seating_plan[seat
    ][1][PRIORITY]-1]
                            total_move_possibilites +=1
                        #Prevent indexing error
                        if seat != WIDE_WING_SEATS-1:
                            is_person_right_moving = seating_plan[seat+1][0][
    PRIORITY] != -1 and seating_plan[seat+1][0][INTERNAL_COCK] >= TIME_TO_MOVE
                        else:
                            is_person_right_moving = 0

                        if is_person_right_moving:

                            priorities[0] = priority_weightings[seating_plan[seat
    +1][0][PRIORITY]-1]
                            total_move_possibilites +=1

                        #Decide who moves above and below
                        if total_move_possibilites > 0:
                            #Reset time
                            seating_plan[seat][0][INTERNAL_COCK] = 0

                            #frick knows what is happening here but it works so
    it stays
                            move = numpy.argwhere(priorities == numpy.amax(
    priorities))
                            possible_moves[(random.choice(move))[0]](0,seat,False
```

```
)



            for current_aisle in range(3, WIDE_WING_SEATS,7):



                for row in range(1,WIDE_WING_ROWS):



                    #Check if aisles place is empty
                    if seating_plan[current_aisle][row][PRIORITY] == -1:

                        priorities = [0,0,0]
                        possible_moves = [aisle_take_above_wide, aisle_take_below_wide, aisle_take_right_wide ]
                        total_move_possibilites = 0

                        #Get things to check
                        is_person_above_moving = seating_plan[current_aisle-1][row][PRIORITY] != -1 and seating_plan[current_aisle-1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT

                        if is_person_above_moving:
                            priorities[0] = priority_weightings[seating_plan[current_aisle-1][row][PRIORITY]-1]
                            total_move_possibilites +=1

                        is_person_below_moving = seating_plan[current_aisle+1][row][PRIORITY] != -1 and seating_plan[current_aisle+1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT
                        if is_person_below_moving:

                            priorities[1] = priority_weightings[seating_plan[current_aisle+1][row][PRIORITY]-1]
                            total_move_possibilites +=1
                        #Prevent indexing error
                        if row != WIDE_WING_ROWS-1:
                            is_person_right_moving = seating_plan[current_aisle][row+1][PRIORITY] != -1 and seating_plan[current_aisle][row+1][INTERNAL_COCK] >= TIME_TO_MOVE
                        else:
                            is_person_right_moving = 0

                        if is_person_right_moving:

                            priorities[2] = priority_weightings[seating_plan[current_aisle][row+1][PRIORITY]-1]
                            total_move_possibilites +=1

                        #Decide who moves above and below
                        if total_move_possibilites > 0:
                            #Reset time
                            seating_plan[current_aisle][row][INTERNAL_COCK] = 0

                            #frick knows what is happening here but it works so it stays
```

```
                                    move = numpy.argwhere(priorities == numpy.amax(
    priorities))
                                    possible_moves[(random.choice(move))[0]](row,
    current_aisle,False)


                        # I fricking HATE INDENDATION


                        #Get total amount of move posibilites
                        #total_move_possibilites = is_person_above_moving  +
    is_person_below_moving + is_person_right_moving


                    for current_aisle in range(0, WIDE_WING_SEATS,7):


                        # Move down
                        for seat in reversed(range(0,2)):
                            # Loops through all 37 rows

                            #Move towards aisle
                            for row in range(0,WIDE_WING_ROWS):
                                if seating_plan[seat+current_aisle][row][
    INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT:
                                    move_down(seat+current_aisle,row)

                                    #Incrasese internal clock


                        # Move up
                        for seat in range(5,7):
                            # Loops through all 37 rows

                            #Move towards aisle
                            for row in range(0,WIDE_WING_ROWS):
                                if seating_plan[seat+current_aisle][row][
    INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT:
                                    move_up(seat+current_aisle,row)
                            #Incrasese internal clock
                    for i in range(WIDE_WING_SEATS):
                        for k in range(WIDE_WING_ROWS):
                            seating_plan[i][k][INTERNAL_COCK] += TIME_STEP


                elif boarding_type == NORMAL:

                    #Exit square
                    if seating_plan[AISLE_INDEX][0][PRIORITY] != -1 and seating_plan
    [3][0][INTERNAL_COCK] >= TIME_TO_MOVE:
                        #Empty square
                        seating_plan[AISLE_INDEX][0] = [-1,0]

                        #End code shot
                        left_plane += 1
                        if left_plane == NUM_ROWS*NUM_SEATS:
                            test_cases.append(total_time)

                            break

```

```
                #Aisle handling code
                for row in range(0,NUM_ROWS):
                    #Check if aisles place is empty
                    if seating_plan[AISLE_INDEX][row][PRIORITY] == -1:


                        priorities = [0,0,0]
                        possible_moves = [aisle_take_above, aisle_take_below, aisle_take_right ]
                        total_move_possibilites = 0

                        #Get things to check
                        is_person_above_moving = seating_plan[AISLE_INDEX-1][row][PRIORITY] != -1 and seating_plan[AISLE_INDEX-1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT

                        if is_person_above_moving:
                            priorities[0] = priority_weightings[seating_plan[AISLE_INDEX-1][row][PRIORITY]-1]
                            total_move_possibilites +=1

                        is_person_below_moving = seating_plan[AISLE_INDEX+1][row][PRIORITY] != -1 and seating_plan[AISLE_INDEX+1][row][INTERNAL_COCK] >= TIME_TO_MOVE_PAST_SEAT
                        if is_person_below_moving:

                            priorities[1] = priority_weightings[seating_plan[AISLE_INDEX+1][row][PRIORITY]-1]
                            total_move_possibilites +=1

                        #Prevent indexing error
                        if row != NUM_ROWS-1:
                            is_person_right_moving = seating_plan[AISLE_INDEX][row+1][PRIORITY] != -1 and seating_plan[AISLE_INDEX][row+1][INTERNAL_COCK] >= TIME_TO_MOVE
                        else:
                            is_person_right_moving = 0

                        if is_person_right_moving:

                            priorities[2] = priority_weightings[seating_plan[AISLE_INDEX][row+1][PRIORITY]-1]
                            total_move_possibilites +=1


                        #Get total amount of move posibilites
                        #total_move_possibilites = is_person_above_moving  + is_person_below_moving + is_person_right_moving

                        #Decide who moves above and below
                        if total_move_possibilites > 0:
                            #Reset time
                            seating_plan[AISLE_INDEX][row][INTERNAL_COCK] = 0

                            #frick            knows what is happening here but it works so it stays
                            move = numpy.argwhere(priorities == numpy.amax(priorities))
```

```
                            possible_moves[(random.choice(move))[0]](row,False,
    False)



                # Move down
                for seat in reversed(range(0,2)):
                    # Loops through all 37 rows

                    #Move towards aisle
                    for row in range(0,NUM_ROWS):
                        if seating_plan[seat][row][INTERNAL_COCK] >=
    TIME_TO_MOVE_PAST_SEAT:
                            move_down(seat,row)
                            #Incrasese internal clock


                # Move up
                for seat in range(5,7):
                    # Loops through all 37 rows

                    #Move towards aisle
                    for row in range(0,NUM_ROWS):
                        if seating_plan[seat][row][INTERNAL_COCK] >=
    TIME_TO_MOVE_PAST_SEAT:
                            move_up(seat,row)
                            #Incrasese internal clock

                for i in range(NUM_SEATS+1):
                    for k in range(NUM_ROWS):
                        seating_plan[i][k][INTERNAL_COCK] += TIME_STEP





            total_time += TIME_STEP

            #Update render comment out if not using
            if VISUALIZER: update_render(seating_plan)
    print(text + str(sum(test_cases)/len(test_cases)))
    rows.append(test_cases)
    return test_cases
#Types
render_type = [intalize_render,intalize_render_widebody,intalize_render_two_thing
    ]
bag_shit_type = [bag_shit, bag_shit_wide,bag_shit_Two]
locker_shit_type = [check_locker_space,check_locker_space_wide,
    check_locker_space_Two]
#What thing to do
NORMAL = 0
WIDEBODY = 1
TWO = 2
#Test stuff
N_TEST_CASES = 50
VISUALIZER = True
TIME_STEP = 0
boarding_type = TWO
#Data csv
import csv
```

```
fields = []
rows = []
index = []
#Add the indexing
for i in range(N_TEST_CASES):
    index.append(i)

rows.append(index)
#Vroom
seating_plan = []

'''
#Flat body
off_the_plane(random_deboard, 'Random: ')
off_the_plane(sections, 'Sections: ')
off_the_plane(reverse_wilma, 'Reverse Wilma: ')
off_the_plane(generate_front_to_back, 'Front to back Row: ')
off_the_plane(back_to_front, 'Back to Front Row: ')
# field names add whatever field names that you are creating data for
fields = ['Index','Random Group Adjusted', 'Front to back - sections', 'Reverse
   Wilma', 'Front to Back Row', 'Back to Front Row']
file_name = 'narrow.csv'


#Wide body
off_the_plane(random_deboard_widebody, 'Random: ')
off_the_plane(sections_widebody, 'Sections: ')
off_the_plane(reverse_wilma_widebody, 'Reverse Wilma: ')
off_the_plane(generate_front_to_back_widebody, 'Front seat to back seat: ')
off_the_plane(back_to_front_widebody, 'Back seat to Front seat: ')
off_the_plane(across_widebody, 'Across: ')
# field names add whatever field names that you are creating data for
fields = ['Index','Random Group Adjusted', 'Front to back - sections', 'Reverse
   Wilma', 'Front to Back Row', 'Back to Front Row', 'Across']
file_name='widebody.csv'
'''
# field names add whatever field names that you are creating data for
fields = ['Index','Back to front', 'Sections', 'Random', 'Reverse Wilma', 'Front
   to back']
off_the_plane(two_reverse_sections_360, 'back to front')
#off_the_plane(two_reverse_sections_360, 'Sections')
#
#off_the_plane(two_front_to_back, 'front to back')
#off_the_plane(two_reverse_wilma_widebody, 'reverse wilma')
#off_the_plane(two_random, 'random')
file_name='twoaisles.csv'

'''
nbsensitivity = []

for i in range(0,41):
    NAUGHTY_BOY_COEFFICIENT = (i*2.5)/100

    # put method wanted in here
    nbsensitivity.append(off_the_plane(back_to_front, 'back to front: '))

    print('for test with NB coefficient {}'.format((i*2.5)/100))
'''
'''
#Makes the shotinto colums honestly magic
```

```
rows = zip(*rows)
#Create the rows
with open(file_name, 'w', newline='') as f:

    # using csv.writer method from CSV package
    write = csv.writer(f)

    write.writerow(fields)

    write.writerows(rows)
'''
```

Appendix F

```
import random
import matplotlib.pyplot as plt
import numpy
import math

# visualizer things

# render stuff that I don't understand
def intalize_render():

    global plane

    #Absolute mess of code
    image = []
    for i in range(NUM_SEATS+len(AISLES)):
        subimage = []
        for k in range(NUM_ROWS):
            if k % 2 == 0:
                subimage.append(-1)
            else:
                subimage.append(0)

        image.append(subimage)


    fig,ax = plt.subplots(1,1)
    plt.set_cmap('OrRd')
    print(image)
    image = numpy.array(image)

    im = ax.imshow(image)
    number_of_runs = range(1,NUM_ROWS)    # use your actual number_of_runs
    ax.set_xticks(number_of_runs, minor=False)
    ax.xaxis.grid(True, which='major')




    ax.set_yticks(numpy.arange(0.5, NUM_SEATS+len(AISLES)+.5, 1).tolist(), minor=
   False)
    ax.yaxis.grid(True, which='major')

    if plane == 'wide wing':
        ax.set_yticklabels(['A','B','C','Aisle','D','E','F','G','H','I','Aisle','
   J','K','L','M','N','O','Aisle','P','Q','R','S','T','U','Aisle','V','W','X'])
```

```
    elif plane == 'narrow body':
        ax.set_yticklabels(['Row A','Row B','Row C','Aisle','Row D','Row E','Row F'])
    elif plane == 'two entrance two aisle':
        ax.set_yticklabels(['Row A','Row B','Aisle','Row C','Row D','Row E','Aisle','Row F','Row G'])
    ax.set_ylim(top=-0.5)


    ax.set_xticks(numpy.arange(0.5, NUM_ROWS+.5, 1).tolist(), minor=False)
    ax.xaxis.grid(True, which='major')

    xticklist = []
    #Create list of numbers between
    for i in range(NUM_ROWS):
        if ((i+1) % 5 == 0) and (i != 0):
            xticklist.append(str(i+1))
        else:
            xticklist.append('')

    ax.set_xticklabels(xticklist)
    ax.set_xlim(left=-0.5)

    return im,fig
def update_render(seat_plan):

    visualizer = []
    for i,column in enumerate(seat_plan):
        visualizer.append([])
        for seat in column:
            if i not in AISLES:
                if seat != -1:
                    visualizer[i].append(0)
                else: visualizer[i].append(-1)
            else:
                if seat != '':
                    visualizer[i].append(0)
                else: visualizer[i].append(-1)



    im.set_data(visualizer)
    fig.canvas.draw_idle()
    plt.pause(0.01)










# --------
# stuff to board the plane with (given a boarding queue)
# --------

# calculate time taken to get to seat if someone in the way
def get_past_people(seating_plan, passenger, current_row):

```

```
    # number of people blocking seats
    N=0
    time_to_stop_blocking_aisle = 0


    # aisle seat
    for aisle in AISLES:
        if abs(passenger[1] - aisle) == 1:
            time_to_stop_blocking_aisle += TIME_TO_MOVE_PAST_SEAT
    # middle or window seat: people are in the way
    else:

        for aisle in AISLES:

            if passenger[1]-aisle == -3:


                # if aisle seat taken IMPORTANT to check aisle seat first so f is
    maximised
                if seating_plan[passenger[1]+1][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=1
                # if middle seat taken
                if seating_plan[passenger[1]+2][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=2

                break

            elif passenger[1]-aisle == -2:

                # if aisle seat taken
                if seating_plan[passenger[1]+1][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=1

            # window seat F
            elif passenger[1]-aisle == 3:
                # if aisle seat taken IMPORTANT to check aisle seat first so f is
    maximised
                if seating_plan[passenger[1]-2][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=1
                # if middle seat taken
                if seating_plan[passenger[1]-1][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=2

            # middle seat B
            elif passenger[1] == 2:
                # if aisle seat taken
                if seating_plan[passenger[1]-1][NUM_ROWS-current_row-1] != -1:
                    N+=1
                    f=1


        if N==0:
            time_to_stop_blocking_aisle = TIME_TO_MOVE_PAST_SEAT
        else:
            time_to_stop_blocking_aisle += TIME_TO_SIT_OR_STAND +
```

```
    TIME_TO_MOVE_PAST_SEAT*(N+f+1)

    return time_to_stop_blocking_aisle, N
# stow in overhead lockers
def check_locker_space(passenger, current_row, lockers, passengers_loaded_bags, aisle=0):

    # if passenger has no baggage
    if passenger[2] == 0:
        return 0

    # if on right side of aisle

    for aisle in AISLES:

        if abs(passenger[1]-aisle) <= 3:

            correct_aisle = aisle
            break


    if passenger[1] > correct_aisle:
        if [passenger[0],passenger[1]] not in passengers_loaded_bags:
            nbins = lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][1]
            lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][1] += passenger[2]
            passengers_loaded_bags.append([passenger[0],passenger[1]])
        else:
            nbins = lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][0]-passenger[2]
    else:

        if [passenger[0],passenger[1]] not in passengers_loaded_bags:
            nbins = lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][0]
            lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][0] += passenger[2]
            passengers_loaded_bags.append([passenger[0],passenger[1]])
        else:
            nbins = lockers[AISLES.index(correct_aisle)][NUM_ROWS-current_row-1][0]-passenger[2]
    # derivations in writeup


    if passenger[2] == 1:
        t = (4)/(1-(0.8*nbins)/6)
    if passenger[2] == 2:
        t = (4)/(1-(0.8*nbins)/6) + (2.25)/(1-(nbins+1)/6)

    return t

# board the plane
def board_the_plane(boardingQueue, family=False):
    # initialize seating plan, top queue (if multiple aisles) and overhead lockers
    seating_plan =  [[-1 for _ in range(NUM_ROWS)] for _ in range(NUM_SEATS + len(AISLES))]
    for aisle in AISLES:
        seating_plan[aisle]=['' for _ in range(NUM_ROWS)]
```

```python
    top_queue = ['' for _ in range(NUM_SEATS+len(AISLES))]
    lockers = [[[0,0] for i in range(NUM_ROWS)] for j in range(len(AISLES))]
    seated = []
    passengers_loaded_bags = []

    n_passengers = len(boardingQueue)

    total_time=0

    # this is false for all scenarios except where families are prioritized
    if family == False:
        time_to_move = TIME_TO_MOVE
    else:
        time_to_move = FAMILY_TIME_TO_MOVE

    while True:

        #print(seating_plan)

        # loop through top queue
        for current_column, passenger in enumerate(reversed(top_queue)):

            if passenger != '':

                # increase internal clock
                passenger[3] += TIME_STEP

                # check if passenger in right aisle and thus they can seat
                if (NUM_SEATS+len(AISLES)-current_column-1) in AISLES and abs(passenger[1]-(NUM_SEATS+len(AISLES)-current_column-1))<=AISLES[0]+1:

                    # move into aisle
                    if seating_plan[NUM_SEATS+len(AISLES)-current_column-1][0] == '' and passenger[3] >= time_to_move:
                        #reset internal clock
                        passenger[3]=0
                        seating_plan[NUM_SEATS+len(AISLES)-current_column-1][0] = passenger
                        top_queue[NUM_SEATS+len(AISLES)-current_column-1]=''

                else:
                    # if passenger in front has moved
                    if top_queue[NUM_SEATS+len(AISLES)-current_column] == '' and passenger[3] >= time_to_move:
                        # move people along
                        top_queue[NUM_SEATS+len(AISLES)-current_column] = passenger
                        top_queue[NUM_SEATS+len(AISLES)-current_column-1] = ''

                        # reset internal clock
                        passenger[3] = 0

        for aisle in AISLES:

            # loop through aisle from back to front
            for current_row,passenger in enumerate(reversed(seating_plan[aisle])):

                if passenger != '':

```

```
                    # increase internal clock
                    #print(seating_plan[aisle])
                    passenger[3] += TIME_STEP

                    # check if passenger in right row and thus they can seat
                    if passenger[0] == NUM_ROWS - current_row:

                        # if passenger has baggage
                        time_to_stow = check_locker_space(passenger, current_row, lockers,passengers_loaded_bags)


                        # time it takes to stop blocking aisle and number of people in the way
                        try:
                            time_to_stop_blocking_aisle=passenger[5]
                        except:
                            time_to_stop_blocking_aisle, N = get_past_people(seating_plan, passenger, current_row)
                            passenger.append(time_to_stop_blocking_aisle)


                        # make sure there is an empty space
                        if N==2 and current_row != 0 and seating_plan[aisle][NUM_ROWS-current_row] != '' and current_row != 0:
                            time_to_wait_for_spot_in_aisle += time_to_move - passenger[3]
                        else:
                            time_to_wait_for_spot_in_aisle=0


                        # if time to wait has finished i.e. SIT DOWN BE HUMBLE
                        if passenger[3] >= time_to_stop_blocking_aisle + time_to_stow + time_to_wait_for_spot_in_aisle:

                            seating_plan[passenger[1]][passenger[0]-1] = passenger
                            seated.append(passenger)
                            #print(seated)

                            # set queue place to empty
                            seating_plan[aisle][NUM_ROWS-current_row-1]=''



                    else:
                        # if passenger in front has moved
                        if seating_plan[aisle][NUM_ROWS-current_row] == '' and passenger[3] >= time_to_move:
                            # move people along
                            seating_plan[aisle][NUM_ROWS-current_row] = passenger
                            seating_plan[aisle][NUM_ROWS-current_row-1] = ''

                            # reset internal clock
                            seating_plan[aisle][NUM_ROWS - current_row][3] = 0

        if VISUALIZER: update_render(seating_plan)


        total_time += TIME_STEP
```

```

        if len(seated) == n_passengers:
            return total_time


        if top_queue[0] == '' and len(boardingQueue)!=0:


            # only considered in method where families board first.
            if family == True and boardingQueue[0] == 'b':
                time_to_move = NON_FAMILY_TIME_TO_MOVE
                boardingQueue.pop(0)

            #Set first place in isle to the first passenger in the seat data
    seating_plan[3] then remove it from seat data
            top_queue[0] = boardingQueue[0]
            boardingQueue.pop(0)


# luggage
def assign_luggage():
    return random.choices([0,1,2], weights=BAG_COEFFICIENT, k=1)[0]

# naughty boy
def is_not_disobedient():
    return random.randrange(100) > NAUGHTY_BOY_COEFFICIENT*100
# create a group size
def group_size(group_weights):

    return random.choices([1,2,3], weights=group_weights, k=1)[0]

# return average of list
def average(x):
    return sum(x)/len(x)







# create order of boarding
def create_boarding_order_for_section_but_with_groups(boarding_section,
    other_section1, other_section2, start_row, end_row):
    current_group_member = 0
    current_group_section = 1
    current_group_size = 1
    boarding_section.append([])

    for row in range(start_row,end_row+1):
        for seat in range(0,NUM_SEATS+len(AISLES)):

            if seat not in AISLES:

                current_group_member += 1

                if current_group_section == 1:
                    boarding_section[-1].append([row,seat,assign_luggage(),0])
                elif current_group_section == 2:
                    other_section1[-1].append([row,seat,assign_luggage(),0])
```

```
                elif current_group_section == 3:
                    other_section2[-1].append([row,seat,assign_luggage(),0])


                if current_group_member == current_group_size:


                    for aisle in AISLES:


                        if seat-aisle in [2,3]:
                            if current_group_section == 1:
                                boarding_section[-1].reverse()
                            elif current_group_section == 2:
                                other_section1[-1].reverse()
                            elif current_group_section == 3:
                                other_section2[-1].reverse()

                        if seat-aisle in [-3,-2,1]:
                            current_group_size = group_size((
    SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,THREESOME_COEFFICIENT))
                        elif seat-aisle in [-1,2]:
                            current_group_size = group_size((
    SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,0))
                        elif seat == 6:
                            current_group_size = 1

                        current_group_member = 0


                    if is_not_disobedient():
                        current_group_section = 1
                        boarding_section.append([])


                    # else they try board during different sections
                    else:
                        if random.randrange(100) < 50:
                            current_group_section = 2
                            other_section1.append([])
                        else:
                            current_group_section = 3
                            other_section2.append([])






# create order of boarding for doing windows first
def create_boarding_order_for_aisle(boarding_section, other_section1,
    other_section2, seats):
    for seat in seats:

        for row in range(1,NUM_ROWS+1):

            # if passenger is not useless
            if is_not_disobedient():
                boarding_section.append([row, seat, assign_luggage(), 0])
            # else they try board during different sections
            else:
```

```
                if random.randrange(100) < 50:
                    other_section1.append([row, seat, assign_luggage(), 0])
                else:
                    other_section2.append([row, seat, assign_luggage(), 0])

# create order of boarding for doing windows first using groups
def create_boarding_order_for_aisle_but_with_groups(boarding_section, other_section1, other_section2, seats):


    for seat in seats:
        # window seats
        for row in range(1,NUM_ROWS+1):


            # check if item in group already appended
            if (not any([row,seat] in x for x in boarding_section)
                and not any([row,seat] in x for x in other_section1)
                and not any([row,seat] in x for x in other_section2)):

                # if passenger is not useless
                if is_not_disobedient():

                    for aisle in AISLES:

                        if aisle-seat == 3:

                            current_group_size = group_size((70,50,20))

                            if current_group_size == 3:
                                boarding_section.append([[row, seat],[row, seat+1],[row, seat+2]])
                            elif current_group_size == 2:
                                boarding_section.append([[row, seat],[row, seat+1]])
                            else:
                                boarding_section.append([[row, seat]])
                        elif aisle-seat==-3:
                            current_group_size = group_size((70,50,20))

                            if current_group_size == 3:
                                boarding_section.append([[row, seat],[row, seat-1],[row, seat-2]])
                            elif current_group_size == 2:
                                boarding_section.append([[row, seat],[row, seat-1]])
                            else:
                                boarding_section.append([[row, seat]])

                        elif aisle-seat==2:
                            current_group_size = group_size((80,40,0))
                            if current_group_size == 2:
                                boarding_section.append([[row, seat],[row, seat+1]])
                            else:
                                boarding_section.append([[row, seat]])
                        elif aisle-seat==-2:
                            current_group_size = group_size((80,40,0))
                            if current_group_size == 2:
                                boarding_section.append([[row, seat],[row, seat
```

```
-1]])
                                else:
                                    boarding_section.append([[row, seat]])

                            else: boarding_section.append([[row, seat]])

                            break

                    # else they try board during different sections
                    else:
                        if random.randrange(100) < 50:
                            other_section1.append([[row, seat]])
                        else:
                            other_section2.append([[row, seat]])



# reduce boarding queue capacity due to Covid
def cull_boarding_queue(boarding_queue):
    #this function has two aims: reduce capacity due to COVID, and remove any
    seats not included in planes

    # first see if need to cull the seats that would be in grid of planes, but
    not there
    # remove them here as easier than having to not add them in the first place
    in every method
    global plane
    if plane == 'wide wing':
        for index,passenger in enumerate(boarding_queue):
            # Seats A B C V W X in rows 1-3
            if passenger[0]-1 in [0,1,2] and passenger[1] in [0,1,2,25,26,27]:
                del boarding_queue[index]
    elif plane == 'narrow body':
        for index,passenger in enumerate(boarding_queue):
            # Row 1 seats D E F
            if passenger[0]-1 in [0] and passenger[1] in [4,5,6]:
                del boarding_queue[index]
    if COVID_CAPACITY==0:
        return boarding_queue
    target_to_kill = math.floor((COVID_CAPACITY)*NUM_SEATS)
    for row in range(NUM_ROWS):

        killed = 0
        for index,passenger in enumerate(boarding_queue):
            if passenger[0] == row:
                killed += 1
                del boarding_queue[index]
            if killed==target_to_kill:
                break
    return boarding_queue



# ----------------
# BOARDING METHODS
# ----------------

# boarding in random order
def random_boarding():

```

```
    test_cases = []
    for _ in range(N_TEST_CASES):
        boardingQueue = []
        for row in range(1,NUM_ROWS+1):
            for seat in range(NUM_SEATS+len(AISLES)):

                # assign bag based on probability that passenger has bag
                if seat not in AISLES: boardingQueue.append([row,seat,
    assign_luggage(),0])

        random.shuffle(boardingQueue)

        test_cases.append(board_the_plane(boardingQueue, AISLES))

    print('Random: ', sum(test_cases)/len(test_cases))




def random_boarding_with_groups():

    test_cases = []
    for _ in range(N_TEST_CASES):
        boardingQueue = [[]]

        current_group_member=0
        current_group_size = group_size((SINGLE_PRINGLE_COEFFICIENT,
    COUPLES_COEFFIENCT,THREESOME_COEFFICIENT))

        for row in range(1,NUM_ROWS+1):


            for seat in range(0,NUM_SEATS+len(AISLES)):

                if seat not in AISLES: boardingQueue[-1].append([row,seat,
    assign_luggage(),0])

                current_group_member += 1

                if current_group_member == current_group_size:

                    for aisle in AISLES:


                        if seat-aisle in [2,3]:
                            boardingQueue[-1].reverse()

                        if seat-aisle in [-3,-2,1]:
                            current_group_size = group_size((
    SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,THREESOME_COEFFICIENT))
                        elif seat-aisle in [-1,2]:
                            current_group_size = group_size((
    SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,0))
                        elif seat == 6:
                            current_group_size = 1

                        current_group_member = 0
                        boardingQueue.append([])

                        break
```

```
        random.shuffle(boardingQueue)

        # flatten groups

        boardingQueue = [j for sub in boardingQueue for j in sub]

        #print(boardingQueue)

        boardingQueue = cull_boarding_queue(boardingQueue)

        test_cases.append(board_the_plane(boardingQueue))

    print('Random with groups: ', sum(test_cases)/len(test_cases))
    print(test_cases)
    return average(test_cases)


# sectional boarding but with groups
def section_boarding_with_groups():

    test_cases = []
    amf, fma = [],[]
    for _ in range(N_TEST_CASES):

        aft,middle,front = [],[],[]

        # aft section
        create_boarding_order_for_section_but_with_groups(aft,middle,front,
    A_SEC_START,A_SEC_END)
        # middle section
        create_boarding_order_for_section_but_with_groups(middle,aft,front,
    M_SEC_START,M_SEC_END)
        # front section
        create_boarding_order_for_section_but_with_groups(front,middle,aft,
    F_SEC_START,F_SEC_END)


        random.shuffle(aft)
        random.shuffle(middle)
        random.shuffle(front)


        #print(boardingQueue)
        boardingQueue = aft+middle+front
        boardingQueue = [j for sub in boardingQueue for j in sub]
        boardingQueue = cull_boarding_queue(boardingQueue)
        amf.append(board_the_plane(boardingQueue))
        #boardingQueue = front+middle+aft
        #boardingQueue = [j for sub in boardingQueue for j in sub]
        #fma.append(board_the_plane(boardingQueue))


    print('Sectional amf: ', average(amf))
    #print('Sectional fma: ', average(fma))

    return(average(amf))
```

```python



# boarding by seat but allowing groups to board together
def seat_boarding_with_groups():

    test_cases = []
    boardingQueue=[]
    for _ in range(N_TEST_CASES):

        window,middle,aisle = [],[],[]

        # window seats
        #window_seats = [aisle-3 for aisle in AISLES] + [aisle+3 for aisle in AISLES]
        #create_boarding_order_for_aisle_but_with_groups(window,middle,aisle,window_seats)
        # middle seats
        middle_seats = [aisle-2 for aisle in AISLES] + [aisle+2 for aisle in AISLES]
        create_boarding_order_for_aisle_but_with_groups(middle,window,aisle,middle_seats)
        # aisle seats
        aisle_seats = [aisle-1 for aisle in AISLES] + [aisle+1 for aisle in AISLES]
        create_boarding_order_for_aisle_but_with_groups(aisle,window,middle,aisle_seats)

        random.shuffle(window)
        random.shuffle(middle)
        random.shuffle(aisle)


        window = [j for sub in window for j in sub]
        middle = [j for sub in middle for j in sub]
        aisle = [j for sub in aisle for j in sub]

        boardingQueue1 = window+middle+aisle
        for x in boardingQueue1:
            if x not in boardingQueue:
                boardingQueue.append(x)

        for passenger in boardingQueue:
            passenger.append(assign_luggage())
            passenger.append(0)

        boardingQueue = cull_boarding_queue(boardingQueue)
        test_cases.append(board_the_plane(boardingQueue))

    print('By seat with groups: ', sum(test_cases)/len(test_cases))

    return average(test_cases)

def prioritize_groups_boarding():

    test_cases = []
    for _ in range(N_TEST_CASES):
        mainBoardingQueue = [[]]
        priorityQueue=[]
        boardingQueue=[]
```

```
        current_group_member=0
        current_group_size = group_size((SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,THREESOME_COEFFICIENT))
        current_boarding_section = 2
        for row in range(1,NUM_ROWS+1):


            for seat in range(0,NUM_SEATS+len(AISLES)):

                if seat not in AISLES:

                    if current_boarding_section == 1:
                        priorityQueue[-1].append([row,seat,assign_luggage(),0])
                    else:
                        mainBoardingQueue[-1].append([row,seat,assign_luggage(),0])

                    current_group_member += 1

                    if current_group_member == current_group_size:

                        for aisle in AISLES:


                            if seat-aisle in [2,3]:
                                if current_boarding_section == 1:
                                    priorityQueue[-1].reverse()
                                else:
                                    mainBoardingQueue[-1].reverse()

                            if seat-aisle in [-3,-2,1]:
                                current_group_size = group_size((SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,THREESOME_COEFFICIENT))
                            elif seat-aisle in [-1,2]:
                                current_group_size = group_size((SINGLE_PRINGLE_COEFFICIENT,COUPLES_COEFFIENCT,0))
                            elif seat-aisle == 3:
                                current_group_size = 1


                            break

                        current_group_member = 0

                        if current_group_size == 3:
                            if random.randrange(100) > 80:
                                mainBoardingQueue.append([])
                                current_boarding_section = 2
                            else:
                                priorityQueue.append([])
                                current_boarding_section = 1
                        elif current_group_size == 2:
                            if random.randrange(100) > 20:
                                mainBoardingQueue.append([])
                                current_boarding_section = 2
                            else:
                                priorityQueue.append([])
                                current_boarding_section = 1
                        elif current_group_size == 1:
                            if random.randrange(100) > 5:
```

```
                                mainBoardingQueue.append([])
                                current_boarding_section = 2
                            else:
                                priorityQueue.append([])
                                current_boarding_section = 1



        random.shuffle(mainBoardingQueue)
        random.shuffle(priorityQueue)
        # flatten groups
        boardingQueue =  priorityQueue+['b']+mainBoardingQueue
        boardingQueue = [j for sub in boardingQueue for j in sub]

        #print(boardingQueue)

        test_cases.append(board_the_plane(boardingQueue, True))

    print('Priortizing groups: ', average(test_cases))





















# modified steffen method
def steffen_modified_method():

    test_cases = []
    for _ in range(N_TEST_CASES):


        rightOdd,leftOdd,rightEven,leftEven = [],[],[],[]
        steffenPerfected = [rightOdd,leftOdd,rightEven,leftEven]
        # window seats
        for row in range(1,NUM_ROWS+1):
            for seat in range(-3,4):
                #naughty boy
                if not is_not_disobedient() and seat != 0:
                    steffenPerfected[random.randrange(0,2)].append([row,seat+3,
    assign_luggage(),0])

                elif (seat > 0): #right side
                    steffenPerfected[(row%2)*2].append([row,seat+3,assign_luggage
    (),0])
                elif (seat < 0): #left side
```

```
                    steffenPerfected[(row%2)*2+1].append([row,seat+3,assign_luggage(),0])



        random.shuffle(steffenPerfected[0])
        random.shuffle(steffenPerfected[1])
        random.shuffle(steffenPerfected[2])
        random.shuffle(steffenPerfected[3])



        steffenPerfected = [j for sub in steffenPerfected for j in sub]
        test_cases.append(board_the_plane(steffenPerfected))

    print('By steffen perefected: ', sum(test_cases)/len(test_cases))

    return(average(test_cases))









plane = 'narrow body'

if plane == 'narrow body':
    NUM_ROWS = 33
    NUM_SEATS = 6
    AISLES = [3]
    F_SEC_START = 1
    F_SEC_END = 11
    M_SEC_START = 12
    M_SEC_END = 22
    A_SEC_START = 23
    A_SEC_END = 33
elif plane == 'wide wing':
    NUM_ROWS = 14
    NUM_SEATS = 24
    AISLES = [3,10,17,24]
    F_SEC_START = 1
    F_SEC_END = 5
    M_SEC_START = 6
    M_SEC_END = 9
    A_SEC_START = 10
    A_SEC_END = 14
elif plane == 'two entrance two aisle':
    # simulating only the back half of the plane
    NUM_ROWS = 20
    NUM_SEATS = 7
    AISLES = [2,6]
    F_SEC_START = 1
    F_SEC_END = 7
    M_SEC_START = 8
    M_SEC_END = 14
    A_SEC_START = 15
    A_SEC_END = 20
```

```
elif plane == 'two entrance two aisle first class':
    # total loading time for 2E2A plane will be 2E2A + 2E2A first class
    NUM_ROWS = 3
    NUM_SEATS = 6
    AISLES = [2,5]

# CHANGE THESE FOR SENSITIVITY
BAG_COEFFICIENT = (20,80,10)
NAUGHTY_BOY_COEFFICIENT = 0.18
COVID_CAPACITY = 0.5 #0, 0.3 0.5 or 0.7

N_TEST_CASES = 100
VISUALIZER = True
TIME_STEP = 0.1

# all measured in standard units (m,s,m/s etc)
AVERAGE_WALKING_SPEED = 0.8
AVERAGE_SEAT_PITCH = 0.78
TIME_TO_MOVE = AVERAGE_SEAT_PITCH / AVERAGE_WALKING_SPEED
FAMILY_TIME_TO_MOVE = 1.3 * TIME_TO_MOVE
NON_FAMILY_TIME_TO_MOVE = TIME_TO_MOVE
TIME_TO_SIT_OR_STAND = 2.5
TIME_TO_MOVE_PAST_SEAT = 2
# proportions of group sizes
SINGLE_PRINGLE_COEFFICIENT = 70
COUPLES_COEFFIENCT = 20
THREESOME_COEFFICIENT = 0

if VISUALIZER: im,fig = intalize_render()

#Data csv
import csv
fields = []
rows = []
index = []
#Add the indexing
for i in range(N_TEST_CASES):
    index.append(i)

rows.append(index)



# BOARDING METHODS: comment out if not using
#random_boarding()
#section_boarding()
#seat_boarding()
#random_boarding_with_groups()
#section_boarding_with_groups()
#seat_boarding_with_groups()
#prioritize_groups_boarding()

# steffen methods can only be used with narrow body
#steffen_deeznuts()
#steffen_modified_method()

#naughty_boy_sensitivity()
#bag_sensitivity()

# field names add whatever field names that you are creating data for
```

```
fields = ['Index','Section']
```