Assignment 1

Name: Abdullah Alghamdi
Student ID: 1341516

Name: Sonola Sowemimo
Student ID: 1293430

Cmput 313

Simulator Description

Our simulator was developed in Python. We used a variety of libraries that includes:

- **random:** This was used to generate our random seed throughput, and also the random numbers to compare to e
- **math:** This was used to compute simple calculations throughout our simulator program
- **sys:** This was used in being able to work with other system interactions such as output.

Our simulator also used functions in being able to provide the user with the appropriate outcomes:

- **arguments()** - this is in charge of grabbing all the user inputs, this is also in charge of making sure the appropriate inputs are given, odds are you are faced with a program error if you enter a combinations of variables that do not go together. For example: giving and Independent argument along with a burst length not assigned to zero.
- **main() -** simply the main method in any program, no need to be further explained. This those majority, if not all the work in running the program smoothly, including calling every other functions that aids the program.
- **block_measures(K, F) -** this returns block, r. This function handles the situation of taking in the number of blocks, frame size in bits and breaks it into the block this calculating the set of trials.
- **check(initFrame, e, independent, N, b, block_size, parity) -** This is where everything is checked to make sure if they're in a good condition or bad. Checks bit error, blocks received correctly.
- **standardDev_Frames(average_frame_tx, T, succesfull_frames, total_frames) -** returns the standard deviation, in charge of calculating the standard deviation for each frame average.
- **standardDev_Thro(average_throughput, T, succesfull_frames, total_time, F) -** returns the standard deviation for each throughput average.
- **confidenceInts(x_bar, std, T) -** returns the confidence interval 1, confidence interval

In our main program the functions described above were used to calculate the various statistical and mathematical operations. First step of actions is to query the user for simulation specifications that they would like to run through the command line. After this step is done we go ahead and retrieve all this information by using the arguments() function. Our simulation starts by measure the size of the block and how many checkpoints we want to add to it. After that we start initializing the values in the frame depending on their error type. Furthermore, we check the bit errors in a frame during each trial by checking if the input is of type “I”, which means it is independent error. If so, we initialize the frame by generating random number and check if it is less or equal to e (error given in the input). On the other case, if the input is of type

“B”, we calculate e’ = B+N/N instead. If there is an error in the frame, we try transmitting it again for less than R time.

During this process, we calculate and keep track of the following arrays are variables:

1 - average_frame_tx: This is a list that calculate the average number of frames that were successfully received. By the end of every trial, we append the following:

average_frame_tx.append((time/(frame_size+A))/calculatedFrames)

Where :

- Time: is the overall time that includes transmitting and retransmitting processes.
- frame_size: the overall size of a frame in every trial
- calculatedFrames: is the number of successfully received frames per trial

2 - average_throughput: This is a list that calculate the average throughput per trial using the following code: (variables are defined above)

average_throughput.append((F* calculatedFrames)/time)

Using the two arrays we can derive the statistical calculations and overall result.

Project Scope:

Since this is a simulator made by students there is no doubt going to be some potential errors floating around and project scope on a much smaller scale. This simulator by no means fully represent a real life one in which case algorithms will be more complex and process of approaching HSBC also more complex. But this simulator does show us the basic flow of a HSBC program, something that might prove useful to students who decide to follow a career path dealing with networking. This being accompanied with note provided in project description ... “Noting the actual message bits are not generated and details of the HSBC and error decoding schemes are not implemented. We are only required to know how many bit errors occurred within a block/frame, in order to figure out how well the correction scheme will work. This simulation, ignores the time needed to do the error checking and to split message into blocks, etc. and only look at time to transmit the encoded blocks and receive back acknowledgements.” This also providing us with the possible error that since our program doesn't entail the whole system. It's bound to produce possible errors.

Analysis:

This is the portion of the assignment, were observations are noted with changing K values, along with alternating between Burst & Independent.

python3 c313_asn1.py -M (I or B) **-A** 50 **-K** (0, 1, 2, 10, 40, 100, 400, or 1000) **-F** 4000 **-e** (0.0001, 0.0003, 0.0005, 0.0007, or 0.001) **-B** (50, or 500) **-N** (5000, or 1000) **-R** 5000000 **-T** 5 1534546 2133323 377 456548 59998

throughput vs k with different e:

K this is changed from 0, 1, 5, 10, 40, 100 for the corresponding values of **e** 0.001, 0.003, 0.005, 0.007.

python3 c313_asn1.py I 50 **0** 4000 **0.0003** 0 0 5000000 5 1534546 213323 377 456548 59998

python3 c313_asn1.py I 50 **1** 4000 **0.0003** 0 0 5000000 5 1534546 213323 377 456548 59998

python3 c313_asn1.py I 50 **5** 4000 **0.0003** 0 0 5000000 5 1534546 213323 377 456548 59998

…

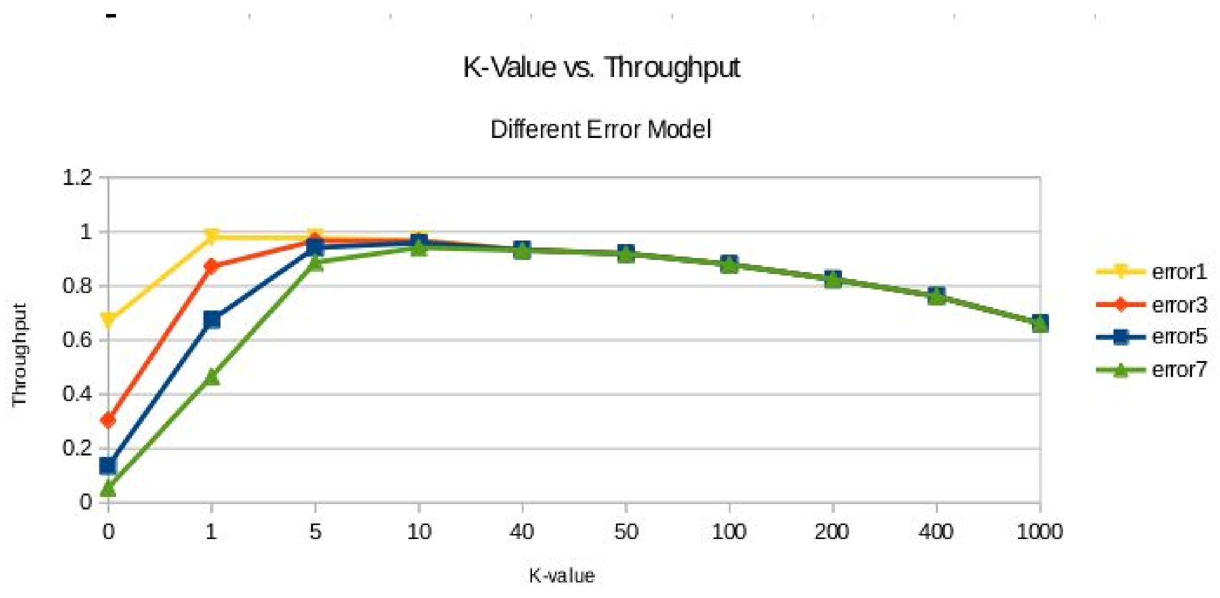
K is changed from 0, 1, 5, 10, 40, 100 for corresponding values of **B, N** (50, 5000 | 500, 1000 | 500, 5000 | 50, 1000) while **e** is kept constant at **0.0005.**

python3 c313_asn1.py B 50 **0** 4000 **0.0005 50 5000** 5000000 5 1534546 213323 377 456548 59998

python3 c313_asn1.py B 50 **1** 4000 **0.0005 50 5000** 5000000 5 1534546 213323 377 456548 59998

python3 c313_asn1.py B 50 **0** 4000 **0.0005 500 5000** 5000000 5 1534546 213323 377 456548 59998

...



This graph showing the process of K-Value being shown against the throughput with different errors.

- error1 - e 0.0001 | error3 - e 0.0003 | error5 - e 0.0005 | error7 - e 0.0007

The graphs shows how at an initial process it makes sense for a lower error input to be spitting out an higher throughput while the reverse works in the aspect of having a high error provides a smaller throughput. At the point of increasing the number of blocks, throughput significantly increases up to a certain point. When the blocks increases even more this then increases the use of error detection and correction. The chance that more errors are now generated per block becomes significantly less than blocks having a bigger size in bits. This is because each transmission transmits one block, and for lower block sizes, there are less possible error prone bits per total bits sent for each transmission. And since there are less possible errors, occurred errors are more likely to be less than 2, which makes the blocks more likely to be correctable and decreases the number of retransmissions.

The slope decreases after a block number of about 35. This is probably due to the fact that too many blocks being sent. This reachers over the sweet spot of accuracy and poses a new problem: using too much bit time and thus achieving the opposite effect, decreasing throughput

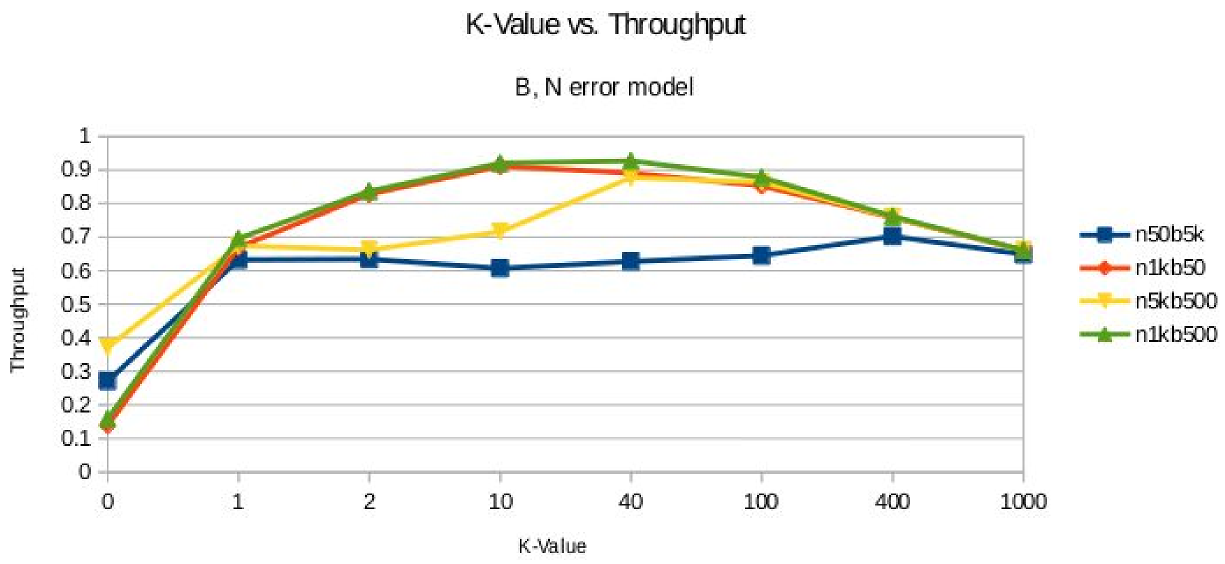
K Value **e = 0.0001**	**Throughput**	**Throughput** **CI Lower**	**Throughput** **CI Upper**
0	0.66809	0.64966	0.68651
1	0.97746	0.97620	0.97708
5	0.97664	0.97620	0.97708
10	0.96852	0.96852	0.96852
40	0.93240	0.93240	0.93240
100	0.87912	0.87912	0.87912
400	0.76190	0.76190	0.76190
1000	0.66116	0.66116	0.66116

K Value	**Throughput** **e = 0.0003**	**Throughput** **e = 0.0005**	**Throughput** **e = 0.0007**
0	0.30325	0.13579	0.05502
1	0.87161	0.67493	0.46675
5	0.96673	0.94100	0.88679
10	0.96580	0.95877	0.94101
40	0.93224	0.93192	0.93048
100	0.87912	0.87912	0.87880
400	0.76190	0.76190	0.76190
1000	0.66116	0.66116	0.66116

The initial table shows the K-value and the throughput along with its CI lower and CI upper.

In order to stay within bounds of page size the next table just shows the appropriate information, this being the throughput for e=0.0003, 0.0005, and 0.0007.

–



This graph shows the K-value against the throughput, keeping the **e** error constant and alternating within the B & N for the B bursts. No need testing with I independent since, this value is shows in the previous graph.

n50b5k = N50 B5000 | n1kb50 = N1000 B50 | n5kb500 = N5000B500 | n1kb500 = N1000B500

We are now faced with the situation of having n1kb500 reaching the most climax even though it starts off at the lowest initial point but all these end up falling to the same throughput value at around 1000.

Unlike previous graph we don't follow this process of this following a trend rather they have their own, this is likely due to the point of the b burst error model, this seems to affect the slope at a more significant level than the n burst error model. Since for the first point of having n50b5k it follows the lowest trend than compared to any other graph slope.

K Value **N=5000, B=50**	**Throughput**	**Throughput CI Lower**	**Throughput CI Upper**
0	0.27079	0.26330	0.27827
1	0.63256	0.61258	0.65254
2	0.63446	0.61956	0.64936
10	0.60719	0.59234	0.62204
40	0.62725	0.61721	0.63729
100	0.64458	0.63356	0.65560
400	0.70258	0.69765	0.70752
1000	0.64851	0.64691	0.64950

K Value	**Throughput**	**Throughput**	**Throughput**

	N=1000, B=50	**N=5000, B=500**	**N=1000, B=500**
0	0.13899	0.371710	0.15914
1	0.66933	0.67429	0.69604
2	0.82798	0.66165	0.83678
10	0.91142	0.71596	0.91989
40	0.89098	0.87722	0.92696
100	0.85256	0.86376	0.87816
400	0.75839	0.75999	0.76190
1000	0.66012	0.66068	0.66116

The initial table shows the K-value and the throughput along with its CI lower and CI upper.

In order to stay within bounds of page size the next table just shows the appropriate information, this being the throughput for N1000B50, N5000B500, B1000B500, N5000B50

Discussion:

Thought it'll be a nice idea to go with the idea of playing around with the size of the frame. A couple experiments were done, we took the approach of drastically increasing or decreasing the frame size. The obvious observation from this approach is that if you do not increase or decrease the frame size in an appropriate manner to amount of time the simulation runs then the throughput changes drastically as well. For example, user decreases the frame size you will have less blocks produced and therefore less check bits to be added, therefore reducing the amount of bits that can be corrupted. In a more well rounded simulation we would also be able to take into consideration the network errors and their approach to a larger or smaller frame size.

Also thought about manipulation of feedback time size (A). We were able to observe that increasing the amount of feedback time will reduce the frames transmitted. While reducing the feedback time it would allow more frame transmissions to exist becomes less of a penalty when time is being assessed. If message is on a high priority network then we are faced with a feedback time that should be low so users can know whether important messages has been received properly. Due to these reasons testing the extreme feedback time helps model the actual real world environment.

If future updates are to be implemented in this simulator I'll recommend actually having to encode with the HSBC and sending frames over an actual physical network where natural bit errors are bound to occur rather than simulating ones. Also allowing further research using real world networks with error detection and correction using HSBC would be more suited with extra dimension so elaboration on the actual working model is more possible.