Technische Universität München, Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. W. Kellerer, Andreas Blenk, Mu He

Analysis, Modeling and Simulation of Communication Networks
SS 2016

Programming Assignment 1

Python-based Discrete Event Simulator

General Outline

In this programming assignment, you will implement a Discrete Event Simulator (DES) of a $GI/GI/1/S/\infty$ queuing system. The simulator should be programmed in Python in an object-oriented fashion for the sake of simple extensibility and re-usability.

We provide you with Python files that should be used throughout the exercises. As a first step, you will have to create the basic functionality of your simulator. After that, you will have to extend it step by step to make simulations and analyze the results. We will always specify where you have to implement parts of the simulator via the following indicators in each file, e.g.,

```
# TODO Task 1.1.1: Your code goes here
```

Note that you can remove the *pass* after having finished the implementation instruction. Check the python documentation for flow control, if needed (`https://docs.python.org/2/tutorial/controlflow.html`).

In general, you should be familiar with basic queuing models that are used to simulate parts of a communication network. If needed, please check the provided literature as introduce in the lecture. If you have any questions regarding Python, please check the online reference for Python 2.7 first (`https://docs.python.org/2/`).

Tasks marked with an asterisk (*) are optional. If you do them, you can collect up to 55 bonus points. We strongly suggest when working on an exercise to read it first completely. For instance, the code verification parts might be helpful when you are working on a subtask.

Submission Guidelines

For each part or section you should hand in a zip file with the filename *DES_partX_#YOUR_MATRICULATION_NUMBER.zip*. This file should contain a folder with your code and the iPython notebook.

- *DES_partX_#YOUR_MATRICULATION_NUMBER*: folder with your source code
- *DES_partX_#YOUR_MATRICULATION_NUMBER.html*: an iPython notebook, saved/ exported as html file with all simulation study and verification results, containing the python console output and all plots as well as the answers to the "Analysis and General Question" sections.

For instance, the submission files for the first part of the exercise could look like this:

You hand in a zip-file *DES_part1_01234567.zip* containing the following folder (source code) and notebook:

- *DES_part1_01234567*
- *DES_part1_01234567.html*

For creating the IPython notebooks, we recommend **Jupyter Notebook** (`http://jupyter.org/`).

1 Implementing the Discrete Event Simulator (110 Points)

In the first part of the exercise, you will create a Discrete Event Simulator step by step. This section consists of four parts:

- Modeling Server and Queue
- Implementing the Event Chain
- Processing the Events
- Creating the Simulator

After that, you will verify your system and answer questions on your implementation.

1.1 Modeling Server and Queue

In this part, you will implement the server and the queue. The queue (buffer) stores customers (network packets or shortly packets) that can not be served directly, thus, have to wait for other customers (packets) to be served first. This means the service unit has to finish the operation on the customer (packet) first. To keep your first implementation simple, you will not have to work on a detailed model for network packets in this assignment.

In general, for your implementation, you need an indication (flag) showing whether the server is busy and a counter showing how many customers are currently waiting in the queue. The queue size can be set, thus, be limited by the simulation parameter *S*. Now, implement the following tasks in the *SystemState* class in the file (Python module) named *systemstate.py*.

Task 1.1.1: Server modeling

5 Points

Create a new boolean variable *server_busy* that indicates whether the server is busy or not. Create a second variable *buffer_content* that indicates the status of the queue (buffer), i.e., how many customers (packets) it contains. Initialize both variables using plausible values in the function *init()*.

Task 1.1.2: Adding Packets to the System

5 Points

Modify the functions *add_packet_to_server()* and *add_packet_to_queue()*. The first function should return *true* if an arriving customer (packet) can be served directly without waiting. The second function should return *true* if the arriving customer (packet) cannot be served directly and has to be added to the queue. Modify the values of the variables that you have generated in the previous task accordingly. Note that you can access (read the value of) the parameter *S* through the member variable *sim*.

Task 1.1.3: Service Completion

5 Points

Modify the remaining two methods *complete_service()* and *start_service()*. The function *complete_service()* is called when the packet is processed completely by the server, i.e., the service is completed. The function *start_service()* returns *true* if the queue is not empty and another packet is taken from the queue and served directly by the server. Modify the variables from task 1.1.1, if needed, accordingly.

1.2 Event Chain

Your DES is based on an event chain that stores all different *SimEvents*. Each *SimEvent* has a time stamp showing the time when it is scheduled, and a priority indication to make sure that when multiple events occur at the same time, they will be processed in the correct order.

In this part, you will implement the functionality of the event chain, that is, inserting and removing events in the right and correct manner. For this, you have to modify two classes that you both find in the file *event.py*: *SimEvent* and *EventChain*.

Task 1.2.1: Comparing two events

10 Points

In order to correctly insert a *SimEvent* into the event chain, the events have to be compared with each other. We use the structure *heapq* for the event chain (`https://docs.python.org/2/library/heapq.html`). Whenever an event is inserted in our event chain, they are sorted by their time and priority. The earliest events are always processed first. If two events happen at the same time, the one with higher priority, i.e., lower priority value, is processed firstly. Implement the function *__lt__()* according to the above specification. Have a look at the python documentation, how it is used. Hint: this is called operator overloading.

Task 1.2.2: Inserting and removing events

10 Points

Modify the functions *insert()* and *remove_oldest_event()* in the module *EventChain* according to the above functionality descriptions of the event chain. The variable *event_list* should be of type *heapq* and would be modified, whenever one of these two functions is called.

1.3 Processing of Events

In the given DES there are three types of events: Customer Arrival, Service Completion and Simulation Termination. In this part you should implement the processing of each event. The following tasks should be implemented in the module *SimEvent* in the file named *event.py*. In each class inheriting from the the abstract class SimEvent, you should implement the process method.

Task 1.3.1: Simulation Termination

5 Points

This event is only processed at the end of a simulation run. Adapt the *process()* function to make sure that the simulation stops when this event occurs. The function should modify the parameters in class *SimState* for this.

Task 1.3.2: Customer Arrival

20 Points

This event should occur once a new customer (packet) arrives in your system. Think about what should happen when a customer arrives and implement the corresponding function. In detail, your function should either serve, enqueue or drop the customer (packet), depending on the current system state. For statistics, do not forget to call the functions *packet_accepted()* and *packet_dropped()* in module *SimState* whenever it is necessary. These functions count how many packets are dropped and how many packets have come into our system. Every packet should be counted in the correct way. Make sure that each CursomerArrival event inserts a new CusomerArrival into the event chain and a Service Completion event, if necessary. The inter-arrival-time between two CustomerArrival events is fixed and the value is stored in *SimParam*. The serving time should be a uniformly distributed random integer between 1 and 1000. Use the function *random.randint()* for that and set your random seed in *SimParam* to your matriculation number (`https://docs.python.org/2/library/random.html`).

Task 1.3.3: ServiceCompletion

20 Points

Whenever a packet has been served completely and leaves your system, this event will be called. Think about how its process function should behave when there are still packets in the queue (buffer) of your system and when the queue (buffer) is empty. Note that you may have to insert new events into the event chain in the processing as well. Note again that the serving time should be random as described in the previous task.

1.4 Creating the Simulator

Now you are ready to setup the whole simulator and make a simulation run. For this, the function *do_simulation()* in the python file *simulation.py* has to be modified. You see that the first and the last event of your simulation have already been inserted. All other events are inserted on the fly according to the implementation. The basic structure contains a while loop that runs until the *simstate.stop* flag is set. In each iteration of the while loop, only one event is processed and the simulation time (*simstate.now*) is updated accordingly.

Task 1.4.1: Creating the Simulator

10 Points

Modify the above-mentioned function *do_simulation()* such that the simulator processes the events in the right and correct manner. Pay attention here and think carefully about how and when to update the simulation time, since this can strongly affect the way your simulation works.

1.5 Verification

Since you have setup the simulator, it's time to verify your simulation results. We provide you a python unittest file that checks the basic functionality of your code. The unittests in file *part1_tests* check the functionalities of different SimEvents, EventChain and SystemState. Furthermore, it can run whole simulations with the seed ranging from 0 to 5, helping you to check the correct implementation of your DES.

When you run a simulation with a maximum queue size of 4 ($S = 4$) for 100 seconds, the system should count roughly between 5 and 20 dropped packets. Try different seeds to check whether this is the case in your simulation. Run all unittests and make sure they do not return any errors.

1.6 Analysis and General Questions

The last section of the first part of the programming assignment contains a few questions that should be answered separately. Explain your answers and write full sentences.

Task 1.6.1: Confidence

5 Points

In the last section you have run your simulation and received a result. How could you establish confidence in this case?

Task 1.6.2*: Event Chain Structure

5 Bonus

Explain why it makes sense to use a heap as your data structure for the event chain.

Task 1.6.3*: Update Event Chain

5 Bonus

Explain why it makes sense to update the event chain during simulation instead of inserting all events at the beginning of simulation. Consider both types of events, Customer Arrivals and Service Completions.

1.7 Simulation Study I

In the previous tasks, you have created a basic version of a DES. Now, perform a little simulation study. Use the given functions in the files *part1_simstudy.py* and write your code in these files. Explain your answers and write full sentences.

Task 1.7.1: Queue Length Determination I

10 Points

Find out which queue length is required such that after 100 s less than 10 packets are lost in at least 80% of 1000 simulation runs. Describe your idea how to design the simulation runs to achieve this goal. Think about cases, when you are close to fulfilling or not fulfilling the requirements and how you could assure a correct result in these cases.

Task 1.7.2: Queue Length Determination II

5 Points

Does the queue size depend on the simulation time? What happens if you simulate for 1000 s and want to have less than 100 dropped packets in 80% of the cases? Describe your observations!

Task 1.7.3*: Comparison of Results

10 Bonus

Compare the results of task 1.7.1 and task 1.7.2. Does the system behave differently or not? Explain your observations! Hint: Think about the distribution of the blocking probability per run. Possible solution: Plotting cumulative distribution functions of the blocking probabilities might be helpful.

2 Extending the DES (105 Points)

In this part, you will extend your simulator, so that you can do some measurements of the system in the simulation.

2.1 Packets

Until now, the queue has been a simple counter. In order to do measurements on the delay of packets, it is necessary to extend our previous implementation and store actual packets in our system. The module *Packet* in file *packet.py* represents a packet. A packet object has eight attributes. An attribute for its arrival time, an attribute for the time when the serving of the packet starts, an attribute for the time when the serving of the packet completes, a flag indicating whether the packet has to wait, a flag indicating whether the packet is currently being served, and a flag indication whether the packet processing is completed. Besides, it contains a reference to the current simulation and the inter-arrival time to the last packet.

Task 2.1.1: Variables of a Packet

10 Points

Implement the given functions. They should modify the variables of a packet, if needed. Think about which function has to modify which variable. Note, that the system time of a packet is defined as the whole time that the packet stays in our system, i.e., the summation of its waiting time and serving time.

2.2 Finite Queue

Now, modify your system, that you have a queue which actually contains packet objects. For this, you have to implement and integrate the class *FiniteQueue* in file *finitequeue.py*. The *FiniteQueue* class is part of your queuing system. An object of the *FiniteQueue* class stores all the queued packets. The finite queue class provides methods for adding packets, removing packets, getting the current queue length, indicating whether the queue is empty, and flushing the queue, i.e., deleting all packets inside. Use Python Queue data structure for the implementation of the *FiniteQueue*.

Task 2.2.1: Implementation of a finite queue

10 Points

Implement the missing methods in the *finitequeue.py* file. Access the capacity of the queue by referencing the parameter of your simulation.

Task 2.2.2: Integration of the finite queue class

10 Points

Modify your module *SystemState*, such that the queue of your system is no longer a simple counter, but a finite queue that contains packets. Adapt your module, so it now contains a variable called *buffer* of type *FiniteQueue* instead of the counter variable. Furthermore, create a variable *served_packet*, that represents the currently being served packet. Adapt all methods that interact with the queue and modify them in the correct way. The functions *add_packet_to_server()* and *add_packet_to_queue()* should create a new packet instance once they are called. Think, which other methods have to modify variables of a packet, e.g., call functions of a packet like *complete_service*. There are no indications where to implement this method, since you have to modify almost all methods in class *SystemState*.

You should also add a new variable into the class *SystemState*: the new variable, called *last_arrival*, should indicate the last packet arrival before the current packet, so that you can calculate the inter-arrival time. You can initialize it with the value 0.

Furthermore, add a function *get_queue_length()* to class *SystemState* that returns the current size of the queue, same as the equally named function in class *FiniteQueue*.

2.3 Counter

The counter module (file *counter.py*) contains an abstract class *Counter*, a class *TimeIndependentCounter* and a class *TimeDependentCounter*. *TimeIndependentCounter* and *TimeDependentCounter* inherit from *Counter*. Both inheriting classes should implement the methods as specified by *Counter*. Use methods provided by *numpy* package if possible to easily get mean, variance, and standard deviation of a set of sample data.

Task 2.3.1: Implementation of TIC

5 Points

Implement the class *TimeIndependentCounter* with all its methods, except the two methods that are related to confidence intervals. Think carefully about the variance calculation, assuming you only have a limited number of samples.

Task 2.3.2: Implementation of TDC

10 Points

Implement the class *TimeDependentCounter* with all its methods. Note, that you can't use standard *numpy* methods for statistics calculation here any more. Hint: For the variance calculation, use the formula of raw moments.

2.4 Histogram

In order to visualize data, it makes sense to plot histograms. Histograms contain a given number of bins, which depends on the input data and the number of samples. We have provided a class *Histogram* in the file *histogram.py*. The structure of the classes in this file is similar to the counter implementation, having an abstract class *Histogram* and time independent and time dependent realizations. Adding values to an instance of the class *Histogram* is done similarly as adding values to an instance of the class *Counter*. Histograms can be plotted with the given function *plot()*. Note, that the *plot()* function is part of the abstract module *Histogram* and is called at the end of the *report()* function.

Task 2.4.1: Creating a Histogram of Time Independent Values (TIH)

10 Points

For the TIH, you have to modify the methods *count()* and *report()*. The method *count()* should add values to the internal array. Some parts of the function *report()* have been implemented already. Your task is to generate the variables: *self.histogram* and *self.bins*. You can use the given function *numpy.histogram()* and modify some of the input parameters. Refer *numpy* documentation to see how to use it.

Task 2.4.2: Creating a Histogram of Time Dependent Values (TDH)

5 Points

For the TDH, you have to modify the functions *count()* and *reset()*. Note, that now each value should be weighted by its duration. For this, you have to add values to the array *weights* as well.

Task 2.4.3: Integration of Counters and Histograms in Simulator

5 Points

The class *CounterCollection* in the file *countercollection.py* contains some predefined counters and histograms. It provides the methods *count_packet()* and *count_queue()*. The first method should be called whenever the service of a packet is finished, e.g., in the function *complete_service()* of class *SystemState*. For an easy implementation, the second method *count_queue()* can be called after each time update step, i.e., in *do_simulation()* in the class *Simulation*.

Task 2.4.4*: Creating a Side-by-Side Plot

10 Bonus

Side-by-Side plot is useful in comparing results of different input parameters. Create one figure for the bar plots side-by-side. For this, you have to modify the function *plot()* in class *Histogram* and modify the side-by-side section. For plotting the bars side by side please refer to matplotlib examples, e.g., `http://matplotlib.org/examples/api/barchart_demo.html`.

2.5 System Utilization

Your simulator should be able to measure the system utilization. Add all necessary lines to the function *count_queue()* in class *CounterCollection*.

Task 2.5.1: Measuring the System Utilization

5 Points

Think about how to measure the system utilization with the given time dependent counter *cnt_sys_util* in class *CounterCollection* and implement it.

2.6 Verification

Again, we provide you a python unittest, which checks the basic functionality of your code. The unittests in file *part2_tests* check the functionality of the different extended SystemState, FiniteQueue, Packet and the Counters. Furthermore, it can run whole simulations with the seed ranging from 0 to 5, to give you the ability to check the correct implementation of your extended DES.

When you run a simulation with a maximum queue size of 4 for 100 seconds, the system should count roughly between 5 and 20 dropped packets. Try different seeds to check, if this is the case in your simulation. Run all unittests and make sure, they do not return any errors.

2.7 Simulation Study II

Now that you have implemented your extended DES, it's time to perform some analysis. Do the coding in file *part2_simstudy.py*.

Task 2.7.1: Systems with Different Queue Capacity I

15 Points

Create plots of the mean waiting times of a packet and the mean queue length (buffer fill status). Perform this simulation for queue capacity *S = 5, 6, 7* and simulation time of 100s. Run your simulation 1000 times add the averages to respective histograms. Plot your histograms. You should have three histograms for the waiting times and three for the queue lengths. For easier comparison, you can also display the results for the different queue capacities in one plot only (for the mean queue length, you can use the side-by-side plot for better comparison). Think of a reasonable number of bins. Hint: You can and should use additional configuration parameters defined in class *SimParam*, like *S_VALUES*. For better performance you can also temporarily comment all unused counters in class *CounterCollection*.

Task 2.7.2: Systems with Different Queue Capacity II

5 Points

Create the same plots for a simulation time of 1000s. Still, simulation has to run 1000 times.

2.8 Analysis and General Questions

Answer the following questions separately and in full sentences. Explain your answers.

Task 2.8.1: Systems with Different Queue Capacity

10 Points

Describe the observations, that are highlighted by the plots from tasks 2.7.1 and 2.7.2. What can you observe, when you increase the queue capacity? What are the differences between a simulation time of 100s and 1000s, especially regarding the variance? Why does this effect happen?

Task 2.8.2: Choosing the right number of bins

5 Points

Explain the numbers of bins you choose for the histograms. Write down all your considerations.

Task 2.8.3*: Variance Calculation

10 Bonus

In task 2.3.1 you have calculated the variance. What is the difference between the variance calculated in this manner and the variance of infinite number of samples? Where does the difference come from?

3 Generating Random Numbers (40 Points)

Until now, you have been using a uniformly distributed serving time and a constant inter-arrival rate. Now, you add the possibility to draw numbers from specified distributions.

3.1 Implementation

You will implement a random number generator (RNG) for exponential and uniform distributions. A series of random numbers is called random number stream (RNS). The RNG class contains two RNS, one for the inter-arrival time and one for the service time.

Task 3.1.1: Random Number Generation

5 Points

Implement all methods in the subclasses of the abstract class *RNS*. The subclasses *ExponentialRNS* and *UniformRNS* should be initialized with a seed and all necessary parameters. The method *next()* should return a new sample of a given distribution. You can use the inverse transform method to generate new samples of a distribution. The seed input parameter should be optional.

Task 3.1.2: Integration of the RNG

5 Points

Now, integrate your RNG into the simulation. For this, you have to modify several methods.

- First, modify the method *do_simulation()* in your class *Simulation*. Uncomment the import statements and the generation of an RNG. You should modify the generation in order to receive an RNG with two exponential random number streams. The mean of the inter-arrival time distribution should be 1s, the mean of the service time distibution should be depending on the value *Simparam.RHO*. Note, that you have to reset the RNG when you change the value of ρ!
- After that, modify all lines in the *process()* functions in the classes *CustomerArrival* and *ServiceCompletion*. You should modify at least three lines and call the dedicated RNS instead of a normal random number generation or the constant inter-arrival time.

3.2 Verification

Write your code for the following tasks in file *part3_simstudy.py*. For checking the correct implementation of the system utilization and the usage of rho, you can run the test in *part3_tests.py* first.

Task 3.2.1: Verification of distributions

5 Points

Create two histograms, one for each implemented distributions. Generate a sufficient number of samples by using the classes RNG and/or RNS and choose a reasonable number of bins for your histograms.

Task 3.2.2: Verification of system utilization

10 Points

Since you are now able to generate exponential distributions, verify your implementation of the system utilization by simulating an M/M/1/S system with a limited queue capacity (S = 5) and $\rho = \lambda/\mu = 0.01, 0.5, 0.8, 0.9$. Simulate your system for 100s and 1000s. Note, that depending on your implementation you may have to reset your simulation object when changing the ρ in the parameter object.

3.3 Analysis and General Questions

Answer the following questions separately and in full sentences. Explain your answers.

Task 3.3.1: Seed for Random Number Generation

5 Points

Why does it make sense to use a seed when simulating a non-deterministic system?

Task 3.3.2: Bins for Histograms

5 Points

How does the shape of the histograms change if you reduce/increase the number of bins? What would be a reasonable number of bins and how does it depend on the number of samples?

Task 3.3.3: System Utilization

5 Points

How do you interpret the results from task 3.2.2? What is the expected system utilization, if you simulate for an infinite simulation time? What happens, if S becomes unlimited?

4 Correlation (65 Points)

From now on, your system should always be an M/M/1/S system, i.e., the inter-arrival time (IAT) and the service time should be exponentially distributed.

When analyzing simulation results, it is important to check for correlations. In this part, you will implement a version of the TIC, that is able to calculate the correlation coefficient. If you correlate two different arrays or counters, you speak of cross correlation. If you correlate an array with a shifted version of itself, you speak of auto correlation.

4.1 Implementation

You have to implement all functions in the classes *AutoCorrelationCounter* and *CrossCorrelationCounter*, both in file *counter.py*.

Task 4.1.1: CrossCorrelationCounter

10 Points

Implement all missing methods in this class. Note, that this counter should count two values at once, one value for every array. The method *get_cor()* calculates the cross correlation between the two internal arrays. Use the method *get_cov()* (get covariance) for your implementation of the correlation.

Task 4.1.2: AutoCorrelationCounter

15 Points

Implement all missing methods in this class. Note the same things like in the previous exercise. In this case, the correlation coefficient is dependent on the lag between the one internal array.

4.2 Verification

Do the coding for sections 4.2 and 4.3 in the file *part4_simstudy.py*.

Task 4.2.1: Auto Correlation

10 Points

Verify the correctness of your auto correlation counter by autocorrelating the following sequences:

- 1, -1, 1, -1, 1, -1, 1, -1,...
- 1, 1, -1, 1, 1, -1, 1, 1, -1,...

Hint: Think of a reasonable lag size. Choosing one lag size is not enough for a good verification.

4.3 Simulation Study III

Now, test your code for correctness with the provided test class *part4_tests.py*. Afterwards, have a look at the class *CounterCollection*. You see some correlation counters here, that count different correlations between arrays. The following correlation counters have already been implemented.

- Correlation between IAT and waiting time of a packet
- Correlation between IAT and serving time of a packet
- Correlation between IAT and system time (waiting time + serving time) of a packet
- Correlation between service time and system time of a packet
- Auto-correlation of waiting time with lags ranging from 1 to 20

Task 4.3.1: Correlations in the DES

10 Points

Calculate the above correlations and auto correlations for a given system utilization of $\rho = 0.01, 0.5, 0.8, 0.95$. The queue size should be infinite (you can set it to 10000).

Task 4.3.2*: Auto Correlation of Waiting Times

10 Bonus

Prepare two plots for the autocorrelation for lags ranging from 1 to 20. Prepare one plot for N = 100 and one for N = 10000 simulation runs, where N is the number of served packets. The queue size should be unlimited. Hint: For this, you have to implement the function *do_simulation_n_limit()* in class *Simulation*, such that the simulation stops after a total packet count of N. However, you can take most of the code from your regular simulation routine.

4.4 Analysis and General Questions

Answer the following questions separately and in full sentences. Explain your answers.

Task 4.4.1: Lag size

10 Points

What are reasonable lag sizes for the test sequences in task 4.2.1? What does the correlation coefficient say? Consider especially the extreme values!

Task 4.4.2: Correlation Times in the DES

10 Points

What conclusions can you draw from the correlation coefficients? Interpret all results of task 4.3.1. What can you say about whether packets have to wait or not? Explain!

Task 4.4.3*: Auto Correlation

5 Bonus

If you have done task 4.3.2: What is the problem with N = 100?

5 Confidence (75 Points)

In the lecture, the concept of confidence has been introduced. In this section, you will implement the calculation of confidence intervals and do some simulations to get to know this concept.

5.1 Implementation

Your TIC should be able to create a confidence interval based on the samples in its internal array and a confidence level alpha, given as an input parameter.

Task 5.1.1: Calculating Confidence Intervals

10 Points

Implement the two methods *report_confidence_interval()* and *is_in_confidence_interval()*. The first method should return the half width of the symmetric confidence interval. The second method should calculate, whether a given sample x is in the confidence interval. Use the first function in the implementation of the second one.

You can test the basic functionality of the confidence intervals by running the test in *part5_tests.py*.

5.2 Simulation Study IV

In the following, you will implement your simulation in a way that it can be interrupted based on the width of the confidence interval. The width of the confidence interval should be two times $\varepsilon = 0.0015$.

You will perform the simulation in two ways: The first way is to make multiple runs with a fixed time and to stop your simulation after a specific amount of runs when a given width of the confidence interval is reached. The second way is to make one run and take batches of a fixed time. You calculate your confidence interval based on the values you get from each batch. Once the width is reached, you stop your simulation.

Do the coding for this section in file *part5_simstudy.py*.

Task 5.2.1: Multiple Runs Confidence

10 Points

Implement simulation routine, that runs fixed time simulations until the width of the confidence interval is small enough. For this, call your simulation like in the previous tasks with a simulation time of 100 s and 1000 s repectively. After each run, extract the blocking probability and add it to a new TIC. Calculate the width of the confidence interval and stop your simulations when it is below two times ε. Use $\rho = 0.9$ and S=4 and calculate the number of runs for a confidence level of 0.9 and 0.95.

Task 5.2.2: Batch Confidence I

15 Points

Now there should be no event *SimulationTermination* and the simulation keeps running. For each N = 100 (N = 1000) packets, the blocking probability is read and is added to a new TIC. Then the statistics (*simresult.reset()*) should be reset. Stop the simulation when the width of the confidence interval is below two times ε. Calculate the total simulation time for a confidence level of 0.9 and 0.95 ($\rho = 0.9$, S=4). Hint: If you haven't done it already in bonus task 4.3.2, you have to implement the function *do_simulation_n_limit()* in class *Simulation*, such that the simulation stops after a total packet count of N. However, you can take most of the code from your regular simulation routine.

Task 5.2.3: Batch Confidence II

5 Points

Implement a plotting function that takes confidence intervals as an input. On the x-axis you should have one id per sample (confidence interval). On the y-axis, you should plot the mean value with the corresponding confidence interval.

Task 5.2.4: Confidence Plots

10 Points

Make the following simulation study. Use an $M/M/1/\infty$ system. Calculate the confidence interval for the system utilization of 30 runs. Repeat this 100 times. Take each confidence interval and plot it with your implemented function (see task 5.2.3). On the x-axis you should have values from 1 to 100 (for every repetition) and on the y-axis the given calculated mean value with the error bars. Also plot the theoretical value as a dashed line. Make your study for system utilizations 0.5 and 0.9. Use confidence levels of 0.9 and 0.95. Use 100 s and 1000 s as simulation time.

5.3 Analysis and General Questions

Answer the following questions separately and in full sentences. Explain your answers.

Task 5.3.1: Confidence Interval Width

5 Points

How many runs do you need in each setup of tasks 5.2.1 and 5.2.2? Compare all eight values and explain, why they differ and which are the advantages of the setups.

Task 5.3.2: Confidence Interval Width

5 Points

Compared to the actual blocking probability calculated by an analytic formula

$$P(S) = \frac{(1-\rho)\rho^{S+1}}{1-\rho^{S+2}} \tag{1}$$

the blocking probability in task 5.2.1 always differs. Why is this the case and does it differ in task 5.2.2 as well?

Task 5.3.3: Confidence Interval Width

10 Points

What can you observe in the plots of task 5.2.3? What are the differences between the plots How many intervals are covering the true value and how many not? How does the skewness affect the calculation of confidence intervals?

Task 5.3.4: Variable Simulation Time

5 Points

Would it also be an option to do a simulation with an infinite simulation time (and stop as soon as a given confidence level is reached) and take the samples for the blocking probability each time a packet is dropped? Justify your answer!

6 Statistic Tests (25 Points)

The last part of the programming assignment deals with statistic tests. As an example, you will implement the chi square test to check, if a set of samples fits an estimated distribution.

6.1 Implementation

The basic formulas of the chi square test have been presented in the lecture. You should implement them in the class *ChiSquare* in the file *statistictests.py*.

Task 6.1.1: Implementation of ChiSquare class

15 Points

Implement the functions *init()* and *test_distribution()*. The first function should take two input parameters:

- An array *emp_x* with k+1 values which represent the borders of the bins.
- An array *emp_n* with k values which represent the frequencies of each bin.

The second function should take the significance level alpha, mean value and standard deviation of a normal distribution as an input and should test the empirical values against the distribution and report, if the zero-hypothesis is rejected or not. Hint: Use the formulas provided in the lecture.

In addition, as is shown in the lecture slides, the minimum expected frequency of each interval is 5. If this is not the case, the combination of neighboring intervals should be done accordingly. You should consider the simplest case, in which only 100 samples are provided. Since the expected distribution is a normal distribution, only the intervals on both tails of the distribution should be combined as 2 compound intervals.

6.2 Verification

You should now verify your implementation of the test. You can simply do this by drawing samples from a normal distribution with given mean and variance. Then, you run your test on an distribution with the same parameters. Depending on the number of bins, the hypothesis will not be rejected in most cases. Do the coding for this section in file *part6_simstudy.py*.

In addition, you can run the given test *part6_tests.py* for verification. Note that for the test to work correctly, your *test_distribution()* must return the result: $[chi2, chi2\_table]$, where chi2 represents your calculated chi square value and the second variable represents the value drawn from the table (for given alpha and d.o.f.).

Task 6.2.1: Chi Square Test

5 Points

Draw 100 random samples from a standard normal distribution. Choose a reasonable number of bins and run the chi square test on your samples. The null hypothesis should be: the samples follow the standard normal distribution.

6.3 Analysis and General Questions

Answer the following questions separately and in full sentences. Explain your answers.

Task 6.3.1: Chi Square Test

5 Points

What can you observe, when you vary the number of bins or the significance level alpha in task 6.2.1? What happens, if you change the mean and/or the variance of the samples but don't still compare them to the standard normal distribution in the test? Describe your observations!

Total: 420 Points + 55 **Bonus**