Project A - Semester 2, 2026

Table of contents

1 Discrete Event Simulation of a Modulated Queueing Network

(last edit: August 30, 2026)

Due: Friday, 25 September, 2026, 16:00. Late submissions are not accepted. **Work in pairs** - you must have a single partner for submission. Any deviation from this requires special permission from the course coordinator.

Note: The teaching staff will only answer questions (via Ed, consultation hour, or practicals) regarding this assignment up to the late evening of Wednesday September 23.

Weights and marking criteria: Total number of points: **100**. * There are **10** points for handing in according to the hand-in instructions, including a voice recording, neat output, and most importantly the GitHub repo. * The remaining **90** points are for the project itself. * In general, points **will be** deducted for sloppy coding style. Make sure to have your code properly indented, to use sensible and consistent variable names, and to write code that is in general clean and consistent.

Submission format: This assignment should be submitted via a GitHub Repo and a PDF file via Blackboard.

Specific instructions for the GitHub repo are below. It is important that the GitHub repo be created within (owned) by the github organizatio MATH25042026 and be made **private**. It is also important to name the repo *exactly* as `PROJECTA-2504-2026-<<FIRST NAME1>>-<<LAST NAME1>>__<<FIRST NAME2>>-<<LAST NAME2>>`. So for example if your names are "Little Pug" and "Big Hound" the repo name should be `PROJECTA-2504-2026-Little-Pug__Big-Hound`. Please create the repo as early as possible.

The PDF file should be a nicely formatted file that includes:

- Your names, student numbers, and assignment title (Project A - 2026) on the top.
- A (clickable) link to your GitHub repo.

- A brief description of how you have structured your module and which files contain the important functions.
- Instructions on how to run the main simulation code in the repo via an easy to use `simulation_script.jl` file.
- Output and answers to specific questions formatted in an organized manner.
- Plots and graphs should have clean organized labels, legends, etc.

The recommended way to make the PDF file is via a Jupyter notebook where you copy in some code and output into the notebook, and in certain cases use `include()` to run Julia code if appropriate. You also comment on questions in this PDF (e.g. when asked to answer things not via code). If desired you could keep this jupyter notebook (`.ipynb` file) in the repo. However, this Jupyter notebook will not be checked (only the PDF file), and it isn't required to be a "runnable" notebook. In any case, please avoid pixelated screenshots of code, so for example if you choose to format your PDF file in MS-word (instead of Jupyter), make sure the code is cleanly formatted, and readable (this page may be useful).

Marking responses will be made by the teaching staff by annotating (or leaving textual comments) regarding **selected parts** of your PDF file via blackboard. The teaching staff will also inspect your GitHub repo and potentially run `simulation_script.jl`. A very readable and clean PDF file is important. In printing Jupyter to PDF (or exporting to PDF) there may sometimes be excessive white space. Do not worry about this extra white space; it is not a problem.

Work in pairs but discuss with others: This is work in pairs. Plagiarism between groups will not be accepted. Nevertheless, feel free to consult with friends or classmates via Ed and other means about how to go about certain tasks. Answering Ed questions of others is recommended.

1.1 Setting up your GitHub repo for hand-in (10pts)

- Ideally use the same accounts you used for BigHW. Create your repo within, i.e. owned by that MATH25042026 github organization. Choose one of team members to "own" the repo and the other(s) will be invited as collaborators.

- Make sure the repo is **private**.
- Do **not** make any changes (commits) to the repo after the project due date.
- Create a local clone of the repo. It is recommended to use the `git` command via the shell to work on making changes/additions to the assignment and submitting the changes. However you are free to use any other mechanism (VS-Code, GitHub desktop, etc).

Your GitHub repo should be formatted exactly as follows: * Have a `README.md` file which states your names, the assignment title, and has a (clickable) link to the assignment instructions on the course website (to this document). * Have a `LICENSE.md` file. Choose a license as you wish (for example the MIT license). However keep in mind that you must keep the submission private. * Have a `.gitignore` file. * Have basic running instructions on how to run the code.
* Have `src` and `test` folders. * Note: make sure that your submission repo contains only those files mentioned above (with the exception of perhaps a Jupyter `.ipynb` file if you choose to use it for creating the PDF). You may use the `.gitignore` file to ensure `git` does not commit additional files and output files to your `repo`. * All code (except for scripting) is to be part of a Julia module (with the module statement). Call the module `ModulatedJacksonSim`. (To avoid confusion, be aware that "modulated" for the network has nothing to do with "module" as in the module statement). * Make sure the project has a `Manifest.toml` and a `Project.toml` file that specify all the dependencies exactly. Do not have extra dependencies that are not used. * Have a `simulation_script.jl` file which runs all of the code needed for the tasks. As this script runs, make sure it prints some light intermediate output (e.g. progress), especially for simulations that take significant time to run.

Since this project involves work in pairs, it is recommended that you try to use GitHub effectively. You may work on parallel branches with frequent merges (but this is not required), or work on the same branch via

peer programming. The way you develop is not being assessed, still it is recommended you make good use of GitHub.

1.2 Overview of the model

You may want to listen to an earlier Explainer video by Yoni Nazarathy.

Your key task is to create a discrete event simulation engine for stochastic models that are called "Modulated Open Jackson Networks". Jackson networks are well studied in classical "applied probability" and "stochastic modelling". The phrase "modulated" means that the parameters of the network vary over time. The "Open" phrase implies that customers may enter and leave these networks (in contrast with "closed" networks that have a finite number of customers).

A Jackson network (or "open Jackson network") is a network of queues (also known as "nodes" or "stations") where customers (also known as "jobs") arrive from the outside world, are served at the queues in a "first-come first-served manner" and then move to other nodes or leave. As a very rough motivational example consider an amusement park. There are multiple rides, and individuals move throughout the park and queue up for rides. At any point (after finishing a ride) an individual may go home, or, alternatively, go to the next ride, and so forth. Below is a detailed mathematical description of the model.

There are L stations (e.g. $L = 5$ or $L = 100$) and each station has a **server** and a **waiting buffer** (or queue). There are jobs traversing through the network and the stations. If needed, jobs can be uniquely identified (numbered) via $\ell = 1, 2, 3, ...$ and it is expected that the simulation processes thousands, or millions of jobs, so that if you track individual jobs then ℓ is an increasing counter of the jobs.

A job's location can be one of:

- Out of the system prior to arrival (the job still doesn't exist).
- In the system in a buffer of one of the stations (queueing - i.e. waiting for its turn to enter service).
- In the system at one of the stations being served by the server at that station. This means it is "at the head" of the queue (and being served).
- Out of the system after leaving.

Warning: Your simulation should **not** have entities for individual jobs, but rather only have counts and general descriptors as the system state. The description above about the location of a job is merely for understanding the system being simulated. If you were interested in estimating other quantities such as the distribution of the time for a job in the simulation, then you would have to keep track of individual jobs.

Time is continuous, $t \geq 0$. In this continuous time system, jobs arrive to the system according to a random process and traverse between the stations as we describe now.

Each station $i = 1, ..., L$ has a service rate $\mu_i > 0$ which is the inverse of the mean expected time it takes the server to process a job. Each station has a queue which has infinite capacity - i.e. there is no limit on the number of jobs in the queue. Jobs are served one by one at each station, where the mean service duration of each job is μ_i^{-1} and that duration is statistically independent from all other services at that node and at other nodes.

When a job arrives to station i, if the server is free (and this means there are no jobs in the buffer) the job immediately starts to receive service from the server. If the server is busy, the job queues for service and waits for its turn. Jobs may arrive either after finishing service at another node, or directly from the outside world.

The rate of arrivals of jobs to nodes from the outside world is $\alpha_i \geq 0$ for node i. Hence for nodes i with external arrivals ($\alpha_i > 0$), the mean duration between external arrivals is $1/\alpha_i$. In this project we'll consider the durations of times between external arrivals to be exponentially distributed. This makes external arrival processes "Poisson processes" (this is also a mathematical detail that is not important for the carrying out the project).

As for the durations of service times we will set them also as exponentially distributed. This assumption of Poisson arrival processes and exponentially distributed service time is what makes the network an "Open

Jackson network". Mathematically it means the network can be described by a "continuous time Markov chain" (you do not need to know these details for this project). In such a case, some explicit results exist and you could use these to test some parts (this is in Task 2 and Task 3). However, the **modulation** component of the model, described below, implies that generally, the model is more complicated than an Open Jackson Network. In this case, explicit results are much more scarce and hence the simulation model may yield information that is not available otherwise.

When a job completes service it either leaves the system immediately, or immediately moves to another buffer. The choice of its destination is based on the routing matrix P. Here $P_{i,j}$ is probability of going to station j after completing service in station i. It is assumed that P has non-negative entries and has a spectral radius less than 1. Mathematically this means that the maximal absolute value of eigenvalues of P is less than 1. It also means that jobs may always eventually leave the system and that at least some rows of the matrix P sum up to less than 1. The probability of a job leaving the system (immediately) after service at station i is $1 - \sum_{j=1}^{L} P_{i,j}$.

Tip: Recall (4) from the question on ''Markov chains and LinearAlgebra'' of BigHW. The `sample()` function from `StatsBase.jl` may help generate the location (node) of the next job when carrying out the simulation. Yet keep in mind that "outside of the system" is also a possible location.

How does **modulation** work? With this model feature, the service rates of nodes are not fixed at $\mu_1, \ldots, \mu_L$ but rather alternate between the rates $\mu_1, \ldots, \mu_L$ and the rates $\kappa\mu_1, \ldots, \kappa\mu_L$, with $\kappa > 1$. I.e., the second set of rates is "faster" than the first set of rates. Hence the system is either in "slow mode" (rates denoted μ_i), or is "fast mode" (rates denoted $\kappa\mu_i$). At any given time, the mode of the system is either `slow` or `fast` and when this change happens, all rates instantly change. Very loosely, in the amusement park setting, you can think of `slow` as a situation where rides only work with some employees and equipment while `fast` is a situation where rides work with more employees and equipment and operate faster. Modulation is then the process of changing between `slow` and `fast`, back and forth.

The process of changing (modulating) between "slow mode" (`slow`) and "fast mode" (`fast`) occurs independently of the jobs in the system, and for all servers together. Specifically the system changes between `slow` and `fast` and back as follows: `slow` durations are exponentially distributed with mean γ_1^{-1} where $\gamma_1 > 0$. `fast` durations are exponentially distributed with mean γ_2^{-1} where $\gamma_2 > 0$. The system alternates between `slow` and `fast` and back based on such independent random variables. The system starts in `slow`, then changes to `fast`, then back to `slow`, and so fourth. We denote the long term proportion of time that the system is `slow` via $R^{(1)}$. This quantity is,

$$R^{(1)} = \frac{\gamma_2}{\gamma_1 + \gamma_2} = \frac{\gamma_1^{-1}}{\gamma_1^{-1} + \gamma_2^{-1}}.$$

Similarly the long term proportion of time that the system is in `fast`, $R^{(2)}$ is just $1 - R^{(1)}$ or,

$$R^{(2)} = \frac{\gamma_1}{\gamma_1 + \gamma_2} = \frac{\gamma_2^{-1}}{\gamma_1^{-1} + \gamma_2^{-1}}.$$

Note that if $R^{(1)} = 1$ or $R^{(2)} = 1$, it means that the system is either always `slow` or respectively always `fast`. This can be approximated for "always `slow`" by sending $\gamma_1 \to 0$ (infinitely small "changing to fast" rate). Similarly $\gamma_2 \to 0$ will achieve "always `fast`". We call these **test cases** the **always slow** and **always fast** test cases. They are important because in this case the network is essentially just an Open Jackson Network (not modulated).

When the system changes mode (`slow` to `fast` or the other way), any service time of a job currently in service is purged and the service restarts for that job at a new service duration according to the new mode. So for example assume that server number 7 has a job that entered service at time `9.9` when the system was `slow`. This means that at time `9.9` the service duration was drawn with rate μ_7. Say that this duration is `3.3`. Hence the job is due to leave service at time `13.2`. Now assume that at time `11.4` the system switches

to `fast` mode. Then at that point the job in service for server 7 needs to forget that it is due to leave at time `13.2` and instead a new service duration is drawn. That duration is exponentially distributed with rate $\kappa\mu_7$. Say the new service duration is `2.3`, the job is now due to finish service at `11.4+2.3=13.7`.

In a modelling setting we may expect to have the modulation occur at a rate much slower than the arrival and service rates. I.e. the rates α_i, μ_i, and $\kappa\mu_i$ should generally be much larger than the rates of modulation, γ_1 and γ_2. For example in the amusement park example we can believe that staffing levels change in the order of hours while arrivals and ride durations are in the orders of seconds and minutes.

While this may not be useful for modelling real scenarios, if we take $\gamma_1 \to \infty$ and $\gamma_2 \to \infty$ (very quick modulation), and under the exponential random variable assumptions, it will mean that the system acts like a non-modulated network (just an Open Jackson Network) with service rates,

$$\tilde{\mu}_i = R^{(1)}\mu_i + R^{(2)}\kappa\mu_i.$$

We call this $\gamma_1 \to \infty$ and $\gamma_2 \to \infty$ case the **super fast modulation** case. Note that to avoid that your simulations in this scenario stall, the **service rate should be high relative to the modulation rate!** It is an important test case because like the **always slow** and **always fast** test cases, it should also yield results as an Open Jackson Network (non modulated) with service rates $\tilde{\mu}$.

These averaged service rates, $\tilde{\mu}_i$, are also important in the stability condition we describe below. Note also below, that when we consider notation like μ, or $\tilde{\mu}$, or α, we refer to the L dimensional vector of the individual entries.

1.3 Overview of model parameters

The parameters of the network that we shall use are:

- The dimension of the network (number of nodes): L.
- The external arrival rates: $\alpha_1, \ldots, \alpha_L$, with $\alpha_i \geq 0$ for all i, and $\alpha_i > 0$ for at least one i. (Vector α).
- The slow (normal) service rates: $\mu_1, \ldots, \mu_L$, with $\mu_i > 0$ for all i. (Vector μ).
- The speedup parameter κ with $\kappa > 1$.
- The modulation rates. Slow to fast: γ_1. Fast to slow: γ_2.
- The $L \times L$ routing matrix P.

Note that with the introduction of the speedup parameter we have the averaged service rate vector as,

$$\tilde{\mu} = R^{(1)}\mu + R^{(2)}\kappa\mu = \frac{\gamma_1\kappa + \gamma_2}{\gamma_1 + \gamma_2}\mu.$$

Note also that we may want to use the pair $(\gamma_1, R^{(1)})$ as parameters instead of (γ_1, γ_2). That is, we would like to specify γ_1 which is the rate of modulation (`slow` to `fast`) and $R^{(1)}$ which is the proportion of time for `slow`. With this we just set

$$\gamma_2 = \gamma_1 \frac{R^{(1)}}{1 - R^{(1)}}.$$

Tip: You'll be generating a lot of exponentially distributed random variables. For example this is a distribution with $\mu = 2.5$ or a mean of 0.4.

```
using Distributions, Statistics

 = 2.5

dist = Exponential(1/ )

println("Theoretical:")
```

```julia
@show 1/mean(dist)            #this is the mean() method from Distributions (theoretical mean)

println("\nMonte Carlo:")
data = [rand(dist) for _ in 1:10^6]
@show 1/mean(data);            #This is mean() from Base/Distributions (the sample mean)
```

```
Theoretical:
1 / mean(dist) = 2.5

Monte Carlo:
1 / mean(data) = 2.504070517882621
```

1.4 Arrival rates to nodes and traffic equations

Jobs arrive to nodes either from outside or internally after being routed to a node. Combining both of these arrivals, a basic quantity one can measure is the rate of arrivals to a given node i. That is,

$$\lambda_i = \lim_{t\to\infty} \frac{A_i(t)}{t},$$

where $A_i(t)$ is the number of arrivals (either from the outside world or from other nodes) to node i during $[0, t]$. This quantity can be measured via simulation (as you will do), but it can also be computed directly via the matrix equation,

$$\lambda = P^\top \lambda + \alpha, \qquad \text{(Traffic equations)}$$

where λ is a (column) vector of λ_i values, and α and μ are also column vectors of the respective quantities. These "Traffic equations" hold as long as the solution to the equation $\lambda = (I - P^\top)^{-1}\alpha$ satisfies, $\lambda < \tilde{\mu}$. Here the inequality between vectors requires element-wise inequality on all elements. To understand the traffic equations further, look at them for each i:

$$\lambda_i = \underbrace{\alpha_i}_{\text{external}} + \underbrace{\sum_{j=1}^{L} P_{j,i}\lambda_j}_{\text{internal}}.$$

Intuition: Observe that λ_i is equated to the sum of external arrivals and all internal arrivals from other nodes j routed with proportions $P_{j,i}$. This holds as long as the input rates to all nodes j are less than the averaged (for modulation) service rate.

When $\lambda_i < \tilde{\mu}_i$ for all i, we say the **network is stable** and this means that the arrival rate to each node is less than the effective service rate. In this **stable** case, as time goes to infinity, the network converges to statistical equilibrium or "steady state". This means that if we consider a single long simulation, we may run a time average of that simulation to obtain steady state quantities.

We will denote the **load for node** i via $\rho_i = \lambda_i/\tilde{\mu}_i$ and always work in a regime where $\rho_i < 1$ for all $i = 1, \ldots, L$. We denote $\rho^* = \max\{\rho_1, \ldots, \rho_L\}$ as the "bottleneck load". Clearly, as long as $\rho^* < 1$ the network is stable. The computed parameter ρ^* is the parameter that we will vary for different scenarios and cases - details below.

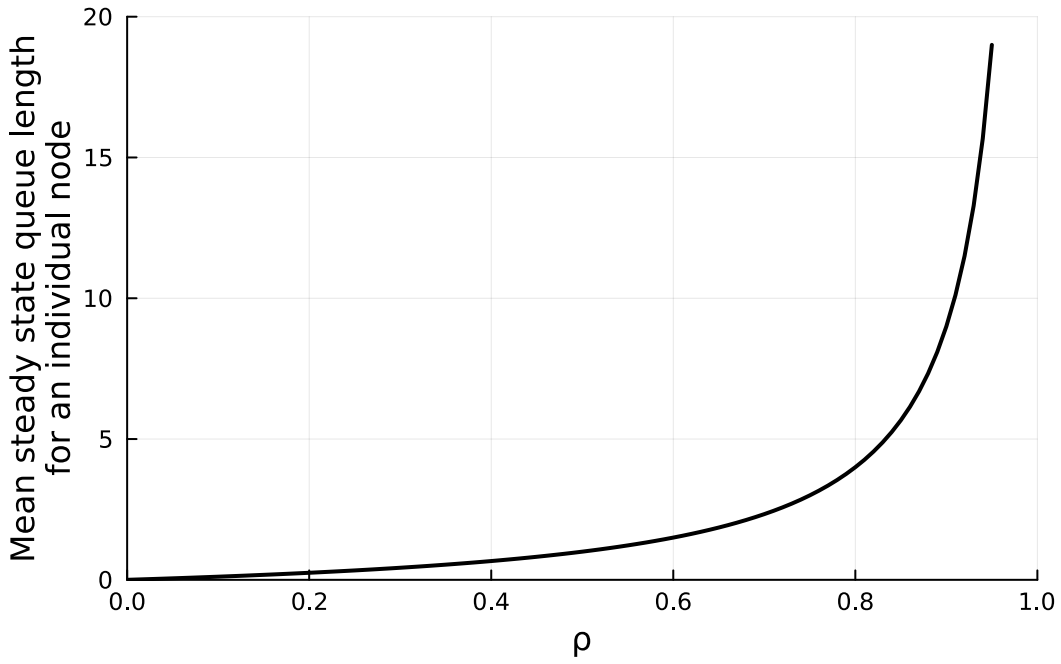
1.5 Known queueing theory results and an overview of questions to investigate

In general, it is known that as each ρ_i grows, the congestion in the individual queue at node i increases in the sense that "on average", there are larger steady state queues and longer wait times. Further, as ρ^* grows,

the general congestion in the network grows. Since we consider $\rho^* < 1$, we only focus on networks that are stable, yet if $\rho^* \approx 1$, e.g. $= 0.95$, they are highly congested.

For a (non-modulated) Open Jackson Network, one can prove that the steady state queue length of each individual queue i has a "geometric distribution" on $0, 1, 2, ...$ with a mean of $\rho_i/(1-\rho_i)$. Many more such theoretical results exist, yet we skip the details. This relationship appears as follows:

```
using Plots
 _grid =0:0.01:0.95
mean_steady_state_queue_size( ) =  /(1- ) #Theoretical steady state mean queue length (also of M/M/1
queue).
plot( _grid, mean_steady_state_queue_size.( _grid),
   xlabel = " ", ylabel = "Mean steady state queue length\nfor an individual node",
   label = false, lw = 2, c = :black, xlim = (0,1),  ylim=(0,20))
```



However with modulation, there is not an explicit simple relationship. One of the main goals of the project is to quantify the effect of modulation.

Tip: In the tasks below you are not expected to look at a node per se but rather at the whole network. You would thus create a superposition of graphs as above, where you sum up all of the queues in the network. When considering the whole network, your x-axis would not be an individual ρ_i but rather ρ^*.

1.6 Example scenarios (networks)

You will work with the following 4 scenarios. These are networks of size 3 nodes, 3 nodes, 5 nodes, or 100 nodes.

```
#You need to install the Parameters.jl package: https://github.com/mauro3/Parameters.jl
#You need to install the Accessors.jl package: https://github.com/JuliaObjects/Accessors.jl
using Parameters
using Accessors
using LinearAlgebra
using Random

@with_kw struct NetworkParameters #The @with_kw macro comes from the Parameters.jl package and
makes nice constructors
```

```
   L::Int
    _vector::Vector{Float64} #This vector is a vector of  _i which can then be scaled
    _vector::Vector{Float64} #This is the vector of service rates considered fixed
   P::Matrix{Float64} #routing matrix
    ::Float64 = (10^-8) #The rate of modulation slow to fast (default is that system is almost
always slow)
    ::Float64 = 1.0 #The rate of modulation fast to slow
    ::Float64 = 2.0 #The speedup parameter
end

# Returns the averaged service rate vector (this is tilde )
averaged_service_rate(net::NetworkParameters) = (net. *net.  + net. )*net. _vector/(net.  + net. )

############################
# Three queues in tandem
scenario1 = NetworkParameters(  L=3,
    _vector = [0.5, 0, 0],
    _vector = ones(3),
   P = [0 1.0 0;
   0 0 1.0;
   0 0 0])

############################
# Three queues in tandem with option to return back to first queue
scenario2 = @set scenario1.P  = [0 1.0 0; #The @set macro is from Accessors.jl and allows to easily
make a
   0 0 1.0; # modified copy of an (immutable) struct
   0.3 0 0]

############################
# A ring of 5 queues
scenario3 = NetworkParameters(  L=5,
    _vector = ones(5),
    _vector = collect(1:5),
   P = [0  .8   0    0   0;
   0   0   .8   0   0;
   0   0   0    .8  0;
   0   0   0    0   .8;
   .8  0   0    0    0])

############################
# A large arbitrary network

#Generate some random (arbitrary) matrix P
Random.seed!(0)
L = 100
P = rand(L,L)
P = P ./ sum(P, dims=2) #normalize rows by the sum
P = P .* (0.2 .+ 0.7rand(L)) # multiply rows by factors in [0.2,0.9]

scenario4 = NetworkParameters(  L=L,
    _vector = ones(L),
    _vector = 0.5 .+ rand(L),
   P = P);
```

For each of these scenarios you can modify certain parameters. Importantly, the way we adjust for a desired ρ^* is by adjusting the `_vector`. Details below.

2 Tasks

Task 1: Thinking about Modelling Assumptions (5pts)

This task does not involve any coding or computation. Your answer can be in paragraph form or bullet point form. Consider the Modulated Jackson Network model and the application of a "Dream World" amusement park. As mentioned above, the model can be used to model customer congestion in the amusement park. However, like any model it is at best a rough approximation of reality. Describe some features that exist in the amusement park setting that are **not** captured by the model. Similarly, describe model assumptions that may be **unrealistic** for the application of the amusement park.

Also answer the following: If you were actually using this model to assess congestion at the park, would it be useful or not? What would you use such a model for in the context of the park? What operational answers can you get with the aid of such simulations? To get insight for answering this question, you may (but are not required to) reference the perspective seminar lecture by Dr. Anna Foeglein. In that lecture Anna describes her work as a simulation professional.

Task 2: Computation for Jackson Networks (12pts)

We provide you with the functions `maximal_alpha_scaling()` and `set_scenario()` for adapting parameters of a scenario for a desired ρ^* value. These functions also set modulation related parameters, κ, γ_1 and $R^{(1)}$.

```
"""
Compute the maximal value by which we can scale the  _vector and be stable.
"""
function maximal_alpha_scaling(net::NetworkParameters)
    _base = (I - net.P') \ net. _vector #Solve the traffic equations
    _base =  _base ./ averaged_service_rate(net) #Determine the load
   return minimum(1 ./  _base)
end

max_scalings = round.(maximal_alpha_scaling.([scenario1, scenario2, scenario3,
scenario4]),digits=3)
println("The maximal scalings for scenarios 1 to 4 are: $max_scalings")

"""
Use this function to adjust the network parameters to the desired  , and other parameters
"""
function set_scenario(net::NetworkParameters,  star::Float64;  ::Float64 = 2.0,  ::Float64 = 10^-8,
R1::Float64 = 1-10^-8)
      1 && error("  is out of range")
   ( star   0 ||  star   1) && error("  is out of range")
   (R1   0 || R1 > 1) && error("R¹ is out of range")
   net = @set net.  =
   net = @set net.  = net.  * R1/(1-R1) #set   based on given   and R1
   net = @set net.  =
   max_scaling = maximal_alpha_scaling(net)
   net = @set net. _vector = net. _vector*max_scaling* star #max_scaling* star is the scaling factor
applied to the original alpha
   return net
end;
```

```
The maximal scalings for scenarios 1 to 4 are: [2.0, 1.4, 0.2, 0.216]
```

Note: In all cases in this project we aim to run simulations only for networks with $\rho^* < 1$. That is only for stable networks. So in any case of testing or analysis, make sure you adjust the parameters of the network using the `set_scenario` function. This type of simulation of stable networks allows us to take long time averages from simulations which approximate "statistical steady state".

For example set the parameters of scenario 2 to have $\rho^* = 0.7$, a default κ, a value of $\gamma_1 = 0.1$, and a value of $R^{(1)} = 0.7$ we do:

```
adjusted_net = set_scenario(scenario2, 0.7;   = 0.1, R1 = 0.7)
@show adjusted_net. _vector
@show adjusted_net. , adjusted_net.

#check computations for R1 and R2
R1,R2=adjusted_net. /(adjusted_net. +adjusted_net. ),adjusted_net. /(adjusted_net. +adjusted_net. )
@show R1, R2

#We can check by solving the traffic equations
 = (I - adjusted_net.P') \ adjusted_net. _vector #Solve the traffic equations
 =   ./ averaged_service_rate(adjusted_net) #This is the vector of   values
_star= maximum( )
@show  _star;
```

```
adjusted_net. _vector = [0.637, 0.0, 0.0]
(adjusted_net. , adjusted_net. ) = (0.1, 0.23333333333333328)
(R1, R2) = (0.7, 0.3000000000000001)
 _star = 0.6999999999999998
```

Whereas if we repeat with $\kappa = 3$ you can see that now α_1 is increased because now in `fast` mode the system is faster. So to accommodate for the same target ρ^* we can have a faster arrival rate.

```
adjusted_net = set_scenario(scenario2, 0.7;   = 0.1, R1 = 0.7,   = 3.0)
@show adjusted_net. _vector

 = (I - adjusted_net.P') \ adjusted_net. _vector #Solve the traffic equations
 =   ./ averaged_service_rate(adjusted_net) #This is the vector of   values
_star= maximum( )
@show  _star;
```

```
adjusted_net. _vector = [0.7840000000000001, 0.0, 0.0]
 _star = 0.7000000000000001
```

Now, after you take time to understand the code and computations above, assume we are in one of three test cases, the **always slow** case, the **always fast** case, or the **super fast modulation** case. Since each of these cases agree with a (non-modulated) Open Jackson Network we'll use these cases as test code later. At this point, set parameters for each scenario and each of these cases (choose the γ_1 and $R^{(1)}$ parameters as you like, as long as they approximate these cases; keep $\kappa = 2$). Then for each of the 4 scenarios and each of the 3 cases create a plot of the Open Jackson Network **total steady state mean queue lengths** as a function of ρ^*. For this, simply use the fact that the theoretical steady state mean queue length for individual queue i is $\rho_i/(1-\rho_i)$. Then add the queue length of all L queues in the scenario. Note the plots you create will not be affected by the type of test case (always slow, always fast, or super fast modulation). So it is enough to present 4 plots (no need for 12). Still, you will use this code below for testing. The idea here is that you know what type of behavior to expect from the network in these cases.

Task 3: A Simulation Engine Based on Queue States (33pts)

Create a simulation engine which can be run either with the following `sim_net_trace` function or the `sim_net_estimation` function.

```
"""
Runs a discrete event simulation of a Modulated Jackson Network `net` and produces traces of queue
sizes and the `slow`-`fast` state.

The simulation runs from time `0` to `max_time`.  The simulation starts with empty queues and in
`slow` modulation state.
```

```
The return value is a trace of the queue sizes of each of the individual stations ( (time, value)
pairs). It is also a trace of the simulation state (`slow`/`fast`,time) pairs. The combination of
all these traces can be plotted to reconstruct the simulation.

This simulation does NOT keep individual customers state, it only keeps the following minimal
state:
* The number of jobs in each queue.
* A boolean value indicating if `slow` or `fast` (for the whole network)
"""
function sim_net_trace(net::NetworkParameters; max_time = 10^2, seed::Int64 = 42)

   #Set the random seed
   Random.seed!(seed)

   #The function would execute the simulation here - and this would use multiple other functions,
   types, and files

   return [] # The return value should be some data-structure which contains the traces
end;
```

```
"""
Runs a discrete event simulation of a Modulated Jackson Network `net` and collects statistics
during the run. Returns an estimate of the total mean queue length.

The simulation runs from time `0` to `max_time`.  The simulation starts with empty queues and in
`slow` modulation state.

Statistics about the total mean queue lengths are recorded from `warm_up_time` onwards and the
estimated value is returned.

This simulation does NOT keep individual customers state, it only keeps the following minimal
state:
* The number of jobs in each queue.
* A boolean value indicating if `slow` or `fast` (for the whole network)
"""
function sim_net_estimation(net::NetworkParameters; max_time = 10^6, warm_up_time = 10^4,
seed::Int64 = 42)::Float64

   #Set the random seed
   Random.seed!(seed)

   #The function would execute the simulation here - and this would use multiple other functions,
   types, and files

   estimated_total_mean_queue_length = 12.3 #the function would estimate this value (12.3 is just a
place-holder)

   return estimated_total_mean_queue_length
end;
```

The code/functions used by `sim_net_trace` and `sim_net_estimation` should be the same code as much as possible (this is the actual discrete event simulation engine). So avoid code duplication wherever possible.

Now use `sim_net_trace` to plot traces for the first three scenarios, each set to have $\rho^* = 0.8$. This plotting can help you debug the simulation as well. As for `sim_net_estimation`, at this point just see that it runs (this is a longer run). You will test it in the next task.

Tip: Implement this main task one step at a time. Then add more components until you get the full

functionality. For example, first have the network without modulation and then add modulation. Also first have deterministic routing and then add random routing. Printing out network states in short debug runs can be helpful also.

Task 4: Doing Statistical Testing (20pts)

Now you can test/explore the simulation in several ways, each time comparing to an Open Jackson Network. In all cases use `sim_net_estimation`. If there are problems with the test you may need to return to Task 3 to adjust your code.

Test #1: For each of the four scenarios, consider the **always slow** parameters that you chose. Then run a grid of ρ^* with steps of 0.01 between 0.1 and 0.9. Compare the values with the theoretical values that you plotted in the previous task. Compare both by plotting the values and computing the absolute relative error over the grid of ρ^*. You will know that your simulation is valid if the absolute relative error will be low and further decrease as `max_time` is increased.

Test #2: Repeat for the **always fast** parameters that you chose.

Test #3: Repeat for the **super fast modulation** parameters that you chose. Note that here the run time may be longer.

These tests may be placed in your `test` folder. For them to be automatic you would require thresholds as they are statistical (noisy tests). So making the tests automatic is not required.

Task 5: Analysis of the effect of modulation on Mean Queue Sizes (20pts)

Now that the simulation appears to be working correctly (based on your tests above), use your simulation engine to carry out a study. In particular we are interested in the effect of modulation on mean queue sizes. For networks with the same ρ^*, how does modulation affect mean queues? We know that as ρ^* increases from 0 towards 1, mean queue sizes increase. But what about modulation? The answer to this is not clear.

For each of the four scenarios stay with $\kappa = 2$ and $\gamma_1 = 0.1$. Now consider $R^{(1)} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.9\}$. Use these 24 parameter values (6×4) to run simulations and create plots of the estimated mean total queue length as a function of ρ^*. A total of 24 curves on four plots, one plot for each scenario (with 6 curves in a plot). In all plots, the x-axis is always ρ^* and the y-axis is the estimated total steady queue size in the network. Make sure the legend is properly marked and placed on each plot and that you have an easy script to create all plots. If you have any conclusions from your simulation plots, briefly describe them. Ponder why you get the curves that you get.

Tip: Start with short simulation runs to make sure your plots are well formatted. Then once formatting is complete, go for longer runs to decrease the variance of the estimated curves. Note that in each case you will use a grid of ρ^*, so there are many simulation runs per curve (e.g. 100) . As you develop this, start with a smaller grid.