Decision Science: Programming Assignment

Autumn Semester 2024

Purpose

The purpose of this assessment is to create a fully functional simulation of a more complex system. Building simulations is one of the key objectives of this course. You will need to be able to do this in an industry or government job working in decision sciences. Building a simulation is also the best way to absorb and gain a deep understanding of the ideas and topics that are discussed in this course. Even if you are just managing simulations projects, you should have some experience in creating the simulation itself.

This is quite a large and difficult task, but the assessment will provide considerable structure.

Outcomes

This Task addresses the following Course Learning Outcomes:

- communicate how randomness and controlled variation can be used to model complex systems in a range of application domains such as industry, health, and transportation;
- create a model of a real-world problem specified in words and implement it as a discrete-event simulation;
- validate results from a discrete-event simulation.

Scenario

You will be simulating a passport (immigration) control queuing system at an airport. This is based on the queuing system for foreign travelers arriving at Qingdao airport.

Passengers get off their flight, walk to the passport checkpoint area and join a single **primary queue**. When they get near the front of this queue they are organised into several short **secondary queues**. Each secondary queue leads to a counter staffed by an attendant, who checks the passenger's passport and clears them to their journey.

Additional features. The airport uses some strategies to make this process more efficient. This includes:

1. extra counters open when the system is busy, and close when it is less busy
2. attendants regularly rotate so they can take breaks
3. passengers may be served at a counter not corresponding to their outside their allocated secondary queue in some circumstances

Assumptions

You should assume the following:

- all queues operates in a First in First Out (FIFO) basis
- there are 2 counters open at the start of the simulation
- Times: all times are independent and
  - The **interarrival time between flights** is exponential with an expected value of 45 minutes
  - The number of passengers on each flight who will pass through this passport control area is uniformly distributed between 15 and 25 passengers (similar to the number of foreign arrivals on a plane to Qingdao)
  - The **transit time** for a passenger to walk from the plane to the queueing system is uniformly distributed between 5 and 30 minutes
  - The **service time** for passengers once they reach a counter is Pareto distributed with a minimum time of 2 minutes and an expected value of 4 minutes

The following assumptions relate to the **additional features** of the system.

- **Additional feature 1.** Extra counters open when the system is busy, and close when it is less busy
  - Let n_{counters} be the number of counters currently open, and N be the total number of passengers in the queuing system (main queue, all secondary queues and in service). An extra counter will open if $N > 10 \times n_{\text{counters}}$. A counter will close if $N < 10 \times (n_{\text{counters}} - 2)$.
  - There are a minimum of 2 counters and a maximum of 5 counters
  - When a counter "closes" this mean that no more passengers are accepted into its secondary queue. Any passengers currently in the secondary queue will be served as usual.
- **Additional feature 2.** Attendants regularly rotate so they can take breaks
  - Some of the counter attendants rotate at an interval of 30 minutes, during which time service at the counter is paused for 1 minute. This means that the service time for any passenger at a rotating counter will increase by 1 minute.
  - For odd numbered counters, the first rotation is at 30 minutes from the start of the simulation (and then every $2 \times 30 = 60$ minutes)
  - For even numbered counters, the first rotation is at 60 minutes from the start of the simulation (and then every $2 \times 30 = 60$ minutes)
- **Additional feature 3.** Passengers may be served at a counter not corresponding to their allocated secondary queue in some circumstances.
  - A passenger at the front of their secondary queue, who has been in their secondary queue for more than 20 minutes will be served next at any available open counter. In the case of multiple passengers in this situation, priority is given to the passenger who has spent the longest waiting time in their secondary queue.
  - If an open counter is available and its secondary queue is empty, they will serve a passenger from another secondary queue. Priority is given to the passenger with the longest secondary queue waiting time.
- The airport is open 24 hours, 7 days a week.
  - The behaviour of the system is not influenced by the time of day.
- The airport has specified that they consider a single simulation running for one day of simulated time (1440 minutes) to be sufficient for statistical relevance.

Questions

The airport's management would like to plan for an anticipated increase in passenger numbers, as the number of arrivals using this passport queueing system is expected to double in the next five years. They would like to understand how best to manage their staff, if additional counters need to be built, and whether their are any aspects of the queuing system that can be fine-tuned. Based upon passenger feedback, passengers are concerned with total amount of time they must spend in this queuing system.

The airport's management wish to identify the minimum number of counters that will ensure the expected total time in the queuing system when the number of customers double is no worse than the current level.

Airport management is also concerned with the well-being of their attendants, so would like to understand how the frequency and length of their rotation impacts passenger experience.

Note. You do not have to answer these questions as part of this assignment, but this motivates the use of parameters so we can vary the assumptions of our model.

Your Task

This assessment is scaffolded into three parts:

- Part 1: Modelling (Module 2)
- Part 2: Programming (Module 3)
- Part 3: Verification and testing (Module 4)

The details of each part are outlined in later in this brief. It is suggested you work through each of these in order, although there may be some back and forth between Parts 2 & 3.

Requirements

Julia code (20%). Write a simulation code in Julia according to the specification provided in this document. There are a total of **80 marks** for this part of the assessment, broken down as follows:

- Basic functionality: code implements the basic functionality of this queuing system, as per the specification. **[40 marks]**
- Additional feature 1: code implements this feature correctly. **[12 marks]**
- Additional feature 2: code implements this feature correctly. **[12 marks]**
- Additional feature 3: code implements this feature correctly. **[12 marks]**
- Code style: submission meets general style guidelines for good Julia code. **[4 marks]**

All the above are detailed in the rubric at the end of this document. Here you are required to submit a single `.jl` file, which contains type definitions, data structures and functions to run the simulation.

- *Submission requirement:* Submit Julia file via Cloudcampus.
- *Submission requirement:* An `include` command on your file must **not** give an error, if it does give an you will get **0 marks** (since this prevents your code from being automatically tested).

If you need clarification about any of the above, please ask course staff during workshop, via email or via the course discussion board.

Part 1: Modelling

In this part, you will model the system that you will be implementing.

The system

You will be simulating a passport (immigration) control queuing system at an airport. This is based on the queuing system for foreign travelers arriving at Qingdao airport.

See page 1 ("Scenario") of this document for a detailed description of the system.

Tasks for Part 1

Before you commence coding you should perform a series of modelling tasks. These tasks will prepare you for Part 2, when you will implement a simulation in Julia. You should work through these tasks and complete the online milestone quiz. Note you are **not** required to submit written responses to these points.

1. Draw a schematic of the system. This can be a rough sketch on a piece of paper - this is a great way to get an initial understanding of the system.
2. Describe the state(s) of the system.

   Hint: What state details are needed to answer supermarket owner's questions?
3. Describe the number of entities in the system in relation to the state(s).

   Hint: A simple equation may be helpful in this description.
4. Assume there are four events in the simulation: passengers leaves the aircraft after landing (`FlightDisembarks`), passenger arrives at queue (`Arrival`), passenger departs queue (`Departure`) and attendants rotating for a break (for additional feature 2, `RotateAttendants`). For each event:
   - describe how each event changes the state of the system.
   - describe the new events that may be created as a result of this event.
5. Draw a state diagram, illustrating the possibly states of the system and how the state changes in response to the various events.
6. Draw a flow chart illustrating your simulation structure.

To be clear, you will **not** be asked to submit written responses to the above but your understanding of these points will be assessed via the milestone quiz (worth 5% of your course mark).

It is not possible to write a simulation code working through the above detail, so take your time here to get some clear ideas about **what** you need to code **before** you start Part 2.

Part 2: Programming

In this part, you will start to program the model.

Reminder

The system to be modelled is described in page 1 of this document ("Scenario")

Tasks for Part 2

Before starting the process of coding up your simulation, go back to your work from Part 1 and think about if there are any further modelling details needed here. In particular you might like to carefully think about how to model/implement the three "Additional features".

The main task here is to write code to implement a discrete-event simulation of the system.

- The code should follow the style of some other codes presented in this course. You might like to use these as a starting point, or perhaps find it easier to start from an empty file.
- The process you need to follow is outlined below under the heading **Specification**.

By the end of this part, you should have a working simulation that can output results.

In Part 3 you will test it and use it to create some data with a simulation harness.

As part of your assessment, you will be required to submit one `.jl` file with your code for implementing the discrete-event simulation.

Specification

This week you will start coding your simulation. A good deal of the structure of the code is provided.

- This is to help you! There is a lot written below, but by following it carefully you will get a big start towards developing your simulation.
- It ensure that everyone has a common starting point.
- It makes it easier to review and assess your progress by ensuring that everyone adopts the same basic structure for their implementation.

The last point is important. Your mark is be based on automated testing of your code. Hence you **must** set up your code in the manner given, paying particular attention to setting up the data types **exactly** as described below, otherwise these test will fail and you will lose marks.

Here are the required specification details:

1. **Filenames.** Your code must be included in a single `.jl` files. These should be named:

   - `AirportOverseasPassports.jl`

   This should contain all of your **data structures** and **functions**. You will be provided with a file to run this code `AirportOverseasPassports_run.jl` which will `include` the first file and run the simulation for a set of parameters.

2. You should use the standard packages that you have been using in this course.

   These include:

   - `DataStructures`
   - `Distributions`
   - `StableRNGs`
   - `CSV`

   You may wish to use a small set of additional packages such as `Dates` or `Printf`.

   Do not use any packages other than these or those that have been discussed in the course.

3. Your code must specify three data structures:

   ```
   abstract type Event end
   mutable struct Passenger
     ...
   ```

`mutable struct State ...`

Each will contain fields as required. The state structure should contain any queues or lists required, for instance, the event list. More detail about each follows.

4. The abstract type `Event` will have the following subtypes:

   - `FlightDisembarks` ... an aircraft lands and unloads passengers
   - `Arrival` ... a passenger arrives at the queuing system
   - `Departure` ... a passenger finish their passport check and leaves
   - `RotateAttendants` ... the counter attendants take a break

   Each subtype of `Event` must be a `mutable struct` and must contain the following fields (with the types indicated):

   - `FlightDisembarks`
     - `id` ... an integer valued event ID number (`Int64`)
     - `flight_id` ... the ID of the flight (`Int64`)
     - `no_passengers` ... the number of passengers from the flight that will be processed by the queuing system, or `nothing` if the flight has not yet unloaded (`Union{Nothing,Int64}`)
   - `Arrival`
     - `id` ... an integer valued event ID number (`Int64`)
     - `flight_id` ... the ID of the passenger's flight
     - `disembark_time` ... the time the passenger's flight was unloaded (`Float64`)
     - `passenger_id` ... an integer valued passenger ID number, or `nothing` if the event does not yet have a customer allocated (`Union{Nothing,Int64}`)
   - `Departure`
     - `id` ... an integer valued event ID number (`Int64`)
     - `passenger_id` ... the ID of the departing passenger (`Int64`)
     - `counter_id` ... the counter that serviced that serviced the passenger, or `nothing` if they have not yet been served (`Union{Nothing,Int64}`)
   - `RotateAttendants`
     - `id` ... an integer valued event ID number (`Int64`)
     - `odd_attendants` ... true if odd numbered counters are rotating, false if even numbered counters are rotating (`Bool`)

   You may optionally define constructor functions for each of these above subtypes (to initialise some fields with `nothing` or a particular value as appropriate).

5. The data structure `Passenger` should contain fields to record important event times in the lifetime of the passenger. These are

   - `id` ... an integer valued customer ID number (`Int64`)
   - `flight_id` ... the ID of the passenger's flight (`Int64`)
   - `disembark_time` ... the time the passenger's flight was unloaded (`Float64`)
   - `enter_primary_time` ... the time the passenger entered the primary queue (`Float64`)
   - `enter_secondary_time` ... the time the passenger entered the primary queue (`Union{Float64, Nothing}`)
   - `start_service_time` ... the time the passenger starts service (`Union{Float64, Nothing}`)
   - `end_service_time` ... the time the passenger ends service (`Union{Float64, Nothing}`)
   - `secondary_id` ... the ID of the passenger's allocated secondary queue (`Union{Int64, Nothing}`)
   - `counter_id` ... the ID of the counter the passenger was served by (`Union{Int64, Nothing}`)
   - `attendant_rotated` ... true if the attendant rotated during service, false if not (`Bool`)

   Initially unknown values above should be set to `nothing` when the `Passenger` is created. Similarly, `attendant_rotated` should be initialised as `false`.

6. The data structure `State` will contain an event list (priority queue), queues for all resources in the system and a few other details. These must be

   - `time` ... the current system time (`Float64`)
   - `event_list` ... the list of scheduled event (`PriorityQueue{Event,Float64}`, here priority is the time of the event)
   - `primary_queue` ... queue of passengers waiting in the primary queue (`Queue{Passenger}`)
   - `secondary_queues` ... vector of queues of passengers waiting in the secondary queues (`Vector{Queue{Passenger}}`)

- `in_service` ... vector of passengers currently being served, or `nothing` if there is no passenger at a given counter (`Vector{Union{Passenger,Nothing}}`)
- `n_entities` ... a counter of the number of entities in the simulation so far (`Int64`)
- `n_events` ... a counter of the number of events in the simulation so far (`Int64`)
- `n_flights` ... a counter of the number of flights that have unloaded passengers (`Int64`)
- `n_open_counters` ... the number of currently open counter (`Int64`)

The constructor function for `State` should initialise all queues with empty queues of the correct type, time to `0.0` and the counters to `0`.

7. **Parameters**

   Your code should have a data structure `struct Parameters` for passing parameters as given in the below table

parameter	type	description
seed	`Int64`	random seed for the simulation
min_counters	`Int64`	minimum number of counters
max_counters	`Int64`	maximum number of counters
secondary_queue_length	`Int64`	length of the secondary queues
mean_interflight	`Float64`	mean time between planes unloading
min_no_passengers	`Int64`	minimum number of passengers from plane to queue
max_no_passengers	`Int64`	maximum number of passengers from plane to queue
min_passenger_transit_time	`Float64`	minimum time to get from plane to queuing system
max_passenger_transit_time	`Float64`	minimum time to get from plane to queuing system
min_service_time	`Float64`	minimum time for passenger to complete service
expected_service_time	`Float64`	expected time for passenger to complete service
final_time	`Float64`	the final time for the simulation
attendant_rotation_interval	`Float64`	the time between attendant rotations
attendant_rotation_time	`Float64`	the time it takes for attendants to rotate
secondary_rush_time	`Float64`	waiting time for a passenger to go to the next available counter
helpful_attendants	`Bool`	counter will serve passenger from outside their secondary queue

   Note some of these parameters are important to implementing some of the additional features:

   - Additional feature 1: `min_counters` and `max_counters`; feature switched off if these are equal.
   - Additional feature 2: `attendant_rotation_interval` and `attendant_rotation_time`; feature can be switched off if the interval is set to be greater than the total simulation time.
   - Additional features 3: `secondary_rush_time` and `helpful_attendants`; feature switched off if `helpful_attendants` is `false`.

8. Random number generators

   Your code will use three random number generators. Store these as functions in a data structure `struct RandomNGs`. For convenience, also store here a function that returns the resolution time. The code to define this data structure is as follows

```
struct RandomNGs
    rng::StableRNGs.LehmerRNG
    interflight_time::Function
    no_passengers::Function
    transit_times::Function
    service_time::Function
end
```

   You should define a constructor function with

```
function RandomNGs(P::Parameters)
```

   that takes the parameters structure as input and returns the random number generator and the four functions. Initialise the random number generator as follows

```
rng = StableRNG( P.seed )
```

   The four functions should similarly use variables from the parameters structure, and be consistent with the modelling assumptions given earlier in this document.

9. Initialisation function

   Your code have an `initialise` function that takes as input the parameters of the system and returns an initial system state and creates the random number generators you are going to use.

   The initialisation function should also create a new system state and inject an initial `FlightDisembarks` at time 0.0. If you are implementing **additional feature 2** the function should add an initial `FlightDisembarks` at `t=P.attendant_rotation_interval` (this is 30 minutes in the example output). The function should return the system state and the random number structure.

```
function initialise(P::Parameters)
    R = RandomNGs(P)
    system = State(P)

    # add an arrival at time 0.0
    t0 = 0.0
    system.n_events += 1
    system.n_flights += 1
    enqueue!( system.event_list, FlightDisembarks(system.n_events,system.n_flights),t0)

    # add the first attendant rotation
    system.n_events += 1
    enqueue!( system.event_list, RotateAttendants(system.n_events,true),P.attendant_rotation_interval)

    return (system, R)
end
```

All you need to do here is copy and paste the above function into your code.

10. Update functions (overall)

    Your code must have a set of `update!` functions with signatures:

    `function update!( S::State, P::Parameters, R::RandomNGs, E::SomeEvent )`

    Each update function should process one of your event types so you will need one function per event type.

    Each `update!` function should modify the state `S` appropriately, including

    - update any state variables
    - allocate a passenger to the event (if appropriate)
    - add any new events created from this one to the event list
    - move entities between the various queues as appropriate (for example, primary to secondary).

    These functions must not have side effects. That is, they should not write out any information to files, or interact with global variables. However, your functions may throw an error if the input is invalid.

    Some additional details of the behaviour of these function can be inferred from the provided example output files.

11. Update function (for `Arrival`)

    This function should process an `Arrival` event in a manner consistent with the system description and assumptions.

    As in example codes from the course, you may find it helpful to write an additional function to use here to move a customers around the various queues or into service (common to this and the `Departure` update function).

12. Update function (`Departure`)

    This function should process a `Departure` event in a manner consistent with the system description and assumptions.

13. Update function (`FlightDisembarks`)

    This function should process a `FlightDisembarks` event in a manner consistent with the system description and assumptions.

14. Update function (`RotateAttendant`)

    This function should process a `RotateAttendant` event in a manner consistent with the system description and assumptions.

15. State-based and entity-based output files

    Your code must output two CSV files. The files should both begin with some metadata and parameters (you can include comments preceded with a `#`). These should be stored in subfolders named for the random seed and some parameter values.

    - The first file `state.csv` should contain a time-ordered list of all events that are processed in the simulation.

      This should be written from the point of view **after** the event. For instance, you should report the system that an arriving passenger sees immediately after their arrival.

      The CSV file should have columns titled:

```
event_ID,time,event,upcoming_arrivals,primary_length,secondary_lengths,in_service,n_total,n_open_counters
```

    Here `upcoming_arrivals` should be a count of `Arrival` events in the `event_list`, and `n_total` is the total number of people checking out, including those in service and those with a problem. See the provided example files for formatting of the vectors in `secondary_lengths` and `in_service`.

    - The second file `entities.csv` should contain a list of all entities that have completed service. The CSV file should have columns titled (note the header should be one line as in the example output):

```
```
passenger_id,flight_id,disembark_time,enter_primary,enter_secondary,start_service,end_service,secondary_id,...
counter_id,attendant_rotated
```
```

    You will need to write and construct these CSV files line by line. This is most easily done with `println` statements, but you may use a package like `CSV` or `Printf` if you prefer.

16. Run function(s)

    You code should contain a top-level function `run_checkout_sim` with the signature

    `run_checkout_sim(P::Parameters)`

    This function should take the parameters structure as its input. It should then

    - initialise the system state,
    - create the output files (and subfolders)
    - write metadata, parameters and the header to the output files
    - **run the simulation** by calling the `run!` function (see below)
    - close the output files

    The `run!` should have the signature

    `run!(system::State, P::Parameters, R::RandomNGs, fid_state::IO, fid_entities::IO)`

    This function should run the main simulation loop up to the final time given in the `Parameters`. It should write output to the `state.csv` file for all events and to the `entities.csv` for `Departure` events only. It should also update the system time **before** the `update!` function is called.

17. Code style

    Your code must be written with good style. See Julia's style guidelines for further information.

    In particular:

    - use comments (in English) efficiently and effectively

    - store commonly used code as functions

Further Julia hints on `PriorityQueue`

When a problem occurs, the depature time of the customer will be extended. That is, you need to change the time, i.e. the priority, of the corresponding departure event.

You can modify the priority of an object in a priority queue in Julia as follows:

```
using DataStructures
pq = PriorityQueue()
pq["a"] = 10; pq["b"] = 5; pq["c"] = 7;
pq
```

Output:

```
PriorityQueue{Any, Any, Base.Order.ForwardOrdering} with 3 entries:
"b" => 5
"c" => 7
"a" => 10
```

To change the priority of an object:

```
pq["a"] = 0 # change the priority of "a"
pq
```

Output:

```
PriorityQueue{Any, Any, Base.Order.ForwardOrdering} with 2 entries:
"a" => 0
"b" => 5
"c" => 7
```

Note that the order of the items in the queue has now changed. However, be careful here since the above approch would not update the time stored in the `Event` structure.

Additionally, you may find it useful to iterate over the keys of a `PriorityQueue` as follows (continuing from the above code):

```
for k in keys(pq)
    if k == "c"
        pq[k] = 3
    end
end
pq
```

Output:

```
PriorityQueue{Any, Any, Base.Order.ForwardOrdering} with 2 entries:
"a" => 0
"c" => 3
"b" => 5
```

This last piece of code may seem overly complex for this example, but can be adapted for the purposes here. Additionally, you may find it helpful to use the `dequeue_pair!`, `peek` and `delete!` functions for priority queues. Try using these on the above example as follows:

```
key, priority = dequeue_pair!(pq) # dequeue, return key and priority
peek(pq) # look at the top of the queue, without removing
delete(pq, "c") # delete an item with a given key
```

Part 3: Verification and testing

Note that no marks are directly associated with the activities described below. This is all about testing your code thoroughly to ensure it works as expected!

In this part, you will test your implementation to make sure it can output results and is consistent with some provided output.

Tasks for Part 3

Although you might not need to modify the code in `AirportOverseasPassports.jl`, you may need to modify it in response to bugs found in testing.

The main tasks will be to verify that your code works correctly (as described in Module 3: Verification), and to construct a small simulation harness in which to run a set of comparison simulations (as described in Module 4: Tools to Automate Simulation). Following these steps will help to detect any issues/bugs and ensure you get the maximum number of marks for the code you have written.

1. Test and verify your code.

   Compare the output of your code to the provided sample outputs:

   - `state_all.csv` and `entities_all.csv` ... output for the simulation with all additional features implemented (parameters in metadata)
   - state_basic.csvandentities_basic.csv' ... output for simulation with no additional features (parameters in metadata)

   If you have implemented your code exactly as specified, and used the same seed and random number generation, your output should look very, very similar (the only differences should be in details such as numbers of decimal points, white-spacing or ID numbers).

   However, your code may result in some differences. Some are important and others less so. You need to be quite analytical to understand which. That is, what differences occur because of a minor change in the order of actions, and what differences are caused by bugs. If there are differences, you should create some of your own tests to understand what is different from the code that produced the above results.

2. Create a test harness that will run your code for 100 different seed values ranging from 1–100. This could simply be a script that looped over these values.

   This will create 100 versions of the simulation, so by selectively examining the output and/or computing summary statistics you can get a better sense of how well your simulation functions. In particular, changing the seed can reveal "edge cases", that is a situation that occurs rarely (and might not have been in the sample output), which may lead to an error in your code.

   Additionally, from these outputs, you could perform some statistical analysis to inform the airport's management about their questions. Note you **do not need to answer these questions** as part of this assessment (that process will be part of the Project Report for a different problem).

Detailed rubric

You will be assessed on the following components of your work. **[80 marks total]**

- Basic functionality: code implements the basic functionality of this queuing system, as per the specification. **[40 marks]**
- Additional feature 1: code implements this feature correctly. **[12 marks]**
- Additional feature 2: code implements this feature correctly. **[12 marks]**
- Additional feature 3: code implements this feature correctly. **[12 marks]**
- Code style: submission meets general style guidelines for good Julia code. **[4 marks]**

The marking scheme for each component is given in detail below.

Basic functionality

Criteria	**Marks**
1 Define all data structures	4
2 Random number generators (service times)	4
3 Random number generators (all other RNGs)	4
4 Run and write functions	4
5 `update!` function (`Arrival`)	4
6 `update!` function (`Departure`)	4
7 `update!` function (`FlightDisembarks`)	4
8 State CSV (setup + initialisation)	4
9 State CSV (event behaviour correct)	4
10 Entity CSV (setup + behaviour correct)	4

Additional feature 1:

Criteria	**Marks**
11 Additional counters open per specification	4
12 Counters close per specification	4
13 Instrumented correctly in state and entity outputs	4

Additional feature 2:

Criteria	**Marks**
14 `update!` function (`RotateAttendants`)	4
15 Service times adjusted per specification	4
16 Instrumented correctly in state and entity outputs	4

Additional feature 3:

Criteria	**Marks**
17 Promote passenger to next server if long waiting	4
18 Passenger can be served by different counter	4
19 Instrumented correctly in state and entity outputs	4

Code style:

Criteria	**Marks**
Use of comments and functions per style guidelines	4