Decision Science: Programming Assignment

Haide College, Autumn Semester 2023

Due: 5 November 2023, 23:59

Weight: 30% course mark

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 self-serve supermarket checkout.

The supermarket checkout system currently consists of a single queue, four self-service registers and a single attendant. Customers who have selected their items in the supermarket will join the queue and wait for a register to become available. The customer at the front of the queue will then proceed to one of the registers and complete their purchase.

Sometimes there are problems with the order, and they will need to call the attendant before continuing with their purchases.

Assumptions

You should assume the following:

- the customer queue operates in a First in First Out (FIFO) basis
- Times: all times are independent and
  - The interarrival times for customers are independent and exponential with a mean of one minute
  - The service times for customers at a self-service register are independent and exponential with a mean of two minutes.
  - The time between problems is independent and exponential with a mean of one minute
  - When there is a problem the time for the attendant to resolve the issue is deterministic with a time of 30 seconds.
- The attendant may only deal with one problem at a time. If multiple problems occur, the attendant deals with them in a FIFO manner.
  - When a problem occurs, if may happen to any of the customers at the checkouts with equal probability.
  - If there are no customers at the checkouts then no problem can occur.
- The supermarket is open 24 hours, 7 days a week.

  - There is always exactly one attendant at the self-service checkouts.
  - The behaviour of the system is not influenced by the time of day.
- The client 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 supermarket's management is planning a major promotional campaign hoping to double the number of customers and wishes to know how many additional self-service registers to install. Based upon customer feedback, customers are concerned with total amount of time they must spend in the checkout system.

The supermarkets management wish to identify the minimum number of self-service registers that will ensure the expected total time in the checkout system when the number of customers double is no worse than the current level.

Supermarket management is also concerned with the amount of time spent waiting for the attendant. Although they have no current plans to change the number of attendants, they would like to know how much time customers spend waiting for the attendant on average.

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. Work through each of these in order.

Requirements

There are a total of **60 marks** for this assessment.

You will be assessed on three components of this work:

- Component 1: Conceptual model - your ability to formulate the model. **[10 marks]**
- Component 2: Functionality — your ability to write code to create a simulation, and ensure the sub-components of it work. **[20 marks]**
- Component 3: Programming style — your ability to write your program according to the specifications and general style guidelines for good code. **[30 marks]**

You are required to submit:

- a PDF document showing your schematic, state diagram, flow chart and any other documentation you created.
- two `.jl` files with your code for implementing the discrete-event simulation.
- a pair of data files (an entities and a state file) produced from your simulation with `seed=1` and where the simulation was stopped at time `T=525960.0`.

Consult the assessment rubric when preparing your submission.

You are welcome to ask questions of course staff at any time.

Grading Criteria

This assessment is worth 30% of your overall grade. Refer to the attached rubric for detailed information on the grading criteria for this assessment.

Appendix 1 - Part 1: Modelling

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

The system

You will be simulating a self-service checkout system such as you may encounter in many supermarkets.

There is a single queue in which customers wait to gain access to one of several self-service checkout machines. For the initial system there are four self-service checkout machines, you will later modify the system to increasing the number of self-service machines so as evaluate alternative options.

There is a single attendant who resolves problems that customers experience in the checkout process.

Tasks for Part 1

In Part 2 of this assessment you will write code to simulate the system.

Before you commence coding you should perform a series of modelling tasks. These tasks will prepare you for Part 2. Some of these tasks will be assigned marks but some will be part of larger tasks assigned marks in Parts 2 and 3.

1. Draw a schematic of the system. (1 marks)
2. Describe the state(s) of the system. (1 marks)

   Hint: What state details are needed to answer supermarket owner's questions?
3. Describe the number entities in the system in relation to the state(s). (1 mark)

   Hint: A simple equation may be helpful in this description.
4. There are four event in this model. 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.

   (2 marks)
5. Draw a state diagram, illustrating the possibly states of the system and how the state changes in response to the various events. (1 mark)
6. Draw a flow chart illustrating your simulation structure. (3 marks)

As part of your assessment, you will be required to submit a PDF document showing your schematic, flow chart and responses to these tasks.

Hint: You can create flow charts with the free diagram drawing tool draw.io, which has an extension available for VS Code. Otherwise, there are many other tools available for drawing connected series of boxes so find a tool you like. Also, your boxes and links don't have to look exactly like those in the course materials, but they have to be: (i) clear and readable, and (ii) consistent.

Appendix 2 - Part 2: Programming

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

Reminder

The system to be modelled is described in Part 1.

Tasks for Part 2

1. Refine your schematic of the system. (see Part 1)
2. Refine your state-diagram of the system. (see Part 1)
3. Write code to implement a discrete-event simulation of the system.
    - The code will follow the style of the code presented in this course. Use these exammple codes as a starting point, but make sure to customise it to this problem or you will get zero marks. **[30 marks]**
    - The process you need to follow is outlined below under the heading **Specification**. **[30 marks]**

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 a PDF document showing your schematic and state diagram, along with the flow chart from Part 1 and responses to these tasks. You will also be required to submit two .jl files with your code for implementing the discrete-event simulation from Task 3 above.

Specification [30 marks]

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. Part of your mark may be based on automated testing of your code. Hence you **must** set up your code in the manner given. Otherwise, you may lose marks for reasons that could be fixed with little effort.

Here are the required specification details:

1. Your code must be included in two stand-alone `.jl` files. These should be named:
    - `checkout_simulation.jl`
    - `checkout_simulation_run.jl`

    The first file `checkout_simulation.jl` should contain all of your **data structures** and **functions**.

    The second file `checkout_simulation_run.jl` should start with an `incude("checkout_simulation.jl")` command. It should then set the simulation parameters to the values given in the specification and run the simulation for `seed=1`.

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

    These include:

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

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.

[1 mark]

3. Your code must specify three data structures:

```
abstract type Event end
mutable struct Customer ...
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:

   - `Arrival` ... a customer arrives
   - `Departure` ... a customer departs
   - `Problem` ... a customer has a problem at checkout
   - `Resolved` ... the problem is resolved by the attendant

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

   - `id` ... an integer valued event ID number (`Int64`)
   - `time` ... the time of the event (`Float64`)
   - `customer_id` ... an integer valued customer ID number, or `nothing` if the event does not yet have a customer allocated (`Union{Nothing,Int64}`)

   Each subtype of `Event` must have a constructor function of the form

   `SomeEvent(time,id)=SomeEvent(time,id,nothing)`

   [2 marks]

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

   - `id` ... an integer valued customer ID number (`Int64`)
   - `arrival_time` ... the customer's arrival time (`Float64`)
   - `start_service` ... time the customer starts checking out (`Float64`)
   - `end_service` ... time the customer finishes checking out (`Float64`)
   - `current_problem_start` ... start time of the current problem (if any), otherwise `Inf` (`Float64`)
   - `problems` ... the number of problems a customer has had (`Int64`)

   The constructor function for `Customer` should be of the form `Customer(id, arrival_time)` with all other time fields initialised to `Inf` and the number of problems initialised to 0.

   [1 mark]

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_queue` ... the event queue (`PriorityQueue{Event,Float64}`, here priority is the time of the event)
   - `waiting_queue` ... queue of customers waiting to check out (`Queue{Customer}`)
   - `in_service` ... queue of customers currently checking out (`PriorityQueue{Customer,Float64}`, here priority is the expected time to end service)
   - `problem_queue` ... queue of customer with a problem (`Queue{Customer}`)
   - `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`)

   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`.

   [1 marks]

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
n_checkouts	`Int64`	number of checkouts in the supermarket
mean_interarrival	`Float64`	mean time between arrivals at the checkout queue
mean_service	`Float64`	mean service time
mean_interproblem	`Float64`	mean time between problems
mean_resolution	`Float64`	mean time for resolution
final_time	`Float64`	final time for the simulation

   [1 mark]

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
    interarrival_time::Function
    service_time::Function
    interproblem_time::Function
    resolution_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.

   [1 mark]

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 arrival at time 0.0 and initial problem at time 4.0 minutes. 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
    enqueue!( system.event_queue, Arrival(system.n_events,t0),t0)

    # add a problem at time 4.0
    t1 = 4.0
```

```
    system.n_events += 1
    enqueue!( system.event_queue, Problem(system.n_events, t1 ), t1 )

    return (system, R)
end
```

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

[1 mark]

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 the time
    - update any other state variables
    - allocate a customer 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, into service).
      - **Hint:** customers in the checkout process should be in either the `in_service` queue or the `problem_queue` but not both at the same time.

    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.

    Each function should return the customer entity that corresponds to the event being processed.

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

    [1 mark]

11. Update function (for `Arrival`)

    This function should proccess 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 customer to the checkout (common to this and the `Departure` update function).

    [2 marks]

12. Update function (`Departure`)

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

    [2 marks]

13. Update function (`Problem`)

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

    When a problem event occurs, your code should randomly allocate the problem to one of the in-service customers. This should be done with the following code:

    `problem_customer = rand(R.rng,keys(system.in_service))`

If no customers are at the checkout (or all customers are in the `problem_queue`), then the next `Problem` should be added a short time after the next event in the `event_queue` (an `Arrival` or `Resolved`). It may help to use the Julia function `eps()`, which returns a "machine epsilon" (that is a very small number).

As in example codes from the course, you may find it helpful to write an additional function to have a problem customer be helped by the attendant (common to this and the `Resolved` upadate function).

[4 marks]

14. Update function (`Resolved`)

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

    [2 marks]

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 customer sees immediately after their arrival.

    The CSV file should have columns titled:

    ```
    event_ID,customer_ID,time,event,n_waiting,n_checkout,n_problems
    ```

    - The second file `entities.csv` should contain a list of all entities that have completed service. The CSV file should have columns titled:

      ```
      customer_ID,arrival_time,service_time,departure_time,no_problems
      ```

    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.

    [2 marks]

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.

    [2 marks]

17. Code style

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

    In particular:

    - choose meaningful variable names and avoid global variables wherever possible **[1 mark]**
    - use comments efficiently and effectively **[3 marks]**
    - use white space well **[1 mark]**
    - store commonly used code as functions **[1 mark]**

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

Appendix 3 - Part 3: Verification and testing

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

In this part, you will test your implementation and make sure it can output results.

Reminder

The system to be modelled is described in Part 1 and builds on the code written in Part 2.

As part of your assessment, you will be required to submit:

- a PDF document showing your schematic, state diagram, flow chart and responses to tasks in Part 1 and Part 2
- two .jl files with your code for implementing the discrete-event simulation
- a pair of data files (an entities and a state file) that you have produced from your simulation with `seed=1` and where the simulation was stopped at time T=1440.0. Part of the marking process involves, checking that these files came from **your** simulation.

When marking your assessment, the output of your code will be checked by running the code that you submit. This is the code you created in Part 2 and which you will now refine by completing the tasks outlined here.

Tasks for Part 3

Although you might not need to modify the code in `checkout_simulation.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 ensure you get the maximum number of marks for the code you have written.

1. Test and verify your code.

   Here are some sample outputs when the code is run with with `seed=2023` for T=100.0 minutes **(note the output you will submit must use `seed=1` and T=1440.0 minutes)**:

   - The start of the example state file showing the event-based output is as follows:

     ```
     # file created by code in checkout_simulation.jl
     # file created on 2023-10-17 at 09:31:52
     # parameter: seed = 2023
     # parameter: n_checkouts = 4
     # parameter: mean_interarrival = 1.0
     # parameter: mean_service = 2.0
     # parameter: mean_interproblem = 1.0
     # parameter: mean_resolution = 0.5
     # parameter: final_time = 100.0
     event_ID,customer_ID,time,event,n_customers,n_problems
     1,1,0.0,Arrival,1,0
     3,2,1.0881231417752497,Arrival,2,0
     6,2,1.8800096399448045,Departure,1,0
     5,3,2.011572939308766,Arrival,2,0
     ```

   - The start of the example entity file showing a list of entities with information output on departure is as follows:

     ```
     # file created by code in checkout_simulation.jl
     # file created on 2023-10-17 at 09:31:52
     # parameter: seed = 2023
     # parameter: n_checkouts = 4
     # parameter: mean_interarrival = 1.0
     ```

```
# parameter: mean_service = 2.0
# parameter: mean_interproblem = 1.0
# parameter: mean_resolution = 0.5
# parameter: final_time = 100.0
event_ID,customer_ID,time,event,n_waiting,n_checkout,n_problems
1,1,0.0,Arrival,0,1,0
3,2,1.0881231417752497,Arrival,0,2,0
6,2,1.8800096399448045,Departure,0,1,0
5,3,2.011572939308766,Arrival,0,2,0
```

2. Create a test harness that will:
   - run your code for 100 different seed values ranging from 1–100
   - run your code for a set of parameters specified in the following CSV file:

     ```
     # parameters
     n_checkouts,mean_interarrival,mean_service,mean_interproblem,mean_resolution,final_time
     4,1.0,2.0,1.0,0.5,1440.0
     ```

From these outputs, you could perform some statistical analysis to inform the supermarket owner about their questions. Although you **won't need to answer these questions specifically**, this next stage that you would take in the process will be addressed in the Project Report for a different problem.

Use of example output

In order to help you test your program, you are provided with some example output files (the full version of the samples shown above). Note these are for `seed`=2023 with T=100.0, but the **outputs you submit must be for `seed=1` and T=1440.0**.

- `state.csv`
- `entities.csv`

1. 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, ID numbers or units such as hours or minutes).
2. 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.

Note that you can see the metadata in the files, and from that determine any parameters used.

Detailed rubric

You will be assessed on three components of your work:

- Component 1: Conceptual model - your ability to formulate the model. **[10 marks]**
- Component 2: Functionality — your ability to write code to create a simulation, and ensure the sub-components of it work. **[20 marks]**
- Component 3: Programming style — your ability to write your program according to the specifications and general style guidelines for good code. **[30 marks]**

Note the Component 1 and Component 3 marks are detail above. More detail on the marks for Component 2 will be provided shortly.