report

October 20, 2023

1 Project 2 - 2023

1.0.1 Mitchell Holt (46977557), Joseph Kwong (46977164)

All code may be found on the GitHub repository here.

2 Perspective Seminar

Anna studied pure mathematics, and earned a PhD in analysis working with regularity theory and the Heisenberg group. Anna moved from Germany to Australia soon after finishing her PhD and started working at The Simulation Group in Brisbane. After the company was bought out by consultants, Anna moved to Integrated Logistics. Anna currently works in the mining sector with the logistics for shipping coal on behalf of several companies in each area. Anna gave some great insight into the value of simulation during her talk. It is hard to describe systems in closed forms, and hard to solve these equations. Simulation is a way to modelreal-world systems and make observations, predictions and decisions that is often far more cost- and time-efficient than exact methods.

The thing that stood out most from the talk was Anna's discussion of simulation packages, particularly AnyLogic. The Simulation Group used to sell their software in Australia, but since they were bought out by consultants, Anna acts as a sales representative for AnyLogic. Often simulation packages with have a domain-specific language to express more complex rules in simulations. Anna spoke about the importance of being able to formulate big questions for analysing data and deciding how to use simulation to address different industry concerns.

3 Task 1

We make the following assumptions: - The simulation reaches a steady state before the end of warm-up time, for any warm-up time. - For task 5, we assume that there are no breakdowns.

Furthermore, the following assumptions for the simulation, implicit or explicit, are unreasonable in the context of simulating queues at a theme park: - Only one person is served at any one time at a given station. Realistically, we would expect that either 1. multiple people are served at the same time by a single server (e.g. a rollercoaster), or 2. there is more than one server at the station (e.g. a fast food stand). - Arrival rates are uniformly distributed independently to time. We would expect that the arrival rates for queues (how busy the theme park is) depends on the time of day. - People *only* leave the queue once they have been served. We would expect that people could give up on a queue at any time, decreasing the queue length independently of the server. - We have assumed that the rate of arrivals to a queue is independent of its length. However, we would expect that arrival rates for longer queues are lower than those for shorter queues. - When we are keeping track of how long individual customers have spent in queues, we additionally need to assume that no customers switch positions in the queue.

4 Task 2

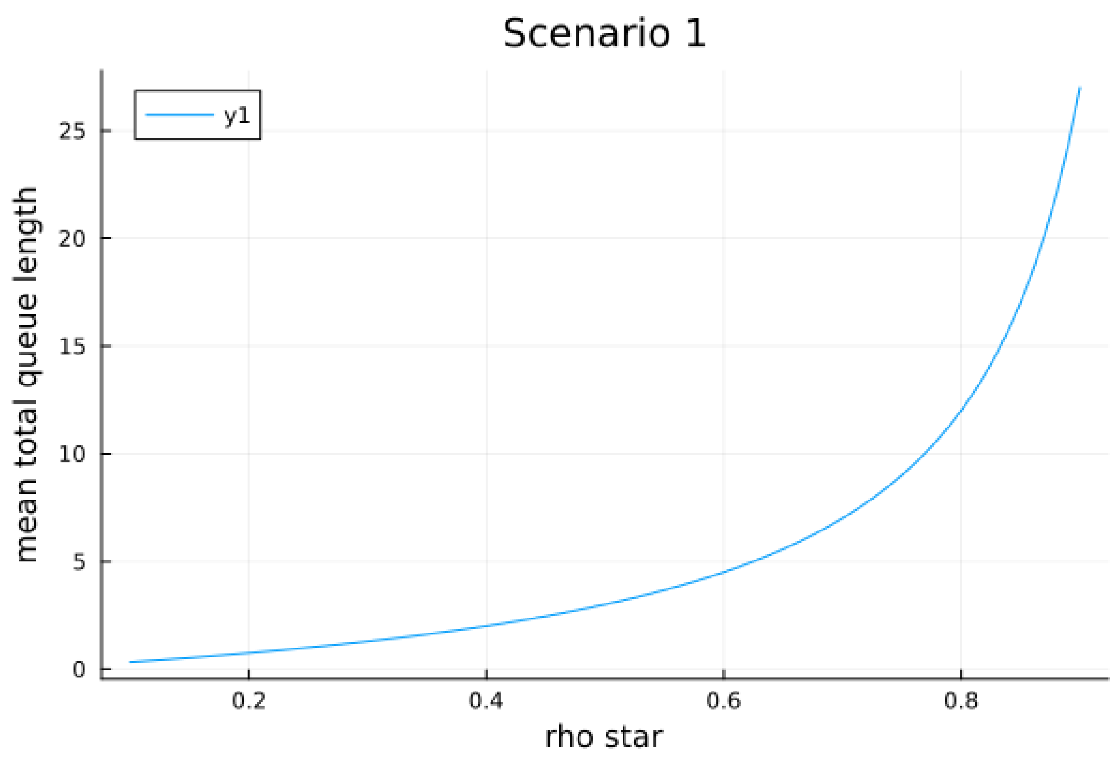
In this task, we plot the theoretical mean total queue length of the network against ρ^*. The following can be found in `test/task_2.jl`.

[ ]:
```
"""
Returns a plot of the theoretical total mean queue length for different values␣
 ↪of rho^*.
"""
function plot_theoretical_mean_queue_length(parameters::NetworkParameters,␣
 ↪scenario_number::Int)
    rho_stars = 0.1:0.01:0.9
    theoretical_mean_queue_lengths = zeros(length(rho_stars))

    for (i, rho_star) in enumerate(rho_stars)
        rho = compute_rho(set_scenario(parameters, rho_star = rho_star))
        theoretical_mean_queue_lengths[i] = sum(rho ./ (1 .- rho))
    end

    plot(rho_stars,
        theoretical_mean_queue_lengths,
        title = "Scenario $(scenario_number)",
        xlabel = "rho star",
        ylabel = "mean total queue length")
end;
```

Applying the function above to the four scenarios give the following plots:

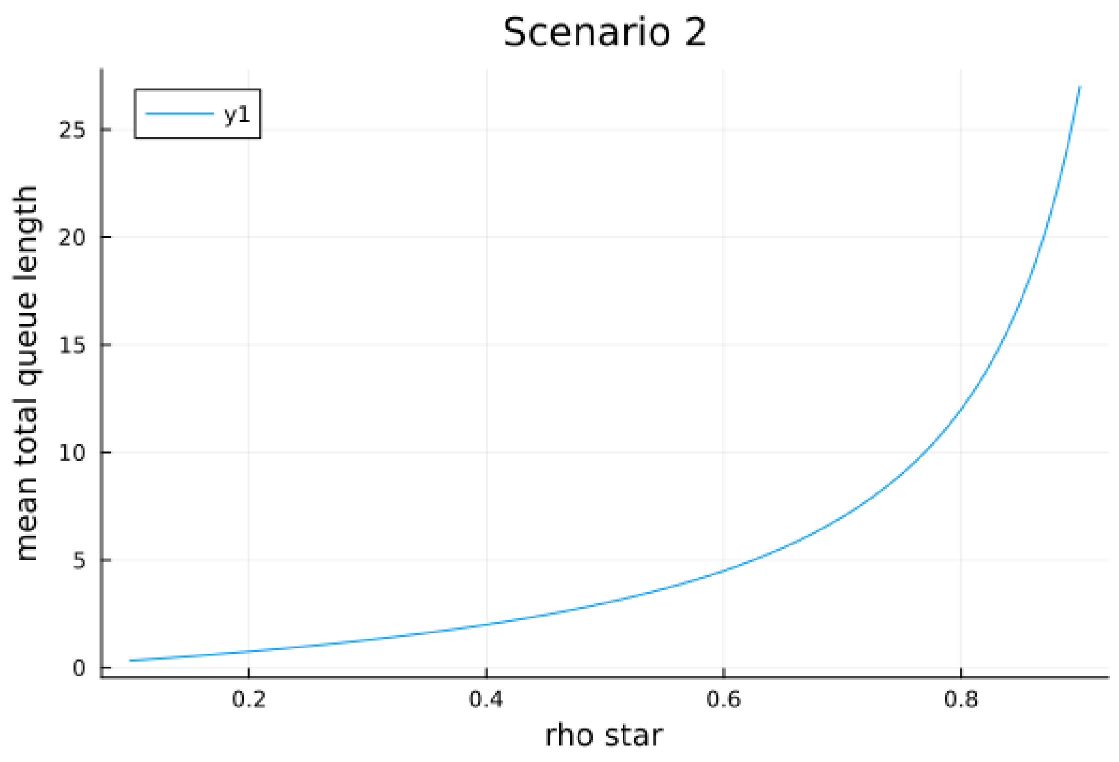
[31]:
```
using Images, FileIO
load("img/task_2_scenario_1.png")
```

[31]:



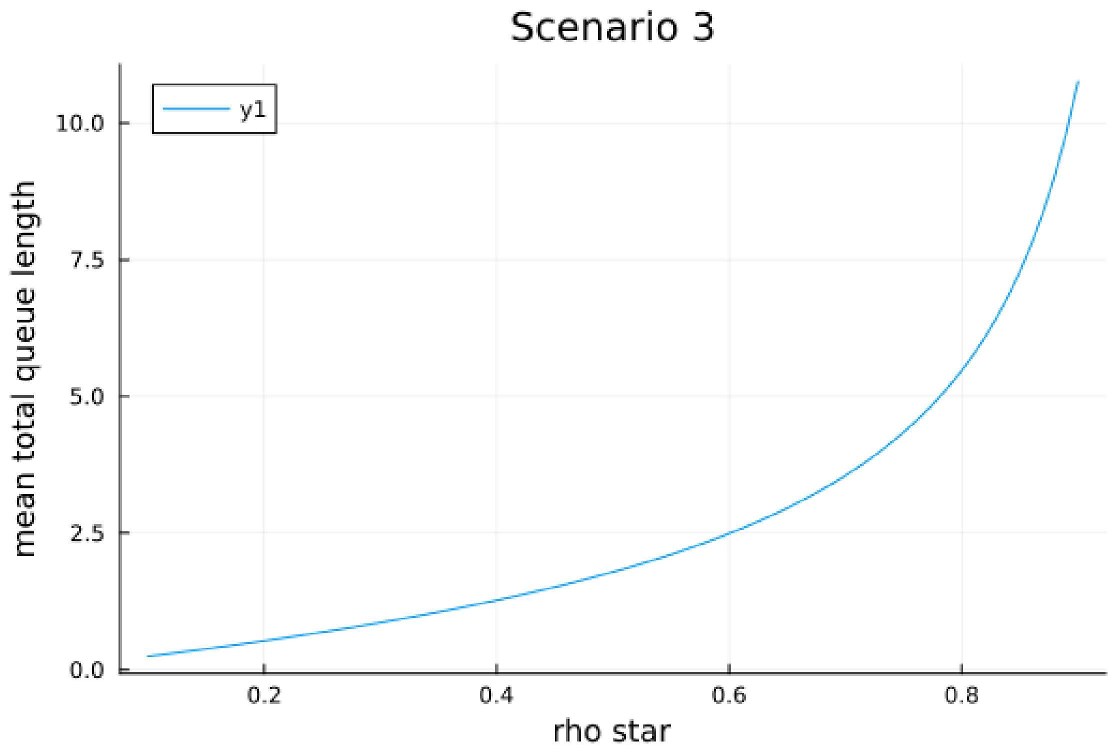
[34]:
```
load("img/task_2_scenario_2.png")
```

[34]:



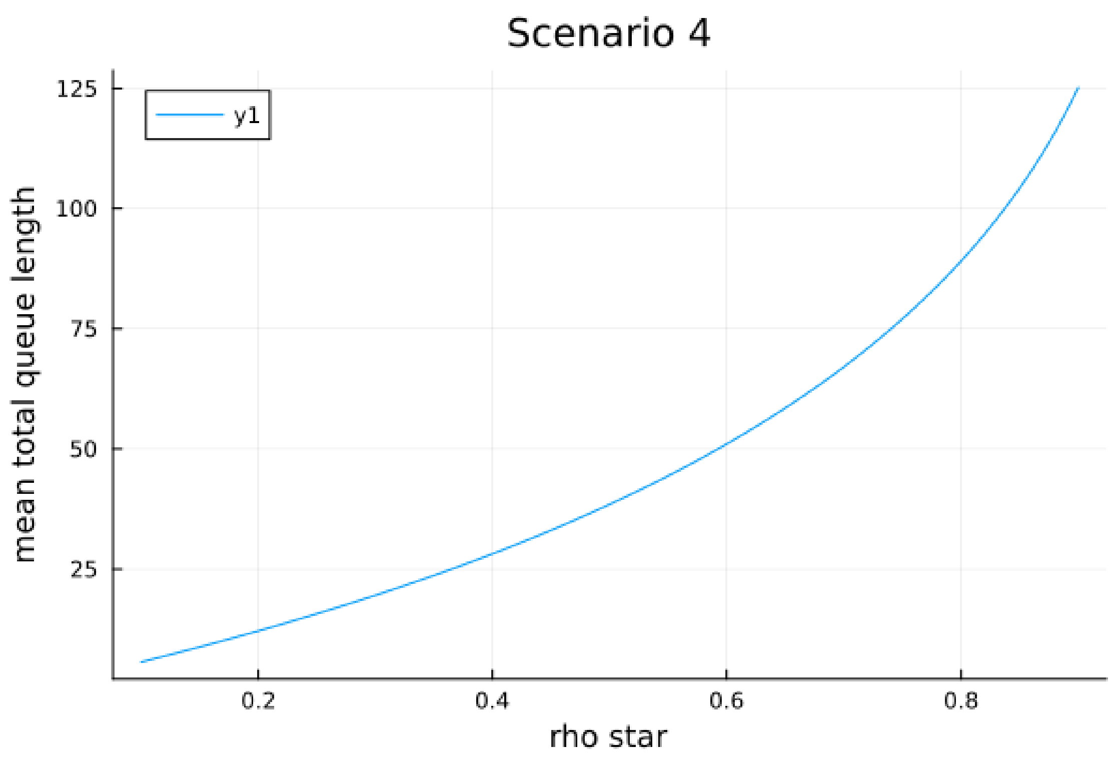
[35]:
```
load("img/task_2_scenario_3.png")
```

[35]:



[36]:
```
load("img/task_2_scenario_4.png")
```

[36]:



5 Task 3

To start the project, we defined the `Parameters` struct, and defined some functions which query or change a `Parameters` struct. The following code can be found in `src/parameters.jl`.

[ ]:
```
"""
This struct stores the parameters for a simulation.

    - L is the number of servers.
    - alpha_vector is a vector of external arrival rates.
    - mu_vector is a vector of services rates.
    - P is the L by L routing matrix.
    - c_s is the squared coefficients of the variation of service processes.
    - gamma_1 is the rate for "on" durations.
    - gamma_2 is the rate for "off" durations.
"""
@with_kw struct NetworkParameters
    L::Int
    alpha_vector::Vector{Float64}
    mu_vector::Vector{Float64}
    P::Matrix{Float64}
    c_s::Float64 = 1.0
    gamma_1::Float64 = 0
```

```julia
    gamma_2::Float64 = 1.0
end

"""
Computes the lambda vector given parameters.
"""
function compute_lambda(parameters::NetworkParameters)
    return (I - parameters.P') \ parameters.alpha_vector
end

"""
Computes the rho vector given parameters.
"""
function compute_rho(parameters::NetworkParameters)
    lambda = compute_lambda(parameters)
    return lambda ./ parameters.mu_vector
end

"""
Computes the proportion the servers are on, R, given parameters.
"""
function compute_R(parameters::NetworkParameters)
    return parameters.gamma_2/(parameters.gamma_1 + parameters.gamma_2)
end

"""
Computes the vector R * mu, given parameters.
"""
function service_capacity(parameters::NetworkParameters)
    return compute_R(parameters) * parameters.mu_vector
end

"""
Computes the maximal value for which we can scale alpha so that the parameter␣
 ↪is
still stable.
"""
function maximal_alpha_scaling(parameters::NetworkParameters)
    lambda_base = (I - parameters.P') \ parameters.alpha_vector
    rho_base = lambda_base ./ parameters.mu_vector
    return minimum(1 ./ rho_base)
end

"""
Adjust a scenario by choosing rho_star, c_s, and R.
"""
```

```julia
function set_scenario(parameters::NetworkParameters; rho_star::Float64=0.5, c_s:
↪:Float64=1.0, R::Float64 = 1.0)
    (rho_star <= 0 || rho_star >= 1) && error("Rho is out of range")
    (R <= 0 || R > 1) && error("R is out of range")
    parameters = @set parameters.gamma_1 = parameters.gamma_2 * (1-R)/R
    max_scaling = maximal_alpha_scaling(parameters)
    parameters = @set parameters.alpha_vector = parameters.alpha_vector *␣
↪max_scaling * rho_star
    parameters = @set parameters.c_s = c_s
    return parameters
end
```

Next, we defined the type `State`, and the subtype `NetworkState`, which does *not* track data about customers. The following code can be found in `src/state.jl`.

Our idea for implementing breakdowns was the following. First, whether or not each server is on or off is stored in `server_status`. The vector `additional_times` stores the amount of time that jobs are frozen. When an `EndOfServiceEvent` is processed at the qth server, we only treat the event as an actual end of service if the associatived job was never frozen, that is, `additional_times[q] == 0`. If the job was frozen (`additional_times[q] > 0`), then the `EndOfServiceEvent` is ignored, but a new `EndOfServiceEvent` is spawned at the current time plus `additional_times[q]`, then `additional_times[q]` is reset to zero. This represents the job continuing after the scheduled end of service time (i.e. the new `EndOfServiceEvent` is still associated to the same job).

```julia
[ ]: """
An abstract type to represent the state of a simulation at a point in time.
"""
abstract type State end

"""
This mutable struct stores the state of a simulation, where we do NOT keep track
of individual customers.

    - queues stores the current number of jobs in each queue.
    - arrivals stores the total number of (both external and internal)
        arrivals at each server.
    - server_status is a boolean vector representing whether the servers are on␣
↪(1)
        or off (0).
    - the qth term of additional_times represents the time that a job was␣
↪frozen
        because the qth server was off.
    - the qth term of last_off is a vector which stores the last time that the␣
↪qth server
        was off or started processing a new job.
"""
mutable struct NetworkState <: State
    queues::Vector{Int64}
    arrivals::Vector{Int64}
    server_status::Vector{Bool}
    additional_times::Vector{Float64}
    last_off::Vector{Float64}
    parameters::NetworkParameters

    function NetworkState(parameters::NetworkParameters)
        L = parameters.L
        return new(zeros(Int, L), zeros(Int, L), ones(Bool, L),zeros(Float64,␣
 ↪L), zeros(Float64, L), parameters)
    end
end
```

We defined some functions which helped us randomly generate the necessary durations arrivals, services, breakdowns, and repairs. We also wrote a function to randomly generate the next location of a customer after they were served. The following code can be found in `src/sampling.jl`.

```
[ ]: """
A convenience function to make a Gamma distribution with desired rate
(inverse of shape) and SCV.
"""
rate_scv_gamma(rate::Float64, scv::Float64) = Gamma(1/scv, scv/rate)

"""
Generates the time until the next external arrival at the qth server.
"""
function next_arrival_duration(state::State, q::Int)
    return rand(Exponential(1/state.parameters.alpha_vector[q]))
end

"""
Generates the time it takes to process a job at the qth server.
"""
function next_service_duration(state::State, q::Int)
    return rand(rate_scv_gamma(state.parameters.mu_vector[q], state.parameters.
 ↪c_s))
end

"""
Generates the time for a server to stay on.
"""
function next_off_duration(state::NetworkState)
    return rand(Exponential(1/state.parameters.gamma_1))
end

"""
Generates the time for a server to stay off.
```

```
"""
function next_on_duration(state::NetworkState)
    return rand(Exponential(1/state.parameters.gamma_2))
end

"""
Generates the next location for a job at the qth server after it
has been served.
"""
function next_location(state::State, q::Int)
    L = state.parameters.L
    P = state.parameters.P
    weights = [P[q,:];[1 - sum(P[q,:])]]
    return sample(1:L+1, Weights(weights))
end
```

Next, we implemented `Events`. The following code can be found in `src/events.jl`.

We implemented the servers switching on and off by creating two new events, `ServerOffEvent` and `ServerOnEvent`. When these events are processed, they spawn the other according to the parameters γ_1 and γ_2.

[ ]:
```
"""
An abstract type to represent all events that happen in the simulation.
"""
abstract type Event end

"""
The end of the simulation.
"""
struct EndSimEvent <: Event end

"""
A person arriving at the qth server.
"""
struct ArrivalEvent <: Event
    q::Int
end

"""
The qth server finishes serving a customer (if there are no breakdowns).
"""
struct EndOfServiceEvent <: Event
    q::Int
end

"""
The qth server breaks down.
```

```
"""
struct ServerOffEvent <: Event
    q::Int
end

"""
The qth server is repaired.
"""
struct ServerOnEvent <: Event
    q::Int
end

"""
A struct which consists of an event and a time stamp, which allows us
to keep track of when events happen.
"""
struct TimedEvent
    event::Event
    time::Float64
end

"""
Compares two TimedEvents based on when they occur. This allows us to
order TimedEvents.
"""
isless(te1::TimedEvent, te2::TimedEvent) = te1.time < te2.time

"""
This function returns a vector of TimedEvents, based on the state of the
simulation at the current time, and the event being processed.
"""
function process_event end

"""
Processes an EndSimEvent by doing nothing.
"""
function process_event(time::Float64, state::State, event::EndSimEvent)
    return TimedEvent[]
end

"""
Processes an external arrival at the qth server.
"""
function process_event(time::Float64, state::NetworkState, event::ArrivalEvent)
    q = event.q
    #@show q
    state.queues[q] += 1
```

```julia
    state.arrivals[q] += 1
    new_timed_events = [TimedEvent(ArrivalEvent(q), time +␣
 ↪next_arrival_duration(state, q))]

    # we just added the only person to the queue, so they can start being served
    # now. Add an end of service event.
    if state.queues[q] == 1
        state.additional_times[q] = 0
        state.last_off[q] = time
        push!(new_timed_events, TimedEvent(EndOfServiceEvent(q), time +␣
 ↪next_service_duration(state, q)))
    end
    return new_timed_events
end

"""
If there were no breakdowns during the service of this job
(i.e. there is no additional time) then we process the end of a job at the qth␣
 ↪server.

If a breakdown occurred during service, then  we do nothing except spawn an
endOfServiceEvent which is processed after the additional time has lapsed.
"""
function process_event(time::Float64, state::NetworkState, event::
 ↪EndOfServiceEvent)
    q = event.q
    new_timed_events = TimedEvent[]

    if state.additional_times[q] == 0
        state.queues[q] -= 1

        if state.queues[q] > 0
            push!(new_timed_events, TimedEvent(EndOfServiceEvent(q), time +␣
 ↪next_service_duration(state, q)))
        end

        next_q = next_location(state, q)
        if next_q <= state.parameters.L
            state.queues[next_q] += 1
            state.arrivals[next_q] += 1

            if state.queues[next_q] == 1
                state.last_off[next_q] = time
                state.additional_times[next_q] = 0
                push!(new_timed_events, TimedEvent(EndOfServiceEvent(next_q),␣
 ↪time + next_service_duration(state, next_q)))
```

```
            end
        end
    else
        push!(new_timed_events, TimedEvent(EndOfServiceEvent(q), time + state.
 ↪additional_times[q]))
        state.additional_times[q] = 0
        state.last_off[q] = time
    end

    return new_timed_events
end

"""
Processes a server breaking down.
"""
function process_event(time::Float64, state::NetworkState, event::
 ↪ServerOffEvent)
    q = event.q
    new_timed_events = TimedEvent[]
    state.server_status[q] = false
    state.last_off[q] = time

    push!(new_timed_events, TimedEvent(ServerOnEvent(q), time +␣
 ↪next_on_duration(state)))
    return new_timed_events
end

"""
Processes a server repairing.
"""
function process_event(time::Float64, state::State, event::ServerOnEvent)
    q = event.q
    new_timed_events = TimedEvent[]
    state.server_status[q] = true

    if state.queues[q] >= 1
        state.additional_times[q] = time - state.last_off[q]
    end

    push!(new_timed_events, TimedEvent(ServerOffEvent(q), time +␣
 ↪next_off_duration(state)))
    return new_timed_events
end
```

Finally, we implemented the main simulation loop. The following code can be found in `src/simulation.jl`.

[38]:
```julia
"""
Simulates a generalised unreliable Jackson network with given parameters.

This simulation does NOT keep track of individual customers.
"""
function sim_net(parameters::NetworkParameters; state =␣
↪NetworkState(parameters),
        max_time = 10^6, warm_up_time = 10^4, seed::Int64 = 42,
        callback = (_, _) -> nothing)

    Random.seed!(seed)

    timed_event_heap = BinaryMinHeap{TimedEvent}()

    push!(timed_event_heap, TimedEvent(EndSimEvent(), max_time))

    for q in 1:parameters.L
        if parameters.alpha_vector[q] > 0
            push!(timed_event_heap, TimedEvent(ArrivalEvent(q),␣
↪next_arrival_duration(state, q)))
        end

        if parameters.gamma_1 > 0
            push!(timed_event_heap, TimedEvent(ServerOffEvent(q),␣
↪next_off_duration(state)))
        end
    end

    time = 0.0

    callback(state, time)

    while true
        timed_event = pop!(timed_event_heap)
        time = timed_event.time
        new_timed_events = process_event(time, state, timed_event.event)

        callback(state, time)

        isa(timed_event.event, EndSimEvent) && break

        for nte in new_timed_events
            push!(timed_event_heap, nte)
        end
    end
end;
```

5.1 Test 1

In this task, we plot the simulated and theoretical mean total lengths of queues against different values of ρ^*, and also plot the absolute relative errors. We decided to omit `max_time = 10^6`, so that the tests could finish in a reasonable amount of time. The following code can be found in `test/task_3_test_1.jl`.

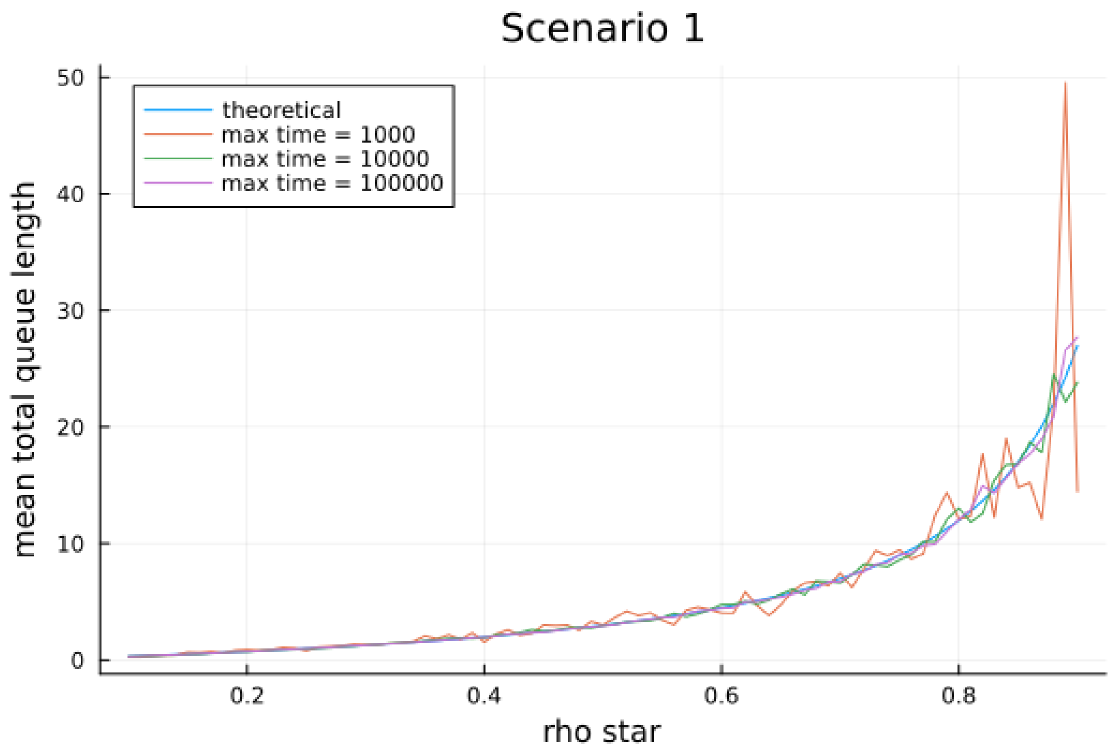
```
[ ]: """
Returns two plots, p1 and p2.

The first plot p1 shows the simulated total mean queue length and the
theoretical result for different max times and values of rho^*.

The second plot p2 shows the absolute relative error between the simulated and␣
 ↪theoretical
results for different max times and values of rho^*.
"""
function plot_simulated_mean_queue_length(parameters::NetworkParameters,␣
 ↪scenario_number::Int)
    rho_stars = 0.1:0.01:0.9
    max_times = [10^3,10^4,10^5]
    warm_up_times = [10^2,10^3,10^4]

    theoretical_results = Vector{Float64}(undef, length(rho_stars))
    for (i, rho_star) in collect(enumerate(rho_stars))
        new_parameters = set_scenario(parameters, rho_star = rho_star)
        rho = compute_rho(new_parameters)
        theoretical_results[i] = sum(rho ./ (1 .- rho))
    end

    simulation_data = [theoretical_results]
    error_data = []

    for (i, max_time) in enumerate(max_times)
        warm_up_time = warm_up_times[i]
        simulation_results = Vector{Float64}(undef, length(rho_stars))
        absolute_relative_errors = Vector{Float64}(undef, length(rho_stars))

        for (i, rho_star) in collect(enumerate(rho_stars))
            new_parameters = set_scenario(parameters, rho_star = rho_star)

            riemann_sum = 0.0
            prev_total = 0.0
            prev_time = 0.0

            function record_integral(state::NetworkState, time::Float64)
                (time >= warm_up_time) && (riemann_sum += prev_total * (time -␣
 ↪prev_time))
```

```
                prev_total = sum(state.queues)
                prev_time = time
            end

            sim_net(new_parameters, max_time = max_time, warm_up_time =␣
↪warm_up_time,
                callback = record_integral)

            simulation_results[i] = riemann_sum / (max_time - warm_up_time)
            absolute_relative_errors[i] = abs((simulation_results[i] -␣
↪theoretical_results[i]) / theoretical_results[i])
        end

        push!(simulation_data, simulation_results)
        push!(error_data, absolute_relative_errors)

    end


    p1 = plot(rho_stars, simulation_data, title = "Scenario␣
↪$(scenario_number)",
    xlabel = "rho star", ylabel = "mean total queue length",
    labels=["theoretical" "max time = 1000" "max time = 10000"  "max time =␣
↪100000"])

    p2 = plot(rho_stars, error_data, title = "Scenario $(scenario_number)",
    xlabel = "rho star", ylabel = "absolute relative error",
    labels=["max time = 1000" "max time = 10000"  "max time = 100000"])
    return p1, p2
end
```

Applying the function above to the four scenarios yields the following plots. Observe that the relative errors are small, and further decrease as `max_time` increases. This indicates that our simulation is valid.

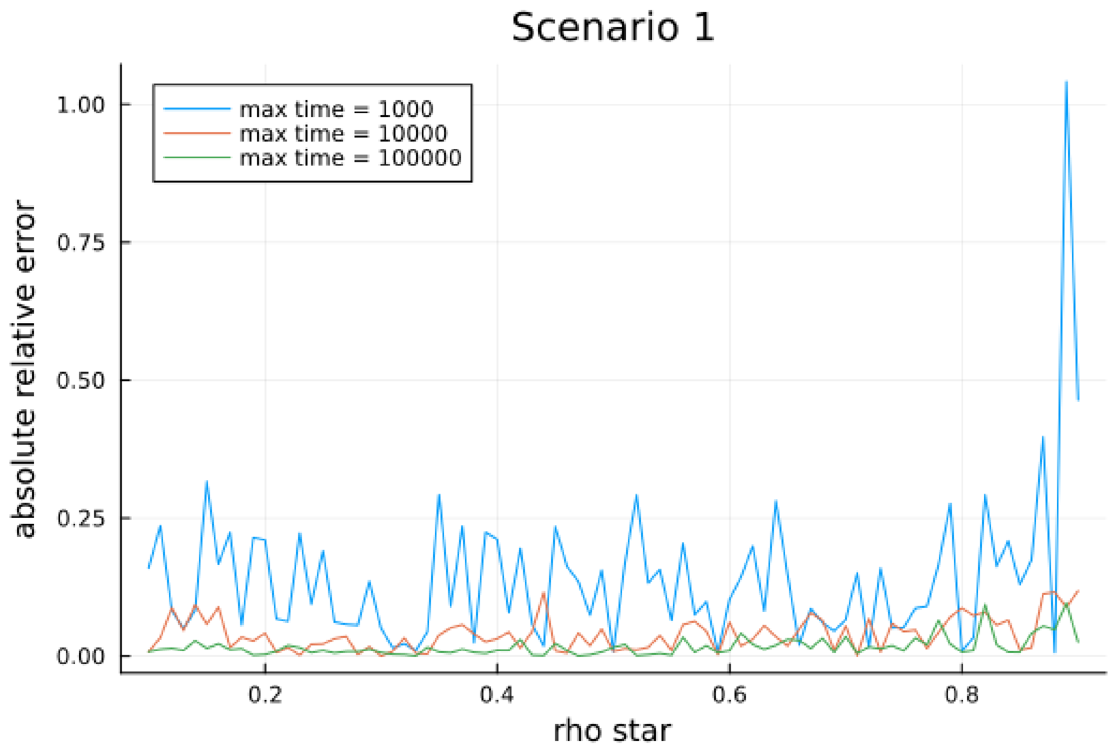
[39]:
```
load("img/task_3_test_1_scenario_1_simulated.png")
```

[39]:



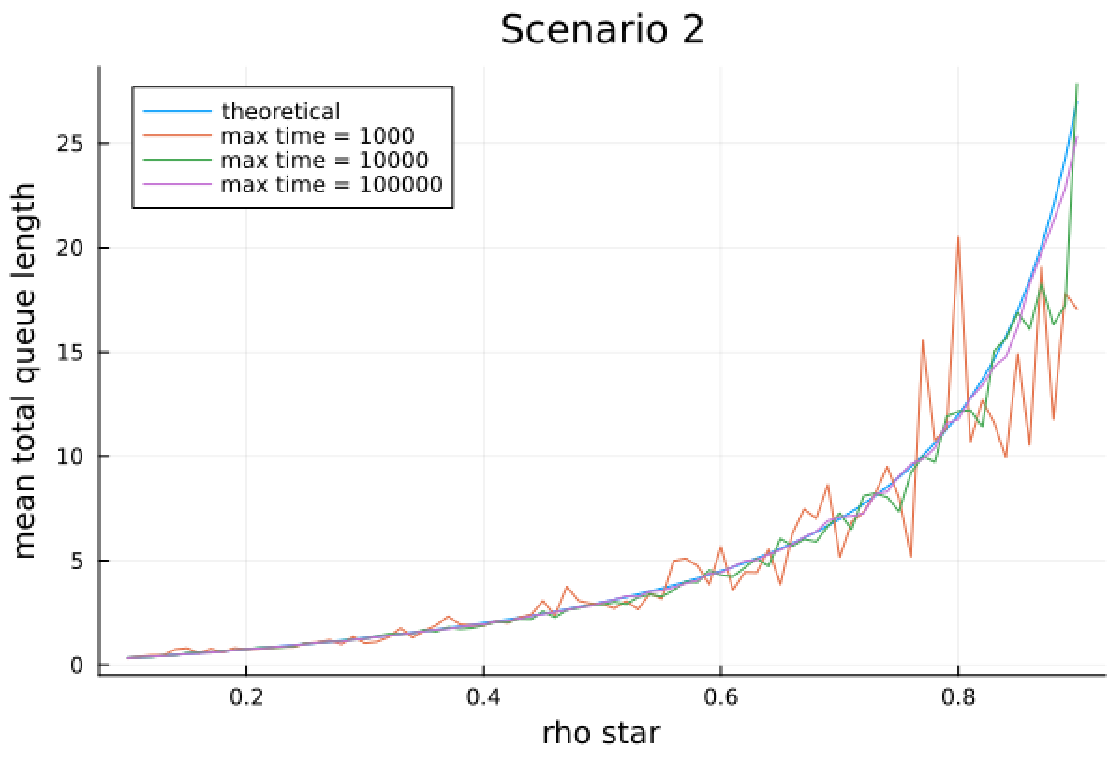
[40]:
```
load("img/task_3_test_1_scenario_1_error.png")
```

[40]:



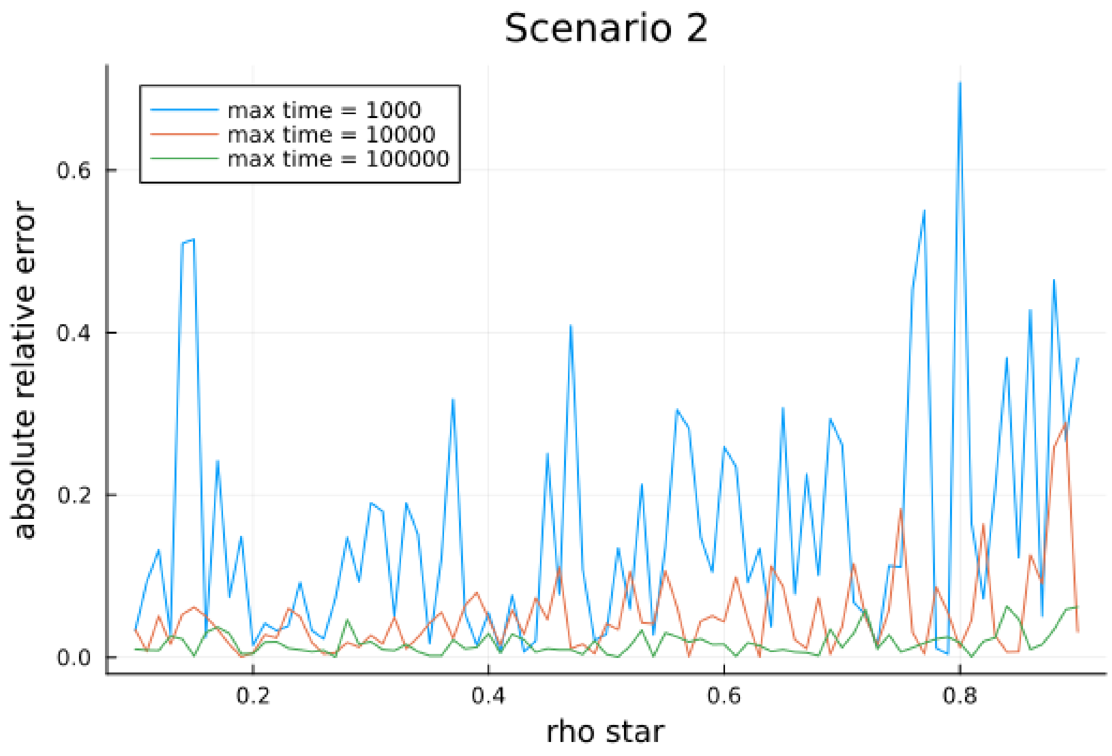
[41]:
```
load("img/task_3_test_1_scenario_2_simulated.png")
```

[41]:



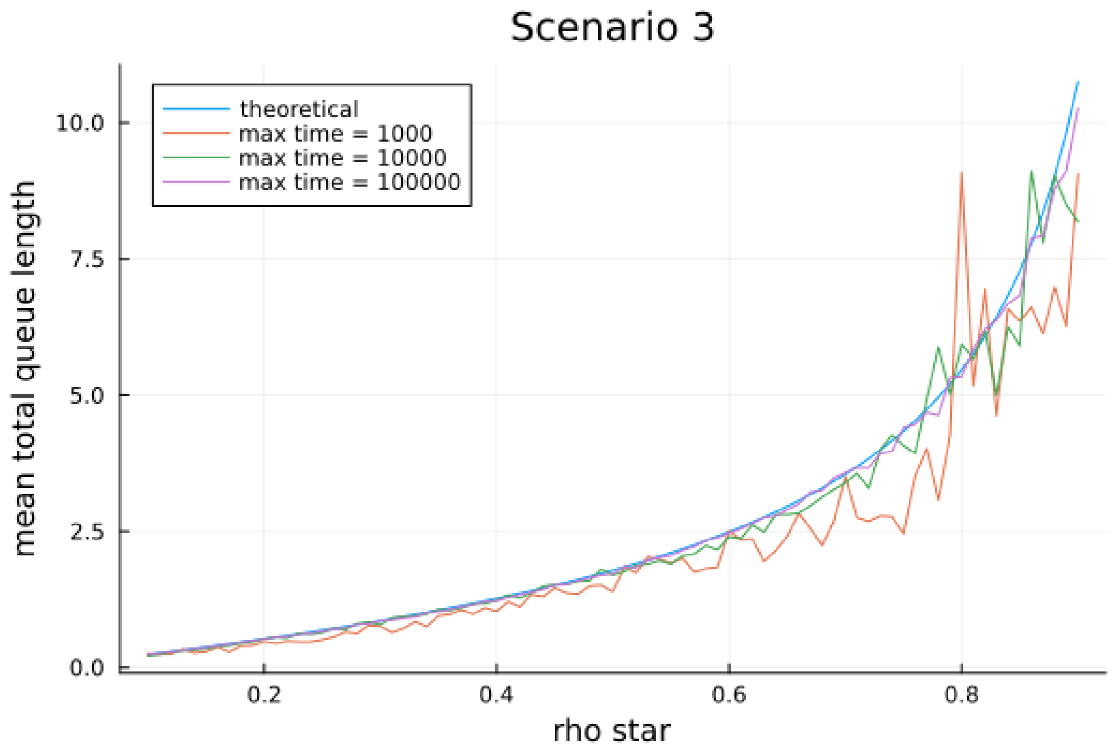
[42]:
```
load("img/task_3_test_1_scenario_2_error.png")
```

[42]:

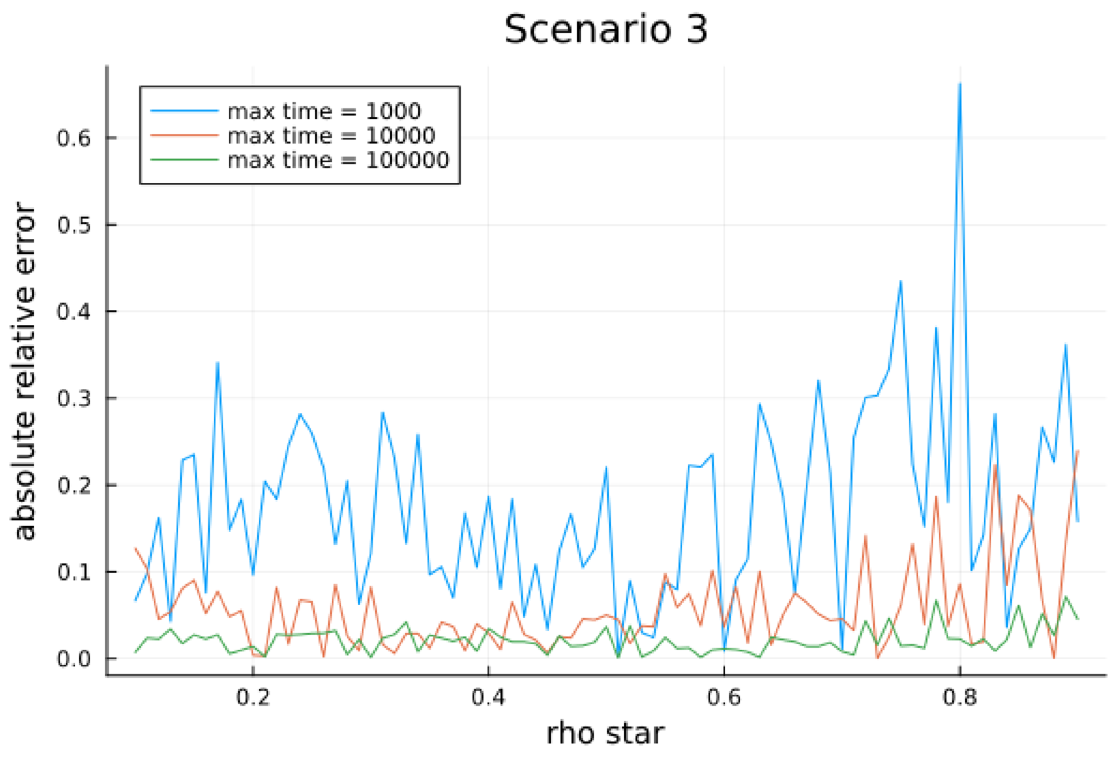


```
[43]: load("img/task_3_test_1_scenario_3_simulated.png")
```

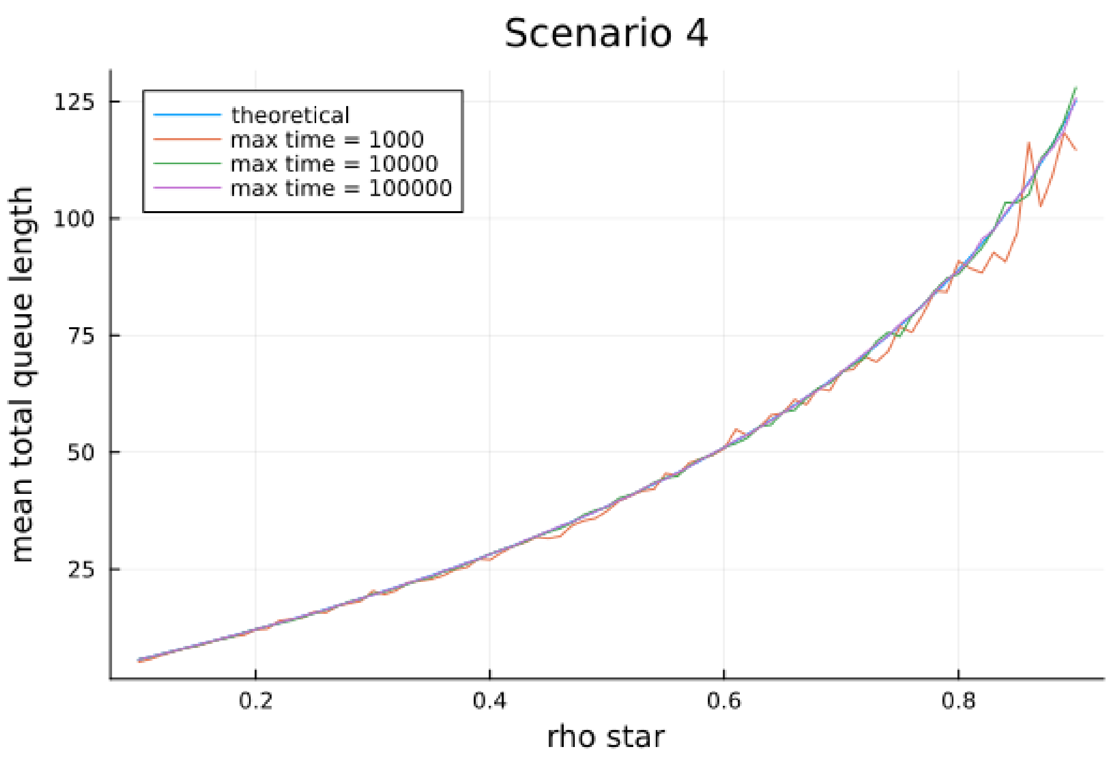
[43]:



[44]:
```
load("img/task_3_test_1_scenario_3_error.png")
```
[44]:

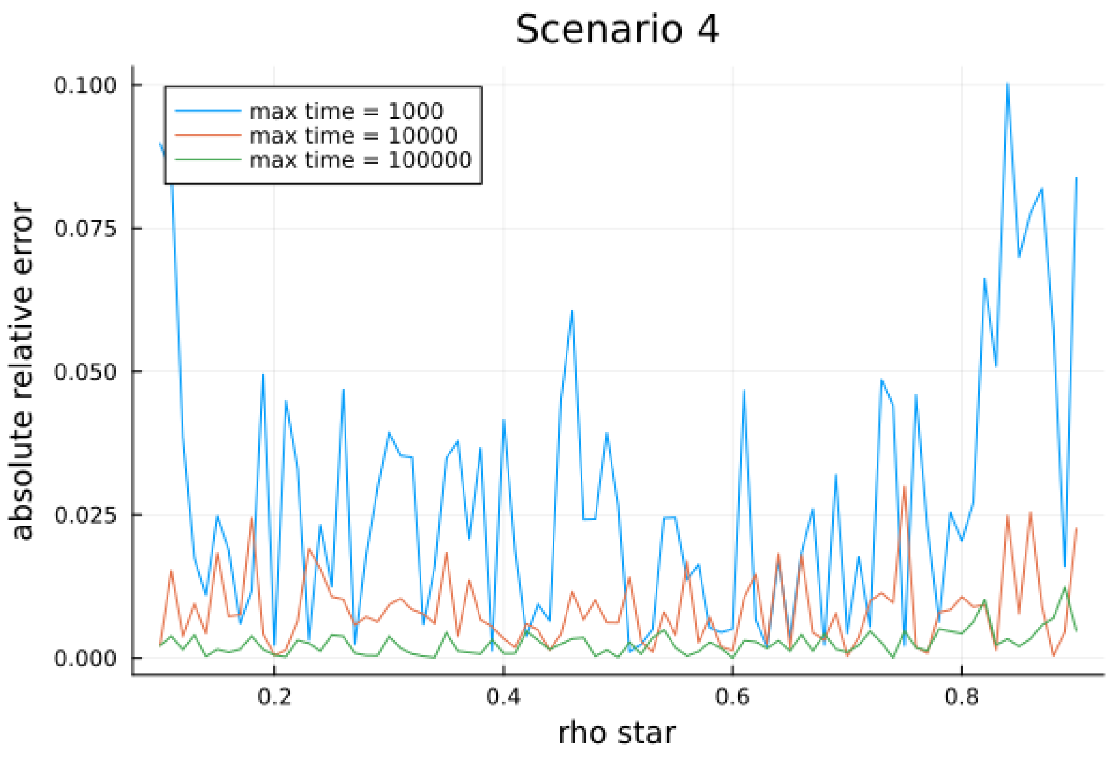


[45]:
```
load("img/task_3_test_1_scenario_4_simulated.png")
```
[45]:



[46]:
```
load("img/task_3_test_1_scenario_4_error.png")
```

[46]:



5.2 Test 2

In this task, we compare the theoretical and simulated arrival rates. The following code can be found in `test/task_3_test_2.jl`.

[49]:
```julia
"""
Prints the simulated arrival rates and theoretical arrival rates for different
values of c_s. We also print the sum of the square errors between the simulated
and theoretical results.
"""
function compare_arrival_rates(parameters::NetworkParameters, scenario_number::
 ↪Int)
    c_s_values = [0.1,0.5,1,2,4]
    max_time = 10^5
    rho_star = 0.5
    for (i,c_s) in enumerate(c_s_values)
        parameters = set_scenario(parameters, rho_star = rho_star, c_s =␣
 ↪c_s_values[i])
        state = NetworkState(parameters)
        sim_net(parameters, state = state,max_time = max_time)
        println("c_s = $(c_s):")
        simulated_arrival_rates = state.arrivals/max_time
        theoretical_arrival_rates = compute_lambda(parameters)
        sum_square_errors = sum((simulated_arrival_rates .-␣
 ↪theoretical_arrival_rates).^2)
        max_index = min(5, parameters.L)
        println("The simulated arrival rates are: ", simulated_arrival_rates[1:
 ↪max_index])
        println("The theoretical arrival rates are: ",␣
 ↪theoretical_arrival_rates[1:max_index])
        println("The sum of square errors is: ", sum_square_errors)
    end
end;
```

Applying the function above to the four scenarios gives the following output. Observe that the simulated arrival rates are close to the theoretical arrival rates, since the sums of square errors are small. We have suppressed the output for scenario 4 so that only the arrival rates of the first five servers is shown.

```
Scenario 1:
c_s = 0.1:
The simulated arrival rates are: [0.49767, 0.49766, 0.49765]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 1.6427000000000122e-5
c_s = 0.5:
The simulated arrival rates are: [0.4975, 0.49749, 0.49748]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 1.8900500000000183e-5
c_s = 1.0:
The simulated arrival rates are: [0.49692, 0.49692, 0.49692]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 2.84592000000005e-5
c_s = 2.0:
The simulated arrival rates are: [0.49709, 0.49709, 0.49709]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 2.5404300000000417e-5
c_s = 4.0:
The simulated arrival rates are: [0.49952, 0.49952, 0.49952]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 6.911999999999437e-7

Scenario 2:
c_s = 0.1:
The simulated arrival rates are: [0.49638, 0.49638, 0.49637]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 3.938570000000034e-5
c_s = 0.5:
The simulated arrival rates are: [0.50072, 0.50072, 0.50072]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 1.555200000000233e-6
c_s = 1.0:
The simulated arrival rates are: [0.49833, 0.4983, 0.49828]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 8.637299999999944e-6
c_s = 2.0:
The simulated arrival rates are: [0.49662, 0.49662, 0.49662]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 3.427319999999988e-5
c_s = 4.0:
The simulated arrival rates are: [0.49649, 0.49647, 0.49647]
The theoretical arrival rates are: [0.5, 0.5, 0.5]
The sum of square errors is: 3.7241899999999776e-5

Scenario 3:
c_s = 0.1:
The simulated arrival rates are: [0.49512, 0.49519, 0.49451, 0.4967, 0.49463]
The theoretical arrival rates are: [0.5, 0.5, 0.5, 0.5, 0.5]
The sum of square errors is: 0.00011681749999999974
c_s = 0.5:
The simulated arrival rates are: [0.49425, 0.49511, 0.49185, 0.49003, 0.49056]
The theoretical arrival rates are: [0.5, 0.5, 0.5, 0.5, 0.5]
The sum of square errors is: 0.0003119115999999993
c_s = 1.0:
The simulated arrival rates are: [0.49182, 0.49048, 0.49118, 0.49208, 0.49198]
The theoretical arrival rates are: [0.5, 0.5, 0.5, 0.5, 0.5]
The sum of square errors is: 0.0003623819999999999
c_s = 2.0:
The simulated arrival rates are: [0.50224, 0.50162, 0.50215, 0.5025, 0.5032]
The theoretical arrival rates are: [0.5, 0.5, 0.5, 0.5, 0.5]
The sum of square errors is: 2.875449999999949e-5
c_s = 4.0:
The simulated arrival rates are: [0.50055, 0.49931, 0.49728, 0.50001, 0.49936]
The theoretical arrival rates are: [0.5, 0.5, 0.5, 0.5, 0.5]
The sum of square errors is: 8.586700000000055e-6

Scenario 4:
c_s = 0.1:
The simulated arrival rates are: [0.24738, 0.24875, 0.24821, 0.2505, 0.24386]
The theoretical arrival rates are: [0.24933377605080587, 0.24936105646502282, 0.24682100538590
The sum of square errors is: 0.0002753083740300219
c_s = 0.5:
The simulated arrival rates are: [0.24712, 0.24941, 0.24644, 0.24624, 0.24346]
The theoretical arrival rates are: [0.24933377605080587, 0.24936105646502282, 0.24682100538590
The sum of square errors is: 0.0002645074897684001
c_s = 1.0:
The simulated arrival rates are: [0.24785, 0.25025, 0.24692, 0.25028, 0.24706]
The theoretical arrival rates are: [0.24933377605080587, 0.24936105646502282, 0.24682100538590
The sum of square errors is: 0.00020232150086946273
c_s = 2.0:
The simulated arrival rates are: [0.25224, 0.24801, 0.24712, 0.24981, 0.25048]
The theoretical arrival rates are: [0.24933377605080587, 0.24936105646502282, 0.24682100538590
The sum of square errors is: 0.0002806431890097714
c_s = 4.0:
The simulated arrival rates are: [0.2486, 0.25172, 0.24922, 0.24663, 0.24617]
The theoretical arrival rates are: [0.24933377605080587, 0.24936105646502282, 0.24682100538590
The sum of square errors is: 0.00020965398949963096
```

5.3 Test 3

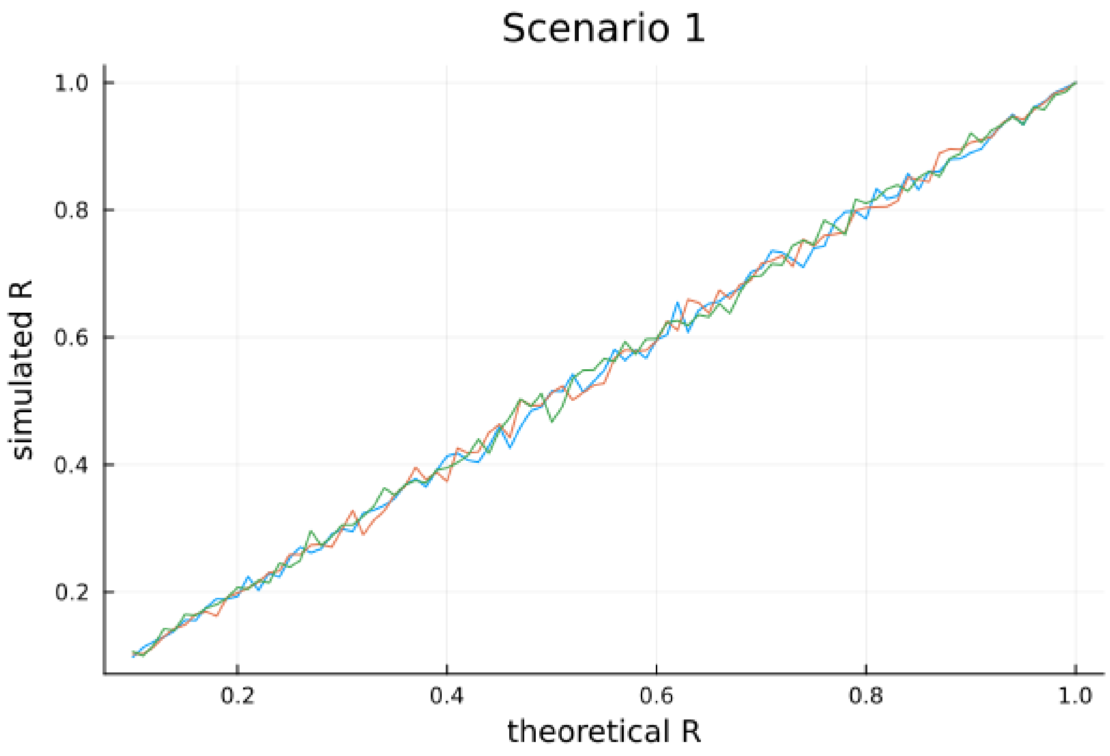
In this task, we plot the simulated R value against the theoretical R value. The following code can be found in `test/task_3_test_3.jl`.

[51]:
```
"""
Returns a plot which shows the simulated R value for different theoretical R␣
 ↪values.
"""
function plot_simulated_R(parameters::NetworkParameters, scenario_number::Int)
    Rs = 0.1:0.01:1
    simulated_Rs = [Vector{Float64}(undef, length(Rs)) for _ in 1:
 ↪min(parameters.L, 10)]
    max_time = 1000
    for (i, R) in enumerate(Rs)
        new_parameters = set_scenario(parameters, R = R)

        riemann_sums = [0.0 for _ in 1:parameters.L]
        prev_time = 0.0
        prev_status = [0 for _ in 1:parameters.L]

        function record_integral(state::NetworkState, time::Float64)
            riemann_sums = @. riemann_sums + ((time - prev_time) * prev_status)
            prev_status = copy(state.server_status)
            prev_time = time
        end

        sim_net(new_parameters, max_time = max_time, callback = record_integral)

        for j in 1:min(parameters.L, 10)
            simulated_Rs[j][i] = riemann_sums[j] / max_time
        end

    end

    plot(Rs, simulated_Rs, title = "Scenario $(scenario_number)",
    xlabel = "theoretical R", ylabel = "simulated R",
    legend = false)
end;
```

The function above applied to the four scenarios yields the following plots. Here, each curve represents different servers. We have suppressed the output for scenario 4 so that only the first 10 servers are shown. Observe that the curves are very close to the line $y = x$. This shows that the simulated R values are close to the theoretical R values, so our `on` and `off` functionality is working as desired. Here, we have done the tests for `max_time = 1000`.

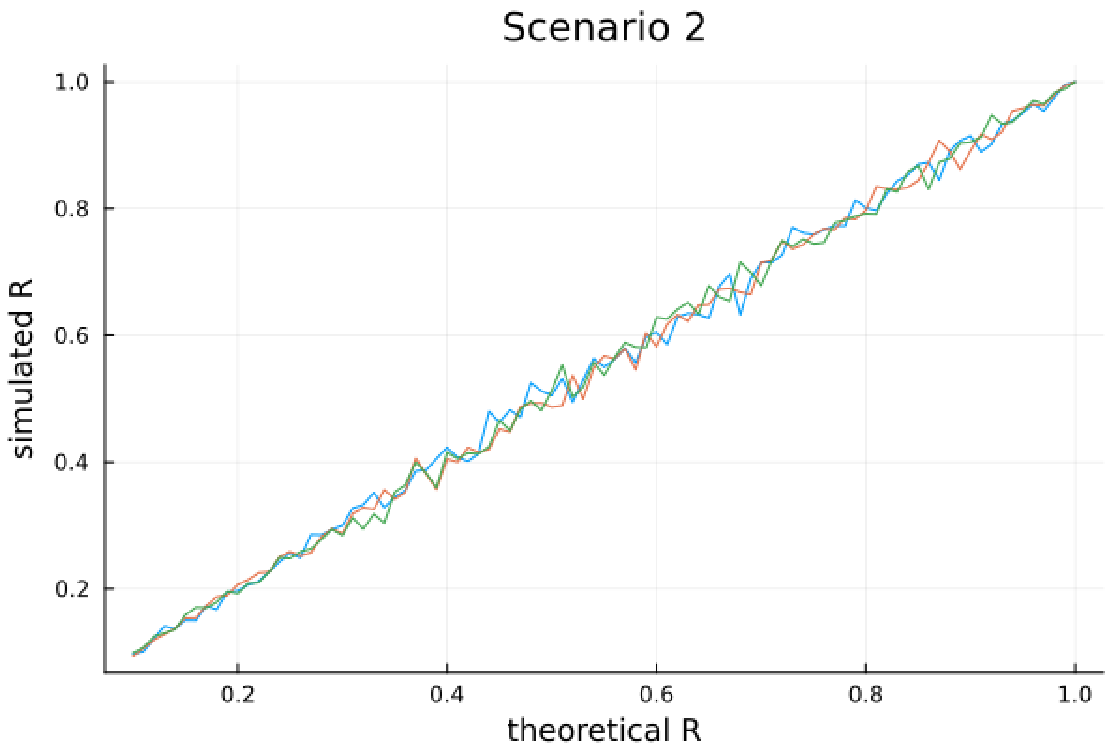
[52]:
```
load("img/task_3_test_3_scenario_1.png")
```

[52]:

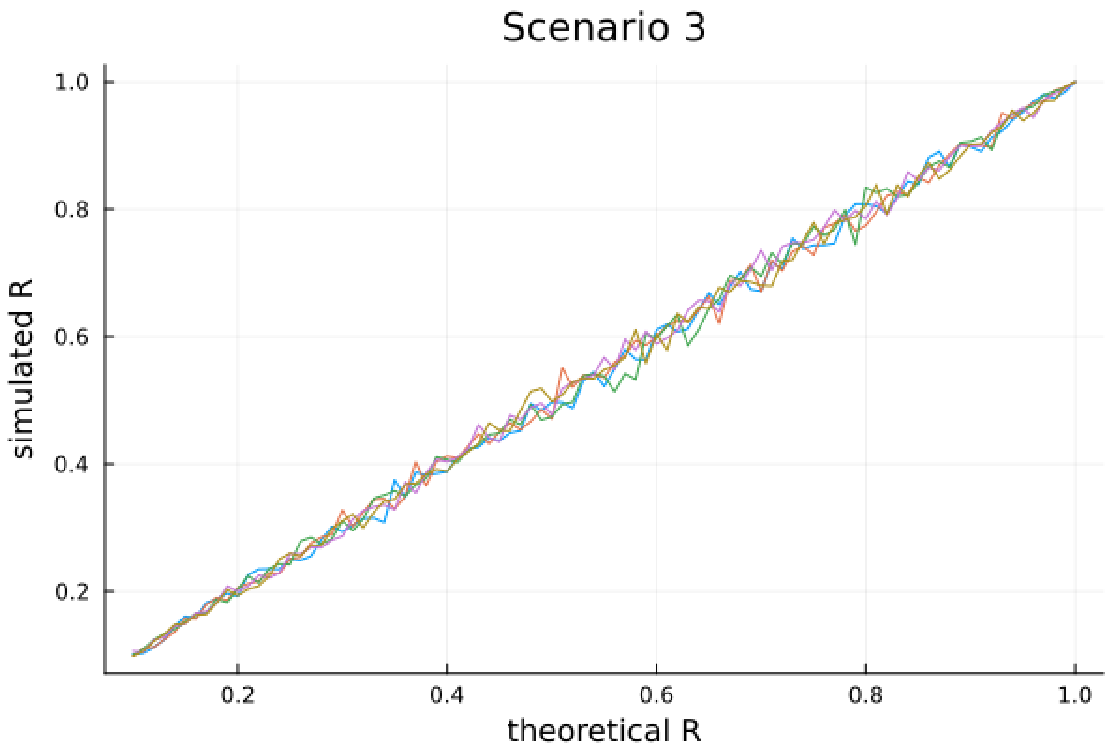


[53]:
```
load("img/task_3_test_3_scenario_2.png")
```

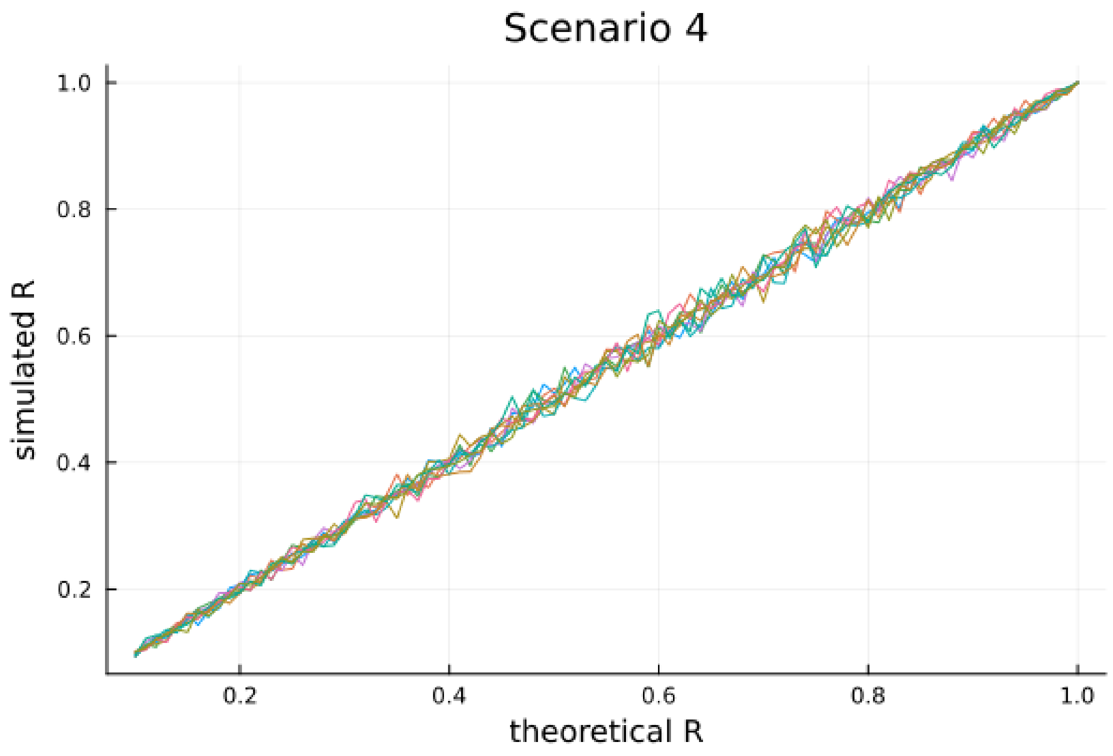
[53]:



[54]:
```
load("img/task_3_test_3_scenario_3.png")
```
[54]:



[55]:
```
load("img/task_3_test_3_scenario_4.png")
```
[55]:



5.4 Test 4

In this task, we plot the simulated mean total queue length against different values of R. Note that the arrival rates are also changed so that the scenarios remain stable. The following code can be found in `test/task_3_test_4.jl`.

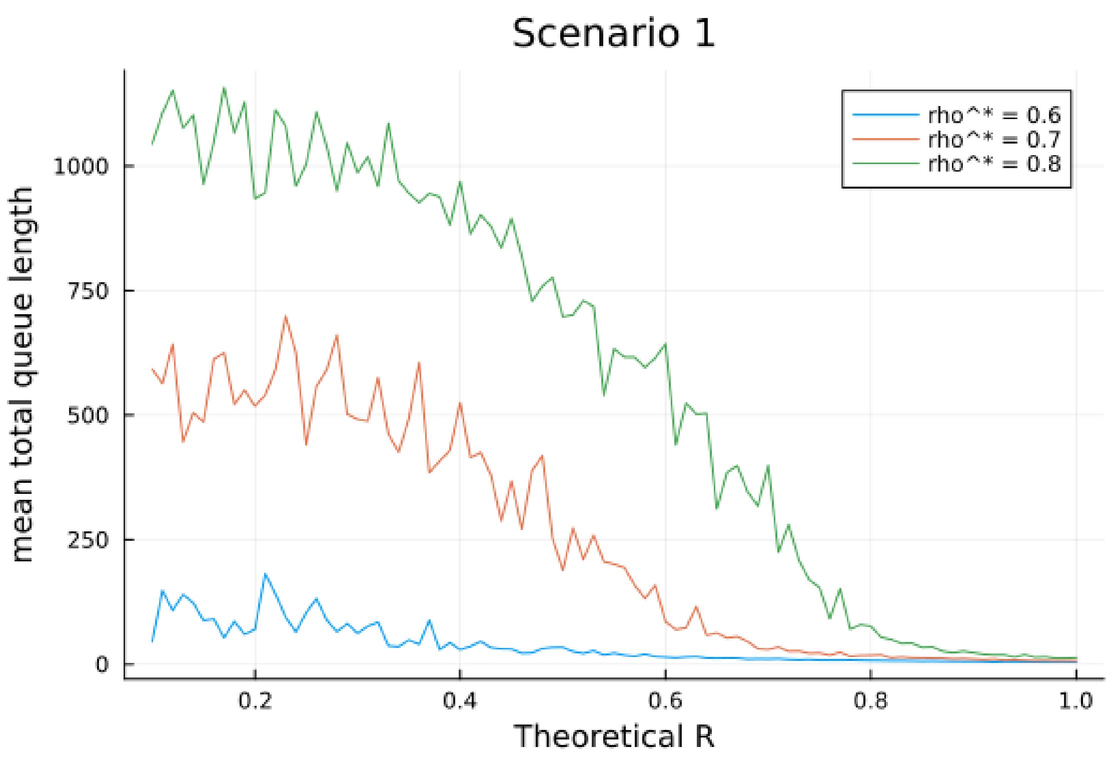
```
[ ]: """
     Returns a plot which shows the mean total queue lengths for different values of␣
      ↪R.
     """
     function plot_R_versus_mean_queue_length(parameters::NetworkParameters,␣
      ↪scenario_number::Int)
         Rs = 0.1:0.01:1
         max_time = 10000
         warm_up_time = 100

         data = []
         for rho_star in [0.6,0.7,0.8]
             mean_queue_lengths = Vector{Float64}(undef, length(Rs))
             for (i, R) in enumerate(Rs)
                 new_parameters = set_scenario(parameters, R = R,rho_star = rho_star)

                 riemann_sum = 0.0
```

```
            prev_total = 0.0
            prev_time = 0.0

            function record_integral(state::NetworkState, time::Float64)
                (time >= warm_up_time) && (riemann_sum += prev_total * (time -␣
↪prev_time))
                prev_total = sum(state.queues)
                prev_time = time
            end

            sim_net(new_parameters, max_time = max_time, warm_up_time =␣
↪warm_up_time,
                callback = record_integral)

            mean_queue_lengths[i] = riemann_sum / (max_time - warm_up_time)
        end
        push!(data, mean_queue_lengths)
    end
    plot(Rs, data, title = "Scenario $(scenario_number)",
    xlabel = "Theoretical R", ylabel = "mean total queue length",
    labels=["rho^* = 0.6" "rho^* = 0.7"  "rho^* = 0.8"])
end
```

The function above applied to the four scenarios yields the following plots. Observe that as R decreases, the mean total queue lengths increase. On each plot, we also consider different values of ρ^*.

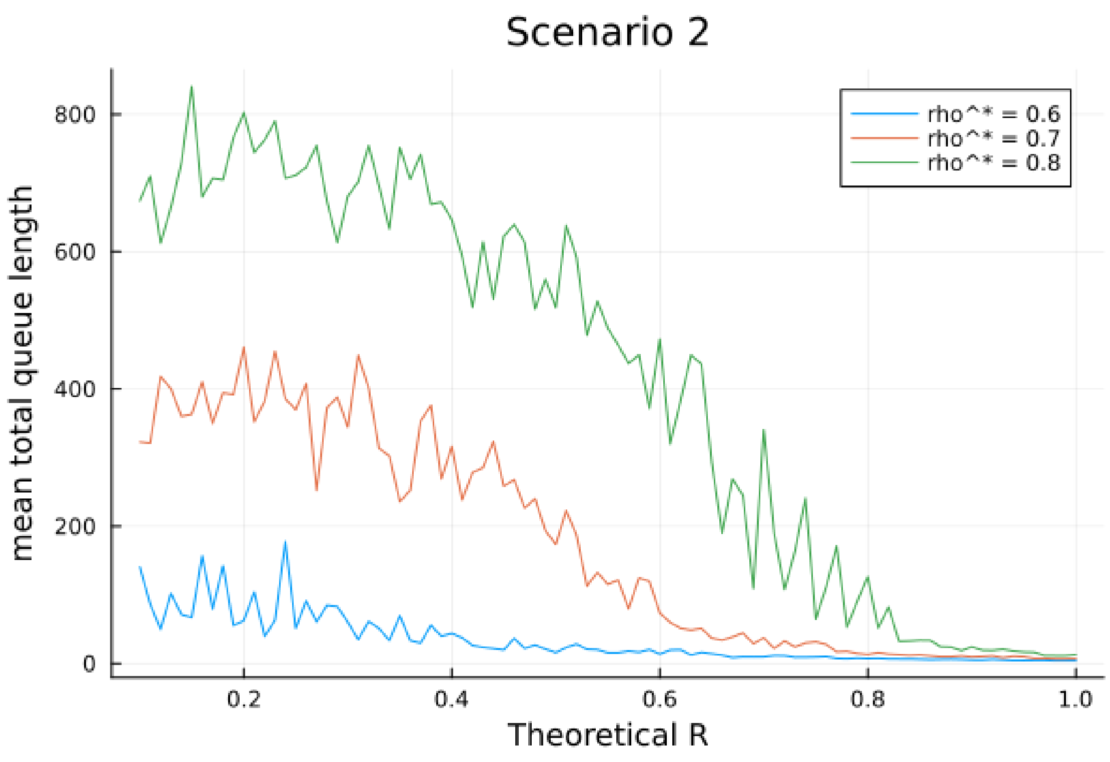
[57]:
```
load("img/task_3_test_4_scenario_1.png")
```

[57]:



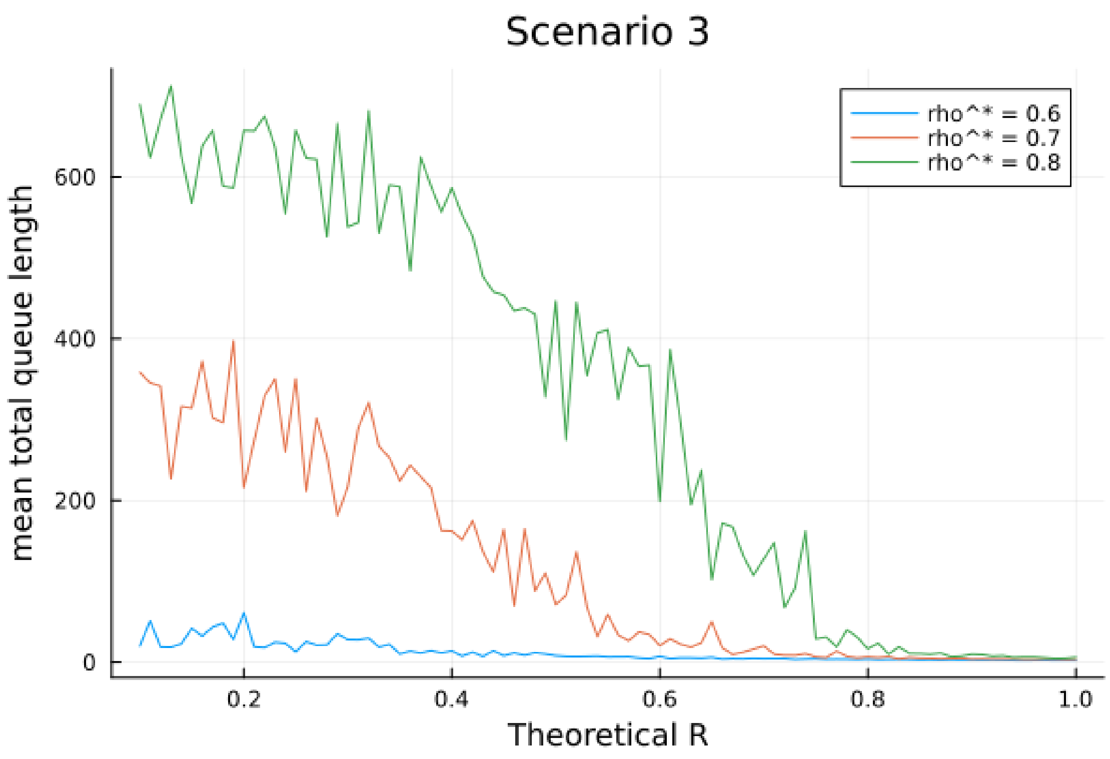
```
[58]: load("img/task_3_test_4_scenario_2.png")
```

[58]:



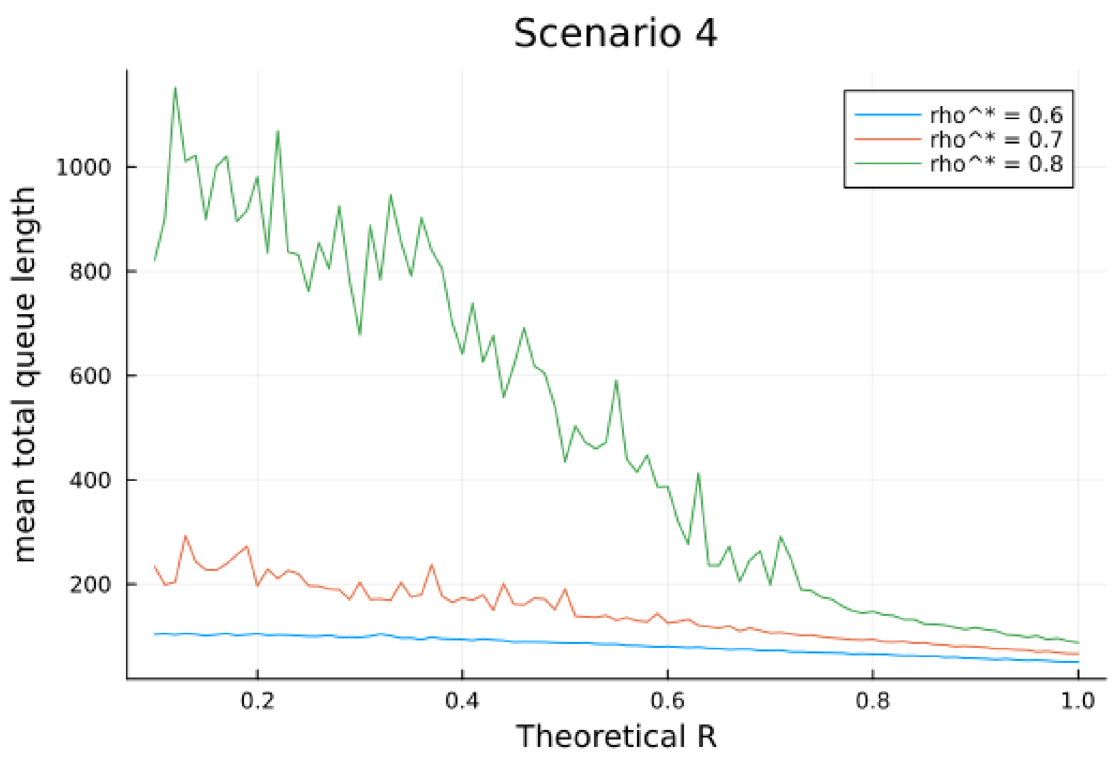
[59]:
```
load("img/task_3_test_4_scenario_3.png")
```

[59]:



[60]:
```
load("img/task_3_test_4_scenario_4.png")
```

[60]:



6 Task 4

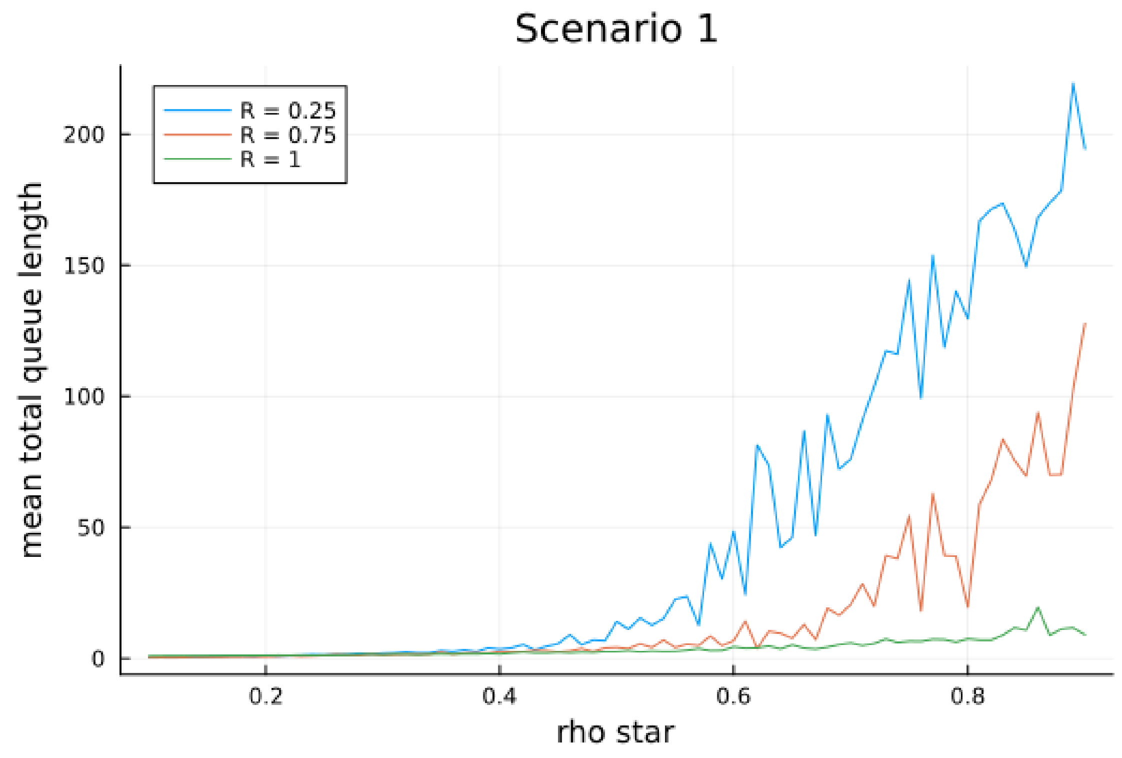
In this task, we plot mean total queue lengths against ρ^* for different values of c_s and different values of R. The first four plots have fixed $c_s = 0.5$, and the next four plots have fixed $R = 0.75$. The following code can be found in `test/task_4.jl`.

```
[ ]: """
Returns two plots p1 and p2.

The first plot p1 shows the simulated total mean queue length for different␣
  ↪values of
R and rho^*, where we fix c_s = 0.5.

The second plot p2 shows the simulated total mean queue length for different␣
  ↪values of
c_s and rho^*, where we fix R = 0.75.
"""
function plot_mean_queue_length_different_R_and_c_s(parameters::
  ↪NetworkParameters, scenario_number::Int)
    c_s_values = [0.1,0.5,1.0,2.0,4.0]
    R_values = [0.25, 0.75, 1.0]
    rho_stars = 0.1:0.01:0.9
    max_time = 10000
    warm_up_time = 100

    data = []
    for R in R_values
        simulation_results = Vector{Float64}(undef, length(rho_stars))
        for (i, r) in collect(enumerate(rho_stars))
            new_parameters = set_scenario(parameters, rho_star = r, c_s = 0.5,␣
↪R = R)

            riemann_sum = 0
            last_time = 0.0

            function record_integral(state::NetworkState, time::Float64)
                (time >= warm_up_time) && (riemann_sum += sum(state.queues) *␣
↪(time - last_time))
                last_time = time
            end

            sim_net(new_parameters, max_time = max_time, warm_up_time =␣
↪warm_up_time,
                callback = record_integral)

            simulation_results[i] = riemann_sum / (max_time - warm_up_time)
        end
        push!(data, simulation_results)
    end

    p1 = plot(rho_stars, data, title = "Scenario $(scenario_number)",
    xlabel = "rho star", ylabel = "mean total queue length",
    labels = ["R = 0.25" "R = 0.75" "R = 1"])

    data = []
    for c_s in c_s_values
        simulation_results = Vector{Float64}(undef, length(rho_stars))
        for (i, r) in collect(enumerate(rho_stars))
            new_parameters = set_scenario(parameters, rho_star = r, c_s = c_s,␣
↪R = 0.75)

            riemann_sum = 0.0
            prev_total = 0.0
            prev_time = 0.0

            function record_integral(state::NetworkState, time::Float64)
                (time >= warm_up_time) && (riemann_sum += prev_total * (time -␣
↪prev_time))
                prev_total = sum(state.queues)
                prev_time = time
            end

            sim_net(new_parameters, max_time = max_time, warm_up_time =␣
↪warm_up_time,
                callback = record_integral)

            simulation_results[i] = riemann_sum / (max_time - warm_up_time)
        end
        push!(data, simulation_results)
    end

    p2 = plot(rho_stars, data, title = "Scenario $(scenario_number)",
    xlabel = "rho star", ylabel = "mean total queue length",
    labels = ["c_s = 0.1" "c_s = 0.5" "c_s = 1" "c_s = 2" "c_s = 4" ])

    return p1, p2
end
```

Observe that as we increase c_s or decrease R, the curves shift up.

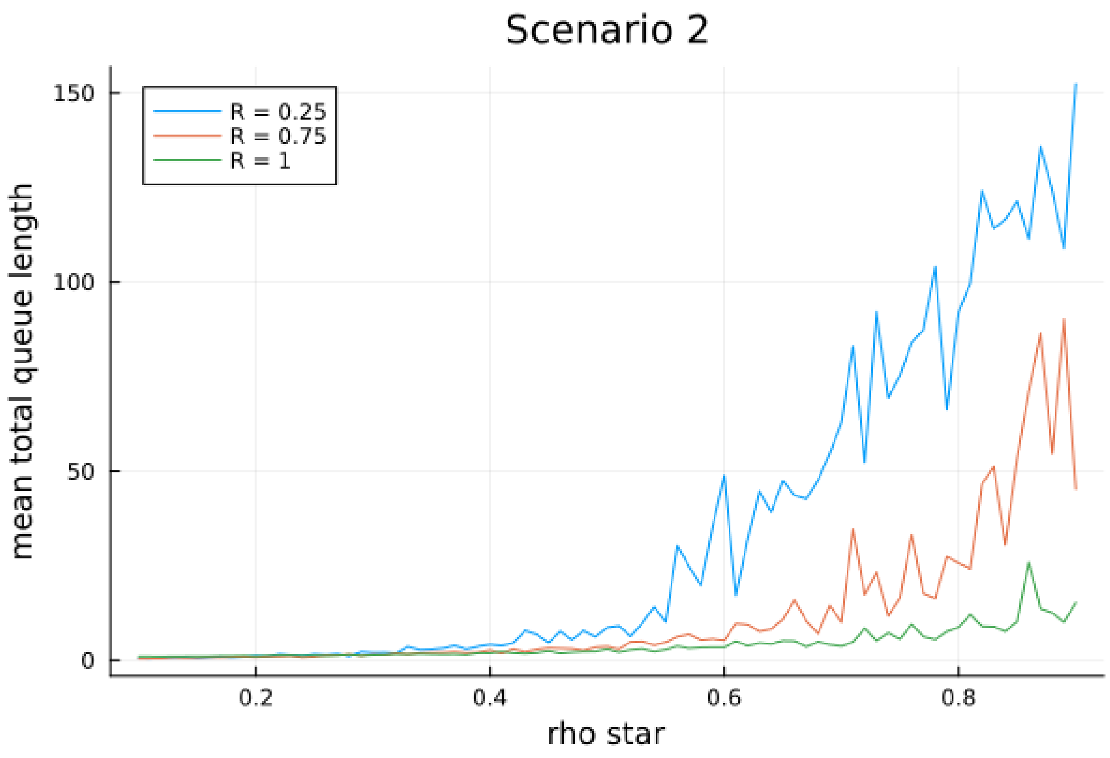
[61]:
```
load("img/task_4_scenario_1_different_R.png")
```

[61]:

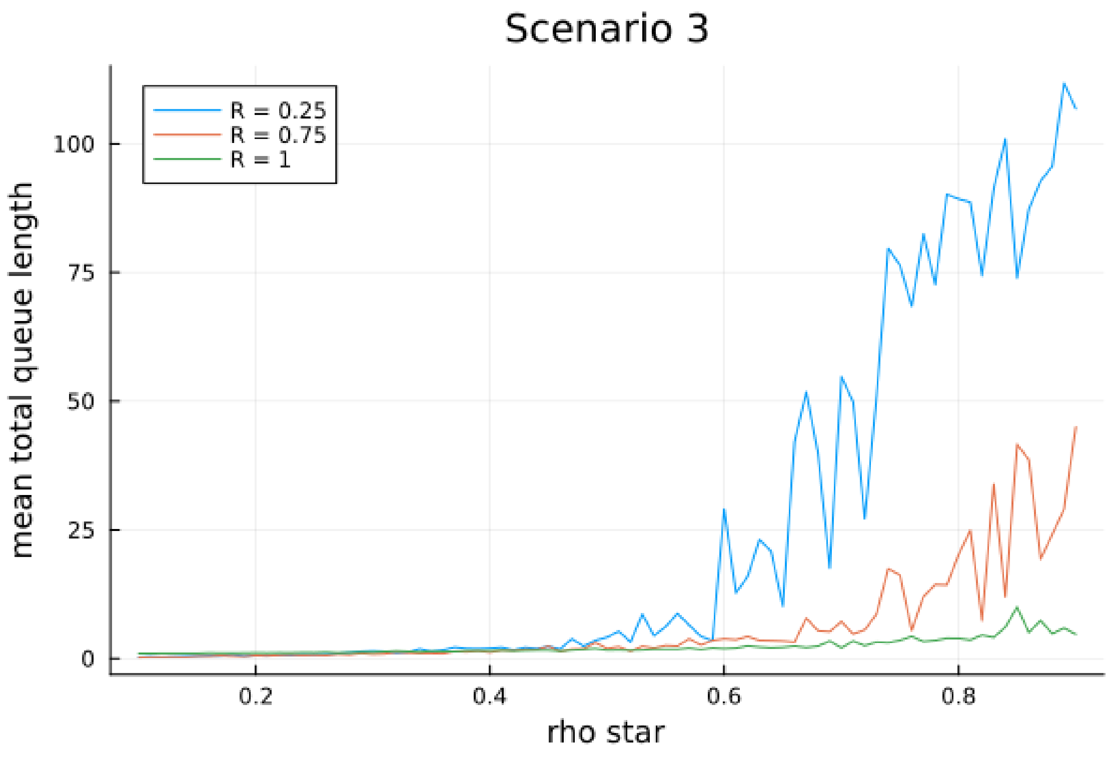


[62]:
```
load("img/task_4_scenario_2_different_R.png")
```

[62]:

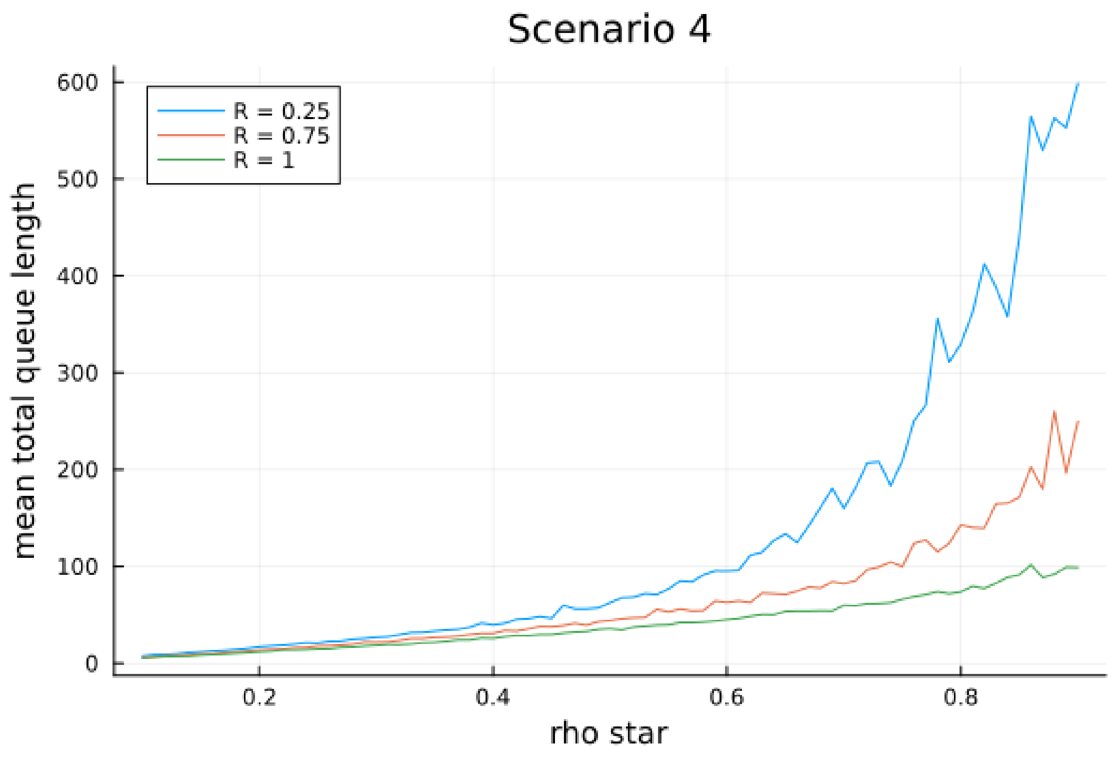


[63]:
```
load("img/task_4_scenario_3_different_R.png")
```
[63]:



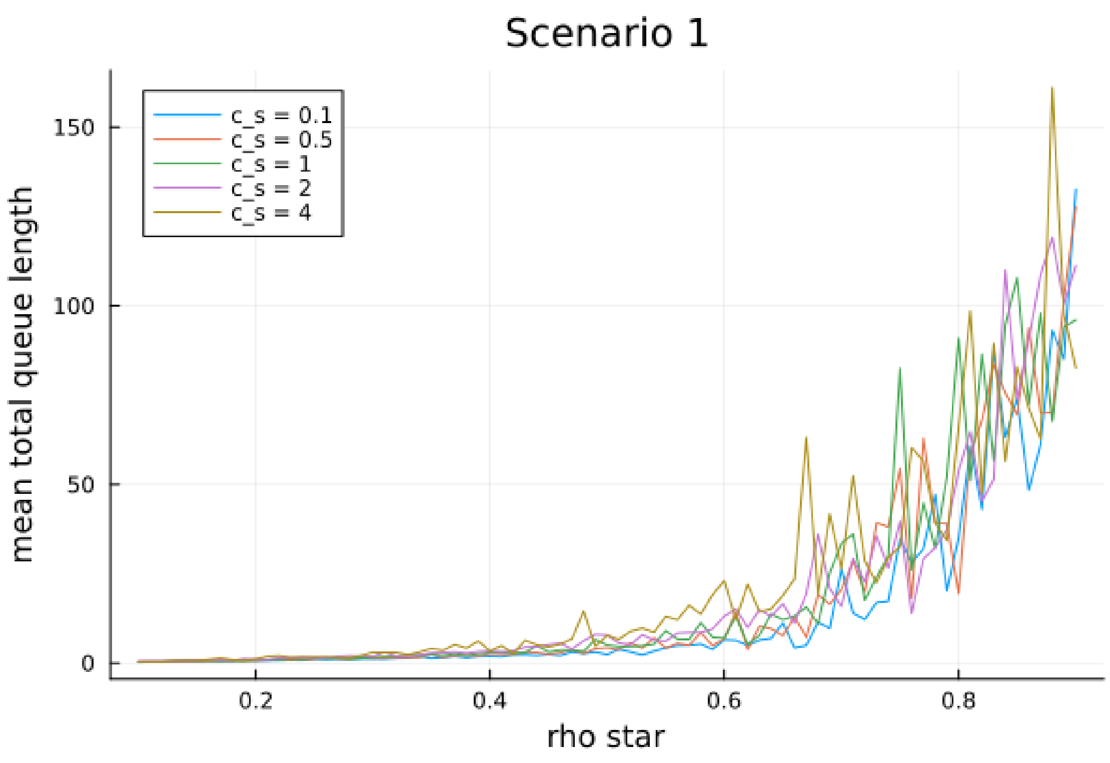
```
[64]: load("img/task_4_scenario_4_different_R.png")
```

[64]:



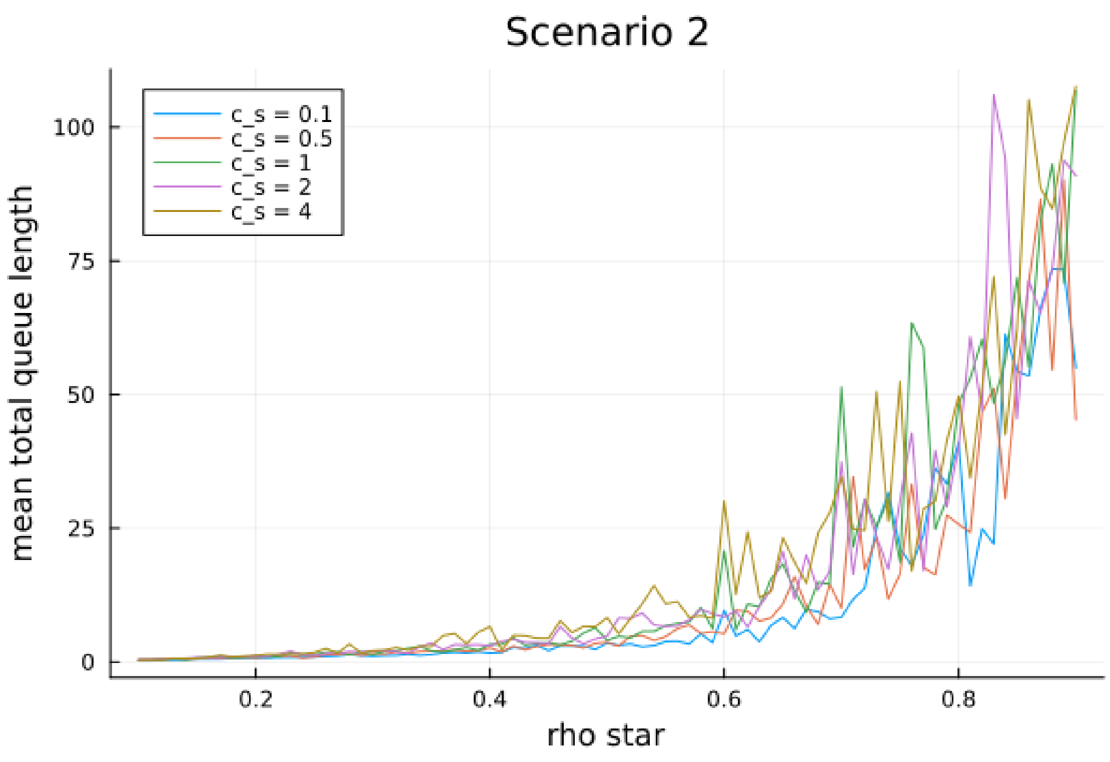
```
[65]: load("img/task_4_scenario_1_different_c_s.png")
```

[65]:



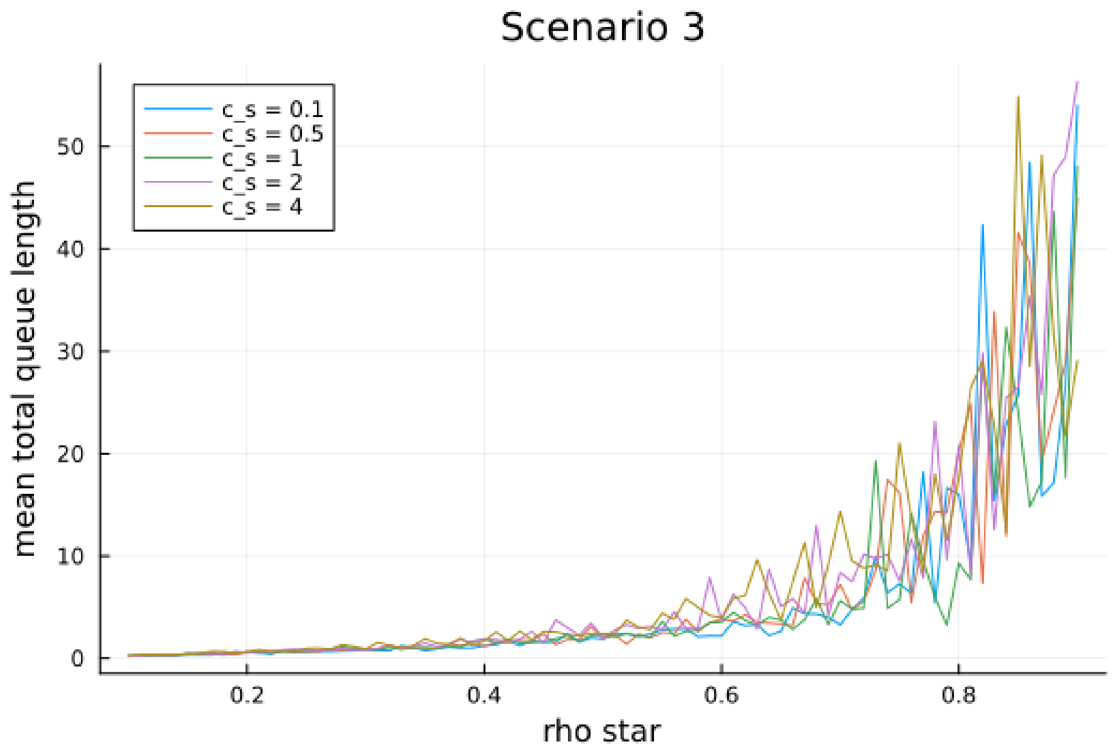
[66]:
```
load("img/task_4_scenario_2_different_c_s.png")
```

[66]:



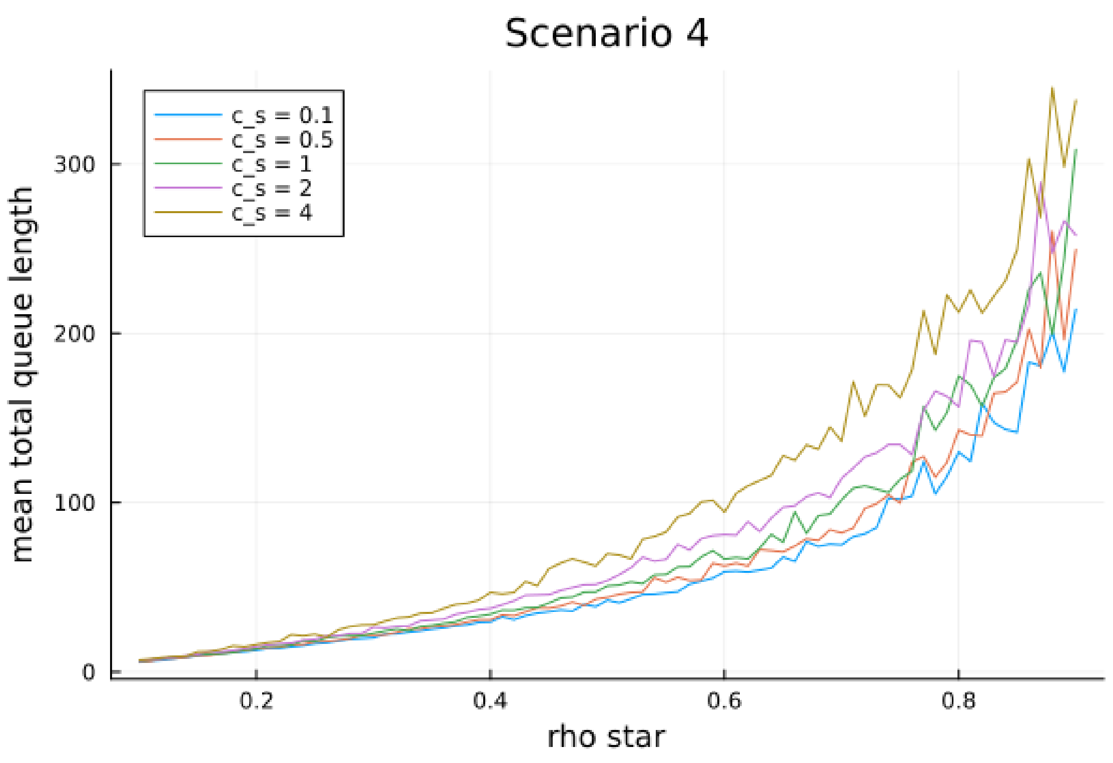
```
[67]: load("img/task_4_scenario_3_different_c_s.png")
```

[67]:



```
[68]: load("img/task_4_scenario_4_different_c_s.png")
```

[68]:



7 Task 5

First, we created a struct called `Customer` to track when customers enter and leave the network. The following can be found in `src/state.jl`.

[ ]:
```julia
"""
This mutable struct represents a customer. The arrival time and departure time
are stored. If the customer has not left the network, then departure time is
set to zero.
"""
mutable struct Customer
    arrival_time :: Float64
    departure_time :: Float64

    function Customer(arrival_time :: Float64)
        return new(arrival_time, 0)  # 0 is placeholder
    end
end
```

Next, we created a new subtype of `State`, called `NetworkStateCustomers`. This is identical to `NetworkState`, except that we keep track of the arrival times and departure times of customers. The following can be found in `src/state.jl`.

```
[ ]: """
     This mutable struct stores the state of a simulation, where we keep track
     of individual customers.

         - queues is a vector of queues which stores the customers lined up in each␣
       ↪queue.
         - arrivals stores the total number of (both external and internal)
             arrivals at each server.
         - server_status is a boolean vector representing whether the servers are on␣
       ↪(1)
             or off (0).
         - the qth term of additional_times represents the time that a job was␣
       ↪frozen
             because the qth server was off.
         - the qth term of last_off is a vector which stores the last time that the␣
       ↪qth server
             was off or started processing a new job.
     """
     mutable struct NetworkStateCustomers <: State
         queues::Vector{Deque{Customer}}
         arrivals::Vector{Int64}
         server_status::Vector{Bool}
         additional_times::Vector{Float64}
         last_off::Vector{Float64}
         parameters::NetworkParameters
         customers :: Set{Customer}

         function NetworkStateCustomers(parameters::NetworkParameters)
             L = parameters.L
             return new(
                 [Deque{Customer}() for _ in 1:L],
                 zeros(Int, L),
                 ones(Bool, L),
                 zeros(Float64, L),
                 zeros(Float64, L),
                 parameters,
                 Set{Customer}())
         end
     end
```

We added a new event called `CustomerArrivalEvent` so that we can store the arrival times of customers. We also changed rewrote `process_event` for `endOfServiceEvent` so that the departure time is recorded if the customer leaves the network. The following can be found in `src/event.jl`.

```
[ ]: """
     A person arriving at the qth server.
     """
```

```julia
struct CustomerArrivalEvent <: Event
    q :: Int
    customer :: Customer
end

"""
Processes an external arrival at the qth server.
"""
function process_event(time::Float64, state::NetworkStateCustomers, event::
 ↪CustomerArrivalEvent)
    q = event.q
    # check if customer is in state yet -- if not, add them
    # note that we only add customers to the set as they arrive in their first
    # queue so that the max_time state is accurate
    if !(event.customer in state.customers)
        push!(state.customers, event.customer)
    end
    push!(state.queues[q], event.customer)
    state.arrivals[q] += 1

    next_time = time + next_arrival_duration(state, q)
    new_timed_events = [
        TimedEvent(CustomerArrivalEvent(q, Customer(next_time)), next_time)
    ]

    if length(state.queues[q]) == 1
        # we just added the only person to the queue, so they can start being
        # served now. Add an end of service event.
        state.additional_times[q] = 0
        state.last_off[q] = time
        push!(new_timed_events,
            TimedEvent(EndOfServiceEvent(q), time +␣
 ↪next_service_duration(state, q)))
    end
    return new_timed_events
end

"""
If there were no breakdowns during the service of this job
(i.e. there is no additional time) then we process the end of a job at the qth␣
 ↪server.

If a breakdown occurred during service, then  we do nothing except spawn an
endOfServiceEvent which is processed after the additional time has lapsed.
"""
function process_event(time::Float64, state::NetworkStateCustomers, event::
 ↪EndOfServiceEvent)
```

```
    q = event.q
    new_timed_events = TimedEvent[]

    if state.additional_times[q] == 0
        served_customer = popfirst!(state.queues[q])

        if length(state.queues[q]) > 0
            # start serving the next customer
            push!(new_timed_events,
                TimedEvent(EndOfServiceEvent(q), time +␣
 ↪next_service_duration(state, q)))
        end

        next_q = next_location(state, q)
        if next_q <= state.parameters.L
            push!(state.queues[next_q], served_customer)
            state.arrivals[next_q] += 1

            if length(state.queues[next_q]) == 1
                state.last_off[next_q] = time
                state.additional_times[next_q] = 0
                push!(new_timed_events,
                    TimedEvent(EndOfServiceEvent(next_q), time +␣
 ↪next_service_duration(state, next_q)))
            end
        else
            # Note the time the customer exits the system
            served_customer.departure_time = time
        end
    else
        push!(new_timed_events, TimedEvent(EndOfServiceEvent(q), time + state.
 ↪additional_times[q]))
        state.additional_times[q] = 0
        state.last_off[q] = time
    end

    return new_timed_events
end
```

Finally, we rewrote the main simulation loop with some minor changes. The following can be found in src/simulation.jl.

```
[ ]: """
     Simulates a generalised unreliable Jackson network with given parameters.

     This simulation keeps track of individual customers.
     """
```

```
function sim_net_customers(parameters :: NetworkParameters;
        state = NetworkStateCustomers(parameters),
        max_time = 10^6, warm_up_time = 10^4, seed::Int64 = 42)

    Random.seed!(seed)

    timed_event_heap = BinaryMinHeap{TimedEvent}()

    push!(timed_event_heap, TimedEvent(EndSimEvent(), max_time))

    for q in 1:parameters.L
        if parameters.alpha_vector[q] > 0
            next_time = next_arrival_duration(state, q)
            push!(timed_event_heap,
                TimedEvent(CustomerArrivalEvent(q, Customer(next_time)),␣
 ↪next_time))
        end

        if parameters.gamma_1 > 0
            push!(timed_event_heap,
                TimedEvent(ServerOffEvent(q), next_off_duration(state)))
        end
    end

    time = 0.0

    while true
        timed_event = pop!(timed_event_heap)
        time = timed_event.time
        new_timed_events = process_event(time, state, timed_event.event)

        isa(timed_event.event, EndSimEvent) && break

        for nte in new_timed_events
            push!(timed_event_heap, nte)
        end
    end

    return state
end

end;
```

We now wish the estimate the first quartile, the median, and the third quartile of the sojourn times of customers. To do so, we wrote a function to compute these quantiles.

[ ]:
```julia
"""
Returns the first quantile, the median and the third quantile for the sojourn
time of a customer in the netowrk.
"""
function estimate_q1_median_q3_customer_time(
        parameters :: NetworkParameters, c_s :: Float64) :: Vector{Float64}

    rho_star = 0.8

    customers = sim_net_customers(
        set_scenario(parameters, rho_star = rho_star, c_s = c_s), max_time =␣
 ↪10000).customers
    data = [c.departure_time - c.arrival_time for c in customers if c.
 ↪departure_time > 0]

    return nquantile(data, 4)[2:4]
end
```

Running the code above for the four scenarios and $c_s \in \{0.5, 1.0, 2.0\}$ gives the following table:

```
Scenario 1:
             Q1          Median      Q3
c_s = 0.5    5.9376      8.878       13.1684
c_s = 1.0    9.1139      14.1961     20.7367
c_s = 2.0    12.0972     17.4686     24.2821

Scenario 2:
             Q1          Median      Q3
c_s = 0.5    6.9704      10.8881     17.0183
c_s = 1.0    11.1498     17.5495     28.178
c_s = 2.0    14.1851     24.7676     42.9931

Scenario 3:
             Q1          Median      Q3
c_s = 0.5    0.8826      3.0646      6.8706
c_s = 1.0    0.9301      3.9741      10.301
c_s = 2.0    0.9583      4.3373      11.065

Scenario 4:
             Q1          Median      Q3
c_s = 0.5    1.2479      2.7865      5.7953
c_s = 1.0    1.1588      3.0137      6.7133
c_s = 2.0    0.9716      3.341       8.3149
```

[ ]:
```

```