The Use of Matlab in Creating M/M/n/∞ Queuing Theory Model

Viliam Mojský[1], Karol Achimský[1]

[1]Department of communications, Faculty of Operations and Economics of Transport and Communications, University of Žilina, Univerzitná 1, 010 26 Žilina, Slovakia

Abstract In our research we dealt with the creation of an algorithm designed to simulate M / M / n queuing systems with endless queues. The algorithm was created in Matlab. When creating the algorithm, we proceeded according to scientific literature, according to which we transformed the formulas into the Matlab algorithm. This process also included the optimization and creation of new outputs based on the analysis of the generated data. The research output is a Matlab algorithm suitable for creating simulations of M / M / n / ∞ systems.

Keywords simulation methods, Matlab, queueing theory, M/M/n/∞

JEL C63

1. Introduction

Through modelling it is possible to create models that simulate the behaviour of a real system. It is a powerful tool that can simulate the likely behaviour of a real system. Outputs from simulations can be used to support decision making.

In the research we dealt with the creation of a simulation model of queuing system in Matlab. In our research we have chosen Markov type of queuing theory models M/M/n/∞. Our aim is to create an algorithm that can be used to obtain statistical data when simulating a real queuing theory system.

2. Methods and materials

The aim of the research was to create a simulation model of M/M/n/∞ queue. The created model will simulate the behaviour of the real queuing system at the post office, which will provide relevant outputs based on which it will be possible to evaluate the system [1,2].

The algorithm will be created in Matlab, in the LiveScript environment. It will give us access to code, outputs, and stored variables on a single screen. This will allow us to analyse the results better and faster and determine the way forward. The code is divided into three parts - initialization, calculation and output. This will make the code clearer and make it easier to identify possible errors and fix it [3,4].

When creating the model and obtaining statistical outputs from the simulation, we followed the scripts System Modelling. They describe an algorithm for creating a simulation model with N lines and an infinite queue, along with algorithms and formulas for obtaining statistical data [5,6]

2.1. Statistics collected

2.1.1. The average number of customers in queue

$$PPC = \frac{\sum PC_i * T_i}{\sum T_i}$$

where:
PPC – The average number of customers in queue
PC_i – The number of customers in queue i
T_i – The duration of i-th time interval, expressed as:

$$T_i - T_i - T_{i-1}$$

Based on the above expression of Ti, the expression is in scripts modified as follows:

$$PPC = \frac{\sum F_i * (T - T0)}{\sum (T - T0)} = \frac{\sum F_i * (T - T0)}{Tk}$$

where:
F_i – The size of queue i
T – Status start time
T0 – Status end time
Tk – Simulation end time

2.1.1. Average server load

$$PVL = \frac{\sum VL_i * T_i}{\sum T_i}$$

where:
PVL – Average server load
VL_i – Line i utilization

Based on the above expression of Ti, the expression is in scripts modified as follows:

$$PVL = \frac{\sum L_i * (T - T0)}{\sum (T - T0)} = \frac{\sum L_i * (T - T0)}{Tk}$$

where:
L_i – Line i status

2.1.1. The average number of customers in the system

$$PPZ = \frac{\sum PZ * T_i}{\sum T_i}$$

where:
PPZ – The average number of customers in the system
PZ – The current number of customers in system

Based on the above expression of Ti, the expression is in scripts modified as follows:

$$PPZ = \frac{\sum PZ * (T - T0)}{\sum (T - T0)} = \frac{\sum PZ * (T - T0)}{Tk}$$

2.1.1. Average waiting time

$$PCC = \frac{\sum_{i=1}^{N} CC_i}{NC}$$

where:
PPC – Average waiting time
CC_i – Customer wait time in line i
NC – Number of waiting customers in line i

The relationship to PCC calculation implies that CC is the sum of the waiting times of all customers in line i. The waiting time of the customer is calculated by capturing the time of his arrival, the line of the queue to which he joined and the position in the queue. Subsequently, after finishing the customer service, the line is queued forward. Once again, we record type of event and the time it occurred. We repeat this until the customer is queued to the server, at which point we stop the waiting time measurement in the queue [5,6].

This sequence is incorporated into the algorithm as follows:

Event: Arriving in the queue

- Recording of the line the customer has accessed, position in the queue and time to variable VFj [Fj]. This variable represents the size of the queue j, where the customer's arrival time is stored. The [Fj] index indicates the customer's position in the queue.

Event: Finishing of customer service

- The difference between the current time in system T and the time spent by the first customer in the queue j VFj [1] is added to CCj. This will return the time he has spent in the queue since his arrival. At the same time, the number of customers served in line j is recorded in the variable NCj.

Event: Start of serving next customer

- After finishing the customer service, the server is ready to serve another customer from the queue. This means that the first customer from the queue will move to the server, so the waiting time was recorded in the previous step. The entire queue will move 1 customer forward.

Event: Simulation end

- After completion of the simulation, the CCj variable stores the total time all customers spent in the queue. Therefore, it is necessary to divide each variable CCj by the variable NCj in order to obtain the average queue time per customer from that line.

Within the algorithm, we used the same variable names as those used in the formulas.

3. Solution

Within the solution we created a simulation model queuing theory in Matlab. When creating the model, we decided to use data from the real system to evaluate the simulation. The data were drawn from diploma thesis: Utilization of queuing theory at a selected post office by Ing. Silvia Ďutková. Compared to the original data, we made a change at the variable Tk, which we calculated in minutes per month.

While writing the code, we first created the code exactly according to the algorithms of the System Modelling scripts. According to these algorithms, we created a flow chart of the first algorithm design. Subsequently, we modified and optimized this code for Matlab. Along with the optimization, we thought of other possibilities of using the obtained variables, which expanded the number of obtained outputs. As we wrote the code, we also found that the statistical data formulas are inaccurate and produce slightly inaccurate and distorted results. We found this error while writing the code. Subsequently, based on the analysis of the outputs and their comparison with the data measured in the real system, we proposed adjustments to the algorithms, which we included among the results of the research.

The development of the algorithm was as follows:

1. Creation of the first version of the code.
2. Code optimization.
3. Add new outputs.
4. Modifying formulas and subsequent code optimization.
5. Creation of the final version of the algorithm.

3.1. Algorithm inputs

The inputs listed in Table 1 were used during the algorithm creation.

Table 1. Algorithm inputs

Time interval	08.00 – 09.00
Hours	1
Workdays	25
Tk (min)	1500
N	6
mi0 (min)	0,84
mi1 (min)	3,23

Tk is the duration of the simulation in minutes. The simulation starts in 0 and takes Tk minutes, so 1500 minutes. N is the number of service lines, there are 6 in the simulation. The customer enters the system approximately every 0.84 minutes and the approximate length of customer service on the line is 3.23 minutes.

3.2. First algorithm version

We divided the algorithm into three logical parts: Initialization part, Simulation part and Output part. We have decided for three parts because they logically divide the program into inputs, operations and outputs. We have described these parts in their own chapters. While writing the first version, we followed algorithms and schematics from System Modelling scripts, where they were stated in Pascal programming language.

3.2.1. Initialization part

The initialization section defines the variables that are used in the algorithm. Variables are assigned their initial states, which can change during the operation of the algorithm.

There are three commands at the beginning of the algorithm: clear, tic, and rng. The clear command is used to clean up the Workspace, that is, to delete all saved variables from programs that were previously run and did not "dump" the variables. This ensures that the results of previous executions of the algorithm will not affect its current performance and will not distort the results and cause errors and bugs. Next follows the tic command, it is a paired tic-toc command. The command is used to measure the execution time of the code between the two commands. The tic code is given at the beginning of the code, toc at the end of it, to monitor the execution rate of the code and to monitor the impact of optimization measures. The rng () command is used to determine the random seed by which the program will generate pseudo-random numbers. We used it to see if editing the code affected the results, which would be undesirable in case of optimization.

At the start of initialization, the code is divided into variables declared by the user and automatically populated variables.

In the section with variables declared by the user, there are 4 variable declarations that are set by the user according to the characteristics of the system he wants to simulate. The variables have the same names as their equivalents in the queuing theory and have already been described in the previous chapter. These are the variables N, Tk, mi0 and mi1. We used data from Table 1 to populate them. These are the only variables that the user sets. Changing other variables will most likely affect the simulation results and will yield irrelevant outputs. In addition to these 4 variables, the user can change the value in the rng () function to a constant number to achieve the same results with each iteration, or set the command to rng ('shuffle') to obtain a different random seed and a series of pseudo-random numbers.

Next, there is a section with automatically populated variables that contains variables that the user should not modify. Their population is automated within the algorithm. At the beginning there is a section with the preallocation of variables. Here are preallocated some one-dimensional variables. Also, the variable N is shifted here. The N = N + 1 command is used to shift the numbering in the simulation. The problem is that the variable indexes depends on the number of servers. We have eg. 4 servers: 1., 2., 3., 4., so we will have mi1, mi2, mi3, mi4 and also the statistics will be for each server separated. The problem is the arrival of the customer, where is used variable mi with index 0 (mi [0]) and also the time variable TU uses index 0. Matlab is not able to work with index 0, all fields must start with index 1. Therefore we had to shift the customer arrival time to index mi [1] and time of arrival of the customer to TU[1], so we had to move all other indexes in the algorithm 1 higher. This means that each line and queue are represented by an index 1 greater than their actual number. Thus, line 1 has an index 2, line 2 has an index 3, and so on. Next, there are 3 declarations of one-dimensional variables cpz, ppz and pz. Cpz is a variable used to determine the total number of customers that were in the system during the simulation. The variables ppz and pz are the statistical variables described before. All variables were pre-allocated with 0.

In the next section, are multi-dimensional variables filled with initial values. Multi-dimensional variables are variables that are of the field type, and store data by indexes, with indexes according to servers in the system. At the beginning of the algorithm is a for loop that iterates from 2 to N, filling the vector variables with zeros. These vector variables include F [i] (line queue), L [i] (line state), and TU [i] event time [i]. F is set to zero because at the beginning the queue is empty. Also, the server is set to 0 because there are no customers, so the server is not serving. When the server starts serving a customer, the variable switches to state 1. TU [i] is the time of event i (arrival of the customer [1], or finishing customer service on line i [2 : N]. Initially, the TU variables are set to Tk, that is, to time of simulation end, to make sure that the first event that occurs in the system is the arrival of the customer TU [1]. The variable mi with index [2: N] stores the value mi [1] defined by the user. The variables ppc and pvl are named according to the formulas in the methods chapter. The maxf [i] variable is used to record the size of the maximum queue on server [i]. The maxft [i] variable is used to record the time when the maximum queue occurred on the line[i] for the first time. Next are declarations of 4 initial variables to start the cycle. The mi0 value stores the user-defined mi0 value. The variable T indicates the current time in the system and is initially set to 0. Next, TU [1] = 1 is assigned to determine that the first customer will enter the system within 1 minute of the start of the simulation. Next is assignment of the value T0, which is used in the formulas to obtain the statistical values.

The last command in this section performs the initial population of the variables with statistics through the for loop. Here, the ppc and pvl variables are first populated.

3.2.2. Simulation part

The simulation part of the algorithm performs the simulation and ensures the collection of statistical data. It is also divided into several parts.

The first part ensures detection and selection of the next event. This is an event model, so selecting the nearest event is crucial for the simulation to work properly. At the beginning of the code is a while loop, which encloses the entire algorithm and determines its completion. It has condition T<= Tk, that is, it will run if the current time T in system is less than or equal to the end time Tk. The cycle is followed by a temporary determination of the next TUk event. TU [1] is selected as the nearest event, and k is assigned a value of 1. This is followed by a cycle that iterates from 2 to N, which compares TU [i] with TUk. This determines whether other event occurs earlier than the TU [1] event. If the result is true, the appropriate TU [i] is stored in the TUk and the index value [i] is assigned to the index [k]. After the cycle, the next event that occurs in the system and its server index is detected. Subsequently, the value T is stored in the variable T0 and the value TUk is stored in T, which becomes the new current system time. This way, the current time of system T and the previous time in system T0 are determined.

Next, there is a decision branch that begins with an if statement to determine if the next event is the arrival of the customer or finishing of customer service. Customer arrives only in case of TU [1] where k = 1, all other indexes mean finishing of customer service on one of the lines. Therefore, the cycle determines whether k = 1. If so, it is the customer's arrival to the system. If not, it is the finishing of customer service.

If a new customer arrives, it is expected to queue, so it is initially looking for the smallest queue (the algorithm assumes that each customer prefers to queue to the shortest queue). Searching is similar to searching for the next event. Initially, it is determined that the smallest queue F [j] is queue F (2) (2 is the first-server index, because of the index shift). The index of the shortest queue j = 2 is also determined. This is followed by a for cycle from 3 to N, which compares queues on other lines with Fj. If any queue is shorter, its value is stored in Fj and its index is stored in j. In this way, the shortest queue index is determined. The following are three formulas for ppz, pz, and ppc statistics, which follow these formulas. On arrival, the number of customers (pz) increases. With these three patterns, it is important that they record events before the queue is increased. They are located before the queue is raised and lowered. Thus, they accurately record the time of occurrence and the time of event change. These are followed by an increase in the F [j] queue by one customer who has just arrived in the system. Next, after the que increases, we find out whether the current queue is the largest one that occurred on the line. The is a simple if statement. If the result is true, the current queue size is stored in maxf and the current time in the system is recorded in maxft. This is followed by an increase in cpz statistics. The last command in the branch generates the time when a new customer enters the system and it is stored to TU [1].

In the if branch of finishing customer service, ppz, pz, and pvl statistics are initially calculated. When the customer leaves, the current number of customers (pz) decreases. This is followed by a command that turns off server and sets it to L (k) = 0 because the customer has been served and left. The last command assigns Tk + 1 to TU (k), ensuring that the variable is not selected as the nearest event to occur.

Next, both if the branches are followed by a branch that checks if there are customers in the queues and if they are, it detects the status of the given server. This is done using two if statements placed in the for loop. The for loop ensures that the detection is performed for all servers. The first if determines whether there are customers in queue F[i]. If yes, then a second if follows, which detects whether the line is busy [1] or free [0]. If it is free, it means that the first customer from the queue can move to the server and the queue decreases. Otherwise, everything remains as it was. If the line is free, a series of commands follows. First, the ppc statistic is recorded according to the formula. Next, the F [i] queue will be shortened by one customer who has moved to the server. Next, pvl statistics are recorded. This is followed by a command that turns on the L [i]. The last command generates the time how long it will take to serve the customer who has just moved from queue to the server and stores the result to TU [i].

This is the last step in the while loop. After all the commands have been executed, the evaluation an performed to determine whether the cycle will continue or end.

In the simulation part, there is a collection of statistics after the while cycle. It runs through a for loop. This ensures that all recent events that occur in the system are recorded in ppc and pvl statistics.

3.2.3. Outputs part

The last part of the algorithm serves to display the simulation outputs in text form. Most statements are enclosed in a for loop to run for each server statistics. The first output is the average number of customers ppc. Before the value is outputted, it is calculated according to formula to obtain its final value. The output is executed by the disp () command, which ensures that the inserted string is displayed in the output window. The string to be inserted is previously declared to the vetappc variable. In it we constructed a sentence that outputs the line and its average queue per hour. Other statements work on a similar principle. Next is listing the maxf max queue along with the maxft time in which it occurred. This is followed by a listing of the average and total number of customers in the system. The last output is the average line load in%.

3.2.4. Evaluation of outputs

We obtained several outputs from the simulation using the algorithm. The results of the simulation showed statistics of the system with current input characteristics. Analysis of these results revealed several inconsistencies. There is a big difference between maximum and average queues on lines. This can be explained by, that most of the time, the queues were empty, and sometimes the system was overwhelmed by customers and the queues reached the maximum values recorded in the output. This implies that customers who entered the system went straight into the service and did not wait in line. The average number of customers in the system says that there was almost always a customer on the server. This results from ppz statistic. There were 6 servers, so if the average number of customers in system was 6.5, then each server was likely to be regularly occupied and there was approximately 1 customer in the queue. The total number of customers in the system was 1804, which is a plausible quantity due to the simulation time and does not contradict the statistics. There is a problem with average server load (pvl). Previous slight irregularities could be justified, but the pvl indicator violates these reasons. Percentage of server usage is in direct conflict with the average number of customers in the system, which says that on average, there has always been a customer in system. According to the percentual utilization of the servers, the busiest server was utilized at 13%, so approximately 87% of the time it was free. Server 6 was even free for 98% of the time of the simulation. The indicator is also very contradictory when it comes to comparing statistics within it. The first three lines have approximately the same% utilization. This results in a very strange statistic When the customer entered the system, before it was served, two other customers entered the system and they occupied lines 1, 2 and 3. For a long time, nothing happened and the situation repeated again, three customers entered the system at once and again occupied the first three lines. This is the most likely explanation. If they enter independently, gradually and not in triplets, they would in most cases go to the first line, because it was free in 87% of the time and waited for the customer. Thus, the explanation for this statistic is that customers most often entered the system in triplets. Less often in quadruples, quintuplets and sixes, but these situations must have also occurred because other lines have also been used. For fun, the worst luck had the customer who entered the system in 255.21 minutes, because then he had to queue on line 6 for the 3rd position. And according to statistics, line 6 was 98% of the time without a customer. These statistics are very unlikely. We do not claim that such situations could not have occurred, everything is possible with random generation, but it is very unlikely. Especially the part where the triplets had to go into the system at the same time, so the third customer had to come before the service of first customer was finished and when he left, there was a long pause during which the system was empty and then three customers entered the system again.

When the number of lines N = 6 was replaced by N = 2, the server usage of server 1 increased to 14% and server 2 to 16%. The average number of customers on queue was 121 on queue 1 and 125 on queue 2. The average number of customers in the system was 497.

These results seemed very unlikely, so we decided to address them in the next chapter in optimization by vectorization. This will allow us to create vectors from the variables that can be plotted and thereby check how many customers were on the servers and when they were active.

3.3. Algorithm optimization

To optimize the variables, we vectorized and preallocated them, and used GPU parallelization to speed up the rendering of outputs.

Vectorization is the transformation of commands into vectors. This is a procedure designed to get rid of unnecessary cycles in order to speed up program execution. Matlab is a vector language and it is more natural and faster for it to work with vectors than cycles. The second important change was the transformation of fields into column vectors. By default, the vector is created as a row, so it stores values in one row. However, Matlab is able to work with columns faster, so we transformed them. Another important change was the preallocation of variables. Preallocationg means pre-populating the memmory in which vector variables will be stored. This preallocation of the memmory will significantly speed up the program execution time. In the algorithm, vectors are expanded by 1 field at each iteration to obtain a complete list of variables. Matlab create vector copies by creating a copy of an vector, adding a new value to the vector, and then saving that copy as the original vector. This is a quick operation, but with many repetitions, it will start to take a lot of time. In addition, over time, the amount of data that needs to be copied increases. Preallocation will speed it up, because instead of creating a new vector on every iteration it will only create it once and then will only rewrite the preallocated values with values from the system.

We verified these statements with our own algorithm, where we tested the speed of variable population. We tried to populate a non-vectorized and vectorized, row and column variable without preallocation and with preallocation. We have generated and stored 100,000 pseudo-random numbers in these variables. The worst result was at non-vectored and not preallocated , column variable with an average time of 3.02 seconds. The fastest was a vectorized, preallocated, column variable with a time of 0.0033 seconds. Therefore, we decided to optimize the algorithm code using the procedure mentioned above.

Within the simulation algorithm, we did not find a suitable cycle that could be accelerated by parallelization without compromising its reliability. However, we managed to implement GPU parallelization in generating graphical outputs. GPU parallelization allows the use of graphics processor cores to speed up the execution of operations. The condition is that the GPU must have CUDA cores that are only in NVidia graphics cards. All changes made to the algorithm are described in the following chapters.

3.3.1. Vectorization and preallocation

We chose vectorization to column vectors, and preallocation for several reasons. The first was to optimize the code for Matlab and speed up code execution. The second was the possibility of obtaining further outputs.

At the beginning, we transformed multidimensional variables such as F [i], L [i] and mi [i]. The transformation was done by adding a second index to the variable declaration. In the first algorithm, the variables were populated by for loop by assignment operations, such as the variable F [i] = 0. The transformation was done by adding index 1 to the second position, so the new declaration has the form: F [i, 1] = 0. The second step was to get rid of the for loop. Matlab can work with vectors much faster than cycles. There are two solutions for this operation. A function or direct vector declaration. Function zeros fills the variable with zeros according to the specified range, so F = zeros (2: N, 1) creates a vector F with rows from 2 to N and with 1 column and fills it with zeros. The second option is a direct declaration F (2: N, 1) = 0, which does the same. Our experiments and the experience of other users in the Matlab forum show that the direct declaration is a little faster. The acceleration depends on the size of the vector being created. In our case, there is probably an improvement by one hundredth, or perhaps a thousandth of a second, but it is still an improvement, so we decided to declare variables directly. We transformed all variables that ranged from 2 to N depending on the line in the system as described. In this way, we eliminated several for loops, created column variables, and all variables are preallocated, which together greatly optimizes the algorithm.

3.3.2. Transformation of formulas

Given the inaccuracies in the output of the previous algorithm, we decided that we need a graphical representation of the queue, servers, and number of customers in the system to compare them with the calculated statistics. Therefore, we decided to modify the formulas so that they do not work with single values. Instead, these values are stored in variables. At the end of the algorithm, the outputs are calculated according to formulas. An example of formula for average number of customers from the algorithm:

```
ppc(j)=ppc(j)+F(j)*(T-T0);
```

The formula gradually adds to itself the multiplication of the size of the current queue and the difference between the current and the previous time. Based on this process, we transformed the formulas into the following form:

```
h(i)=h(i)+1;
FF{i,1}(h(i),1)=F(i);
TTF{i,1}(h(i),1)=T;
TT0F{i,1}(h(i),1)=T0;
ppc(i)=sum(FT{i,1}(:,1).*(TTF{i,1}(:,1)-TT0
F{i,1}(:,1)));
ppc(i)=(ppc(i)/Tk);
```

We replaced the original variables with vectors. In order to draw a custom graph of each variable for each line, we had to create a separate vector for each line. We saved them in a vector, so we created an array of vectors. To index individual values within a vector, we have created an variable h [i] that varies for each server and indexes the values in the vectors. For example, FT {i, 1} (h (i), 1) = F (i), where i = 3 and h [i] = 50. This means that an operation is in progress, e.g. customer's arrival in line 2 (3-1). 50 values are already recorded in the vector. The new value will have an index of 51 (50 + 1). Thus, the variable FF {3,1} (51,1) writes the value F (3), ie the current queue size on queue 2. The FT variable stores the size of the queue, the TTF variable stores the time T for the FT variable, and the TT0F variable to store the time T0 for the variable FT. There are also variables LT, TTL and TT0L in the algorithm, which have similar use, but for servers. The reason for separating TTF time for queues from TTL time for lines is the difference in when they are recorded. The time T for the queue is recorded in the customer arrival branch, the T part for the server in the finishing service branch. There could be a mismatch and were therefore separated. The same procedure was applied to the PZ variable, to which we added the PZT variable to store the time T.

The preallocation of these variables was problematic. The number of their elements cannot be predetermined precisely because their number depends on randomly generated values and can range from 0 to several times the time Tk, depending on the input characteristics of the system. Officially, there is no way to precisely predict the size of a variable with a variable of unknown number of elements. Therefore, we have decided to apply approximated preallocation. In this part of the algorithm, we estimate the size of the variables and based on this estimate, the size of the variables is preallocated. In the algorithm we made these estimates based on system observations. We have determined the capacity of Tk * 2 by the variables, which can be interpreted so that customer arrives or is served every half minute. This is unlikely, especially in a simulation of a real system, and the system input parameters used do not suggest this. In any case, the algorithm should be designed for other input parameters where the quantities could be higher than in our case. Ideally, this would be determined by analysing the relationship between the basic input characteristics N: mi0: mi1: Tk. This analysis would probably require more time to carry out, so we will carry it out as part of further research.

The approximate preallocation proceeds in 2 steps:

1. Field preallocation
2. Vector preallocation

First, there is a preallocated field in which the vector values will be stored. The field is preallocated by the cell function, in which its size is inserted. In the second part, the vectors themselves are preallocated by the zeros command to preallocate Tk * 2 positions with a value of 0.

But there is a problem. By approximating the size of an element, it is likely that we will not preallocate the right amount of memory. This could cause problems when calculating statistics, where the formula would count with zero values. This would not affect the result, n + 0 = n. A problem would arise when plotting graphs where even zero points would be plotted. In addition, in terms of optimization, zero points would take up memory space. Therefore, after completing the simulation part and before the output part, it is necessary to insert another part that will modify the preallocated vectors and remove unnecessary zero elements from them. For this we used the property of the variable TT, which records the time T. T is the current time and at the end of the simulation it equals or is greater than the maximum value and the simulation ends. So, we can assume with certainty that if we find the maximum of TT, we find its last non-zero value within the vector. Any subsequent value must be zero. Based on this assumption, we created a for loop that iterates from 2 to N. The use of the loop was necessary in this case because we could not find a way to vectorize vector variables stored in cell fields. The maximum value position within the vector is searched for in the cycle. When found, all three vectors associated with it are cut off based on its value. So, if we are looking for the maximum for TTF {2,1} - i. the second vector in the TTF field. The result is stored as a coordinate in the variable Yft (i, 1) by the command [~, Yft (i, 1)]. The result is stored as a coordinate in two variables specified by square brackets. The maximum value is stored in the first place and its coordinate in the second. The maximum value is not needed, so its saving is denied using the ~ character and only its coordinate is stored. Subsequently, this coordinate is used to determine the point at which the vector is to be cut.

In this way, a sufficient amount of memory is preallocated for the variables. This may not seem efficient at first glance, but it is, and the result is optimizing code speed, even though we've added a few extra lines to the code. This is because when Matlab expands a vector with a new variable - it must always copy the vector, paste the variable to the copy, delete the old one, and replace it with the new one. These are 4 operations that execute as many times as many variables are written to the vector. On the other hand, the preallocation takes place once, then the preallocated values are only overwritten with new ones and, in the end, the excess space is cut off. So if it were generated 500 values, so without preallocation, about 4 * 500 = 2000 operations would take place, while with approximate preallocation only 1 + 500 + 1 = 502 operations. The efficiency of this method was verified in tests where we tested the speed of the algorithm with and without pre-location. As a result, the implementation of the preallocation algorithm was accelerated by an average of 2.5 seconds, which we rate as an improvement.

In the ways outlined in this chapter, we have modified and optimized all multidimensional variables

3.3.3. Command vectorization and other optimizations

In statement vectorization, we tried to replace cycles in the algorithm. We have described the vectorization of cycles from the initialization part in the previous section. There were also several cycles that could be vectorized within the computational part. Here we have optimized the search for the closest event and the search for the smallest queue. We replaced the for loops with the min function, which looks for the minimum and stores its size and position. We have done the same for the smallest queue, but we only saved the index.

Other optimization activities include the use of short-circuit operator. We used it when deciding whether someone is in the queue and whether the line is serving. Originally it was solved through two if statements. For example, the short-circuit operator is && (logical and). In this context, it is used to determine whether both terms are true. The difference between using & and && is that && evaluates the second expression only if the first is true. So, if the first expression is false, the second does not even evaluate and saves time. Using this command, we merged both if statements into one with optimized evaluation.

3.3.4. GPU parallelization

Vector variables obtained by vectorization allow us to create graphical outputs from the simulation. Creating graphical outputs is hardware-intensive in the case of a large number of plots. This process is parallel by default but is limited by the number of CPU cores. Therefore, Matlab makes it possible to speed up this process by using graphical cores. CPUs usually have from 2 to 8 cores, the number of threads can be doubled. GPUs have several hundreds or thousands of cores depending on the type of card. However, parallelization also copies and creates temporary variables, which slows down the process. Therefore, it should always be considered whether parallelization pays off. Usually, small and unpretentious amounts of operations are not worthwhile, while larger ones are.

In the outputs are plotted many coordinates on the graph area, so we decided to use GPU parallelization. We have added graphical outputs to the end of the algorithm. Initially, it checks to see if a compatible graphics card is available on your computer. This is done by attempting to create gpuArray, and if successful, true is returned. If this field cannot be created, an exception occurs, and a catch branch returns false. Based on the result of this attempt, values are assigned to variables in cycles. We made the assignment through for loops because we could not find a way to vectorize cell fields. We made these assignments so only one variable i sused in the graphical outputs.

3.4. New simulation outputs

Using new variables, it was possible to create new outputs in graphical form. The graphs were outputted through for loops, in which we enclosed all the properties and parameters that should be plotted for the graph. First is the figure function. It is used to separate the plotted graphs from each other, because when plotting from the cycle, an error occurs, in which only the last plotted graph is displayed, and the others are discarded. The graphs are displayed using the painters function, which is suitable for 2D graphs. The position parameter is also set, in which the resolution of the graph is determined. This is followed by the plot command, where the x and y axis parameters are inserted, and the hexadecimal colour is set to blue. This is followed by a hold command with the off parameter, which is used to determine whether the graphs will be drawn into each other and overlap, or each graph will have its own space. The off parameter specifies that each graph will be separate. Next is the title statement, which creates the title of the graph based on the inserted string. The number of the server for which the graph is displayed is inserted in the algorithm. The following are the xlabel and ylabel commands, which create descriptions of the x and y axes based on the entered text.

The graphical outputs are created for the queue size, server state and number of customers in the system. The queue size and server status are plotted separately for each line, the number of customers in the system is total for the system. From the obtained outputs, the queue size output on server 1 in Figure 1 and the status of server 1 in Figure 2 are displayed.

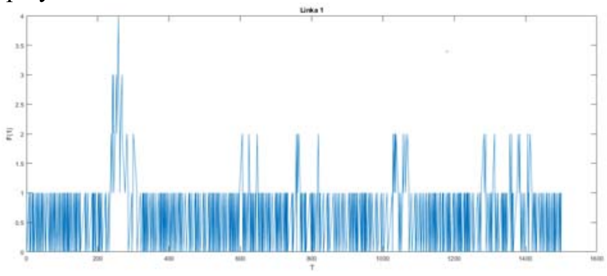

Figure 1. The queue size of queue 1 in time

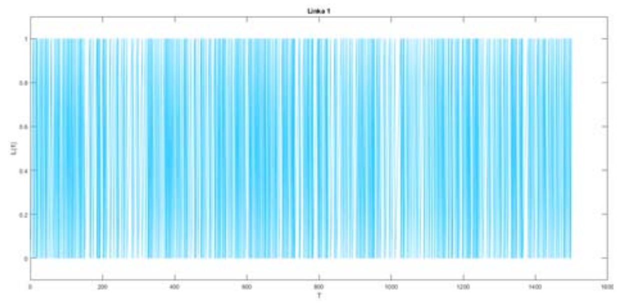

Figure 2. Utilization of server 1

As it can be seen from Figures 1 and 2, the server 1 was almost fully utilized. However, this differ with the statistics obtained, which is strange because they are drawn from the same data from which the statistics are calculated.

To check the results, we placed three more vectors in the algorithm to record the values of the queue and lines at each point in the system. For comparison, server 1 has been plotted from approximately 900 intercepts in Figures 1 and 2. In the new algorithm, approximately 4500 points were captured for Line 1, from which the graph was drawn. It is likely that some of them are duplicates. Nevertheless, it is a much more accurate graph. Given the simulation time, which is 1500 minutes, it can be argued that for line 1 its state was captured approximately every 1/3 minute, that is every 20 seconds. There is no difference between the data in the graphs, both showing the same values. We chose to keep the new vectors values because it plotted the data more accurately and without the oblique edges in the chart, from which it is unclear what they mean. We have also changed the function of plotting a plot from a plot to a stairs function, which creates "staircase" graphs that are more legible and without oblique edges. We also tested its use on data from previous vectors to avoid unnecessarily creating new, larger vectors, but there were inaccuracies in the graphs due to fewer points captured over time. Therefore, we decided to use the new vectors for plotting graphs.

3.5. Algorithm modification based on output analysis

Based on the outputs, we have determined that the formulas used to calculate the statistics are incorrect. By analysing the original formulas, we concluded that the formulas used in the algorithms were incorrectly derived from the original formulas. The problem was at time T0, which symbolizes the previous time in the system. The original formula assumes Tj-T (j-1). T (j-1) has been expressed as T0 and we think that there is a mistake. In the original formula, time T was a two-dimensional vector with times Tj and T (j-1) separately for each link. The reason for it is the connection to the TU time, which is separated for each server. Each server has its own service duration. Initially, T is stored in T0 and TU is stored in T. Thus, T0 is the time of the previous event and T is the time of the next closest event. However, these events may not be performed by the same server. The formula clearly works with one queue, that is, the utilization of just one server. Therefore, in our opinion, it is also necessary to work with the time for the line and not with the general system time. For example, let's say that a finishing customer service event is on server 5 at time T = 20. The event executes and finishes, and the next event selection begins again. The algorithm evaluates the finishing customer service event on server 1 at T = 21 as closest, so it stores the value 20 in T0 and the value 21 in T0. In the formula: ppc (j) = ppc (j) + F (j) * (T- T0), the queue size of server 1 is used, the time T that corresponds to the server 1, but it uses the time T0 that corresponds to the server 5. This greatly shortens the real service time that was different from the time used in the formula. Therefore, we propose to use indexed times T and T0 to record for each line separately and to avoid mixing times between lines.

In the algorithm we solved it directly without having to record the time T0. Matlab has the diff function, which is designed to subtract vector elements in a [i] -a [i-1] way, so we only recorded the time T [i]. So, the difference time T [i] -T [i-1] was calculated using the diff function. We added 0 to the resulting field, because this is the first value of the unmodified vector T and throws it off in the calculation because it is the first value bound to T [i-1] and there is nothing to subtract from it. Therefore, it is added to the beginning of the vector. The new formula in the code:

```
ppc(i)=(sum(FT{i,1}(:,1).*([0 ;diff(TTF{i,1}(:,1))]))/sum(([0 ;diff(TTF{i,1}(:,1))]));
```

Despite the difficult looking code, this is the same formula as in the methodology, only with the correction of the time T.

After this adjustment, the graphs matched the text output, but we noticed another drawback. If we set the number of lines to 16, then lines 10 - 15 have 0% utilization and the first lines create a queue with a waiting time of 0.5 to 2 minutes. It does not make sense. In a real system, if the customer enters a system, he would go to the line where he would be immediately served. We focused on the part where the queue is selected by the customer. The logic of the algorithm is that the customer selects the smallest queue when he arrives at the system. That is correct but let us consider the following situation. Lines 1, 2 and 3 are serving a customer and line 4 is empty. Customers are in queue on lines 1 and 2. There are no customers on lines 3 and 4. If we choose only the nearest minimum queue, the first option is line 3, because its queue is empty. But the algorithm does not take into account that there is a customer on the server and so the customer who entered the queue must wait. While line 4 is completely empty and unused. This is nonsense in terms of a real system. If I have to choose a cash register in the store, I will choose the one that is empty. Therefore, we propose to modify the algorithm so that when selecting a queue, it takes into account not only the queue, but also the state of the line. The server has 2 states. State 0 if empty and 1 if serving. This can be used to determine the total number of customers on the server. Therefore, we calculate the minimum of the sum of the queue and server occupancy to get the total number of customers on the line. In the previous example, the customer will be moved to line 4 because it is empty and will go straight from the queue to the operator without the need for waiting. Now that we have adjusted the algorithm to the number of lines 16, the simulation showed that there was no queue anywhere and the waiting time on the line was 0, so there was an immediate service. We consider this adjustment to be a key one from the point of simulation feasibility, because it brings simulated customers' behaviour closer to reality.

3.6. Final algorithm version

All previous chapters led to the final version of the algorithm, which is the result of the research. To compare the versions of the algorithms, we measured the speed and memory that the algorithms occupied with the same input parameters set out in Table 1. The measurement results are shown in Table 2.

Table 2. Algorithm version comparison

	First run (s)	**Subsenquential runs (s)**	**Memory requirement (byte)**	**Used memory to process speed ratio (MB/s)**
First version	0,48	0,30	982	3,2
Final version without graphs	0,84	0,7	786 578	1097,35
Final version with graphs	3,11	1,23	1 451 497	1152,42

The values were measured on a computer with 4 core i5-7500 processor, 16GB DDR4 2400 MHz RAM in dual channel, NVidia GTX 1050 Ti graphics card and M2 SSD with 900 MB / s write, 1500 MB / s read speed.

Despite an approximately 800-fold increase in memory requirements, there was only an approximately 2-fold increase in the time required to execute the algorithm on the first and final version without graphs. In graph ploting, the first run time is considerably longer, and the space requirement has also increased. The used memory to process speed ratio is how fast the data in the algorithm was processed in MB / s (larger is better). Here can be seen the effect of code optimization on execution speed. It could be interesting to have the final version in the unoptimized version with the same amount of data or running the algorithm on a more powerful computer.

Two statistics were added to the final version of the algorithm, the average queuing time (pcc) and the total number of customers served (NC). Both statistics are output only in text form. Together with other outputs, they provide comprehensive information about the behaviour of the created system model. The algorithm produces both text and graph outputs. These are created on the basis of collected statistics. The simulation model collects two types of statistics, statistics integrated over time and statistics not integrated over time. For statistics integrated over time, the time they occurred is stored. Based on it, it is possible to create graph outputs that show the time sequence of events of a given variable. Non-time-integrated statistics record the average or total values of selected variables. They are mostly used to create text outputs.

Compared to the original version of the algorithm, we consider the modified, final version as a step forward. It provides multiple outputs, allows to view the development of statistics over time, and all the variables obtained are still available at the end of the program and can be processed into other outputs. Due to the highier amount of data to be recorded, the new version requires nearly 800 times more memory than the first version. However, its execution time is only about 4 times longer in subsequential runs. If ploting graphs is turned off, the execution time is only 2.5 times greater. The execution speed is reduced by code optimization, i.e., vectorization, prealocation, and parallelization. This way, the final version was successfully optimized, which is also confirmed by the column of the ratio of used memory to the speed of execution of the algorithm that has the highest data processing speed in the final version.

4. Conclusions

The aim of our research was to create a simulation algorithm that will simulate a queue theory system in Matlab. The aim was to create a model that will simulate the M/M/n/∞ system. When creating the algorithm, we first made a flowchart according to the algorithms from the literature. Then we have written the first version of the algorithm according to the diagram. Next, the algorithm was optimized for Matlab. While optimizing the algorithm, we encountered errors in the algorithms from the literature that caused incorrect outputs. We analysed the errors and suggested some corrections. By analysing the available vectors, we designed new outputs in the form of graphs showing system changes over time.

The research output is an algorithm enabling simulation of M / M / n / ∞ system. The algorithm provides the user with both textual and graphical outputs. Using the created algorithm, it is possible to perform simulations of real systems with N service lines and with infinite queue. Outputs from the algorithm can be used in solving queuing theory problems and research activities.

ACKNOWLEDGEMENTS

This article was published with the support of project EUREKA-E! 11158 U Health Auto-ID technológie a internet vecí na zvýšenie kvality zdravotníckych služieb.

This article was published with the support of project VEGA 1/0721/18 Výskum ekonomických dopadov vizuálneho smogu v doprave s využitím metód neurovedy.

REFERENCES

[1] R. Hušek, J. Lauber. Simulačné modely. Praha: SNTL/ALFA, 1987. 349s. ISBN 978-80-562-0075-9

[2] J. Smieško. Operačná analýza II. Základy teória hromadnej obsluhy. MC Energy Žilina, 1999. 190 s. ISBN 80-968115-6-8

[3] What is Matlab?. [online]. [citované 26.05.2019]. Dostupné na internete: https://nl.mathworks.com/discovery/what-is-Matlab.html

[4] What Is a Live Script or Function?. [online]. [citované 26.05.2019[. Dostupné na internete: https://nl.mathworks.com/help/Matlab/Matlab_prog/what-is-a-live-script-or-function.html

[5] Achimský, K., Čorejová, T., Fitzová, M. Kajánek, B. Projektovanie sietí v pošte I. Vysoká škola dopravy a spojov v Žiline. Edičné stredisko VŠDS, Žilina. 1995. 147s. ISBN 80-7100-238-0

[6] V. Achimská. Modelovanie systémov. Žilina: Žilinská univerzita v Žiline, 2011. 96 s. ISBN 978-80-554-0450-9

[7] S. Ďutková. Využitie teórie hromadnej obsluhy na vybranej poštovej prevádzkarni. Diplomová práca. Žilina. 2017. 82s. EČ: 28330420172014