Computational Human Performance Modeling using Queuing Network in an Open-Source Platform

by

Gandhimathi Padmanaban

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science (Human Centered Design and Engineering) in the University of Michigan-Dearborn 2021

Gandhimathi Padmanaban

gmathi@umich.edu

ORCID iD: 0000-0002-5143-8948

Dedication

To Sree Ram: Thank you for your unconditional love and support

Acknowledgements

At this moment of accomplishment, I greatly thank my thesis advisor and chair Dr. Fred Feng, for accepting me as his student, guiding me throughout my thesis, and trusting my capabilities towards the next step of my academic journey. It has been a great learning path for me, and I am forever indebted to your mentorship, support, and encouragement.

I would like to sincerely thank Dr. Sang-Hwan Kim, for being such a kindhearted, supportive advisor and thesis committee member. You have been a great mentor providing me with timely academic advice.

I sincerely thank Dr. Shan Bao, for being a great inspiration and for being so supportive of my thesis work.

My earnest thanks to all of my instructors in the Human Centered Design & Engineering program for their valuable guidance and support which helped me to succeed as a master's student.

I am extremely grateful to the people who mean a lot to me, my mom, dad, my brother, my mother-in-law and my friends for your selfless love, care, and sacrifice which made me believe in hard work.

I owe heartfelt thanks to my husband, Sree Ram for his abundant love and support who understood my passion for academics and encouraged me to pursue my dream. Without him, this would never have been possible. Thank you so much for your patience and belief in me.

And finally, I thank the almighty for blessing me with the strength, perseverance, and patience to work hard and be who I am today.

Table of Contents

List of Tables

List of Figures

List of Appendices

Abstract

Traditional usability testing requires using human subjects to perform designed tasks and measuring their performance (e.g., task completion time, workload). Although human-subject tests have great values, they are often time consuming and expensive, which prevents User Interface (UI) designers from exploring a larger design space, especially in the early design stages. A computational human model is promising to serve as a supplementary tool for designers to test a large number of design concepts and prototypes using simulations. The cognitive architecture of QN-MHP (Queuing Network-Model Human Processor) simulates human cognition as a queuing network of information processing servers based on the findings from experimental psychology and neuroscience, and it has been shown as a promising tool for UI testing. Previously, QN-MHP was implemented using several commercial software. In this project, a basic version of QN-MHP is implemented in an open-source platform with Python and a discrete event simulation library SimPy. This allows the model to be used and co-developed with the broader research community of computational human modeling. The implemented model was verified and validated with simple visual-manual tasks by comparing the model outputs with the empirical data from the literature.

Chapter 1 Introduction

Humans are considered to be the greatest source of variability due to the complex cognitive system that contributes to different performance measures for different systems they interact with. Hence, the computational human performance modeling area of research has significantly contributed towards advancements in user interface designs and human factors. According to Simon & Newell (1971), "the programmability of the theories is a guarantee of their operationality and iron-clad insurance against admitting magical entities in the head". Many advancements have been made in the past decades in the development of computational MHPs and have been huge assets to solve issues related to human factors and ergonomics.

The Queuing Network - Model Human Processors (QN-MHP) are described to be a set of memories and processors along with a set of operational principles in relativity to a computer network. They are comprised of three interacting subsystems: Perceptual subsystem, Cognitive subsystem, and Motor subsystem (Card et al., 1986). QN-MHP aims to bridge the gap between the mathematical theory of queuing networks and the procedural, knowledge-based methods utilized in the MHP/GOMS approaches, along with an additional step of including structural constraints on human performance based on neurophysiological findings (Feyen, 2002).

Computational human performance models when implemented as open-source systems provide numerous benefits to researchers in human factors and system designers. The potential for easy access, source code availability, enhanced features, and improvements make it a robust and powerful tool. ACT-R is one of the most popular cognitive architecture, a theory for simulating and understanding human cognition (ACT-R, 2021). The ACT-R has become a robust model with the help of researchers across the globe who contributed to implementing the model on various platforms.

The lack of an open-source QN-MHP model makes it difficult for designers and researchers to make use of the flexibility and plenty of assets that QN-MHP has to offer towards human performance modeling. The compelling architecture and consideration of various constraints make it a wonderful choice for testing a variety of systems and user interfaces.

Python is a powerful programming language and has gained a lot of popularity in recent years in scientific computing, machine learning, and other areas. It has extensive standard and third-party libraries which reduces the amount of code one has to write through reusability. SimPy is a process-based discrete-event simulation framework in Python. It supports the development of real-world scenario-based simulations that benefit numerous applications. Python being an open-source language has already been growing with plenty of contributions for SimPy based model applications. But one application that could greatly benefit human factors research is creating a QN-MHP model.

With that being said, these existing gaps led towards the core theme of this thesis dissertation, in which a basic version of the QN-MHP model is implemented and validated using the SimPy library in Python. A core QN-MHP structure is implemented with necessary servers in the network and a simple button press task modeled. The results of the time taken to accomplish the button press event are compared with existing empirical and previous computation model results from different commercial software to validate the performance of the developed model. The research is intended to be published as an open-source project for contributions from the broader research community in computational human performance modeling.

Chapter 2 Literature Review

Card (1986) argues that “one of the chief impediments to progress in human engineering is the lack of a model of human information-processor posed in such a way as to enable approximate engineering calculations to be made of human performance”. This resulted in the development of Model Human Processor (MHP), which to date have taken different variations that provides a great significance towards human factors and ergonomics research of complex systems involving human interaction. Among the plenty of variations in MHP, this thesis is developed with QN-MHP as the human performance model by Liu et al., (2006) as plenty of aspects of the human cognitive architecture could be represented as components of queuing network. A detailed outline of the QN-MHP among the other computational modeling approaches of mental structure and the production systems models of cognitive architecture could be found in Liu, et al., (2006, pg. 3-10).

The MHP consists of three major stages of subnetworks: Perceptual, Cognitive, and Motor. These subnetworks comprise a number of servers that are considered components of the human cognitive architecture responsible for dedicated processes. The visual, auditory, and somatosensory systems are considered as the primitive sensory systems that should be considered for developing any computation model of human performance in an environment. While the original QN-MHP modeled by Liu, et. al (2006) considered the visual and auditory systems, this dissertation is focused on modeling the visual system alone in the perceptual subnetwork. The different servers (see Figure 2-1), their processing logic, server processing times and operators implemented in the original version of QN-MHP could be found in Feyen (2002) and Liu, et. al., (2006). A simplified version of the QN-MHP is implemented in this dissertation with only the necessary servers to model the simple button press task (see Figure 4-1). Similarly, the cognitive subnetwork in the QN-MHP consists of a working memory system and goal execution system among which the task modeled in this dissertation requires the working memory system, specifically the visuospatial sketchpad (Server A) and the central executive server (Server C). The motor network controls the execution of the goal based on the response entity routed through the perceptual and working memory systems. Any desired task could be modeled with entities routed through these subnetworks, based on task components designed using the Natural Goal, Operators, Methods and Selection rules Language (NGOMSL) task analysis method developed by Kieras (1997).

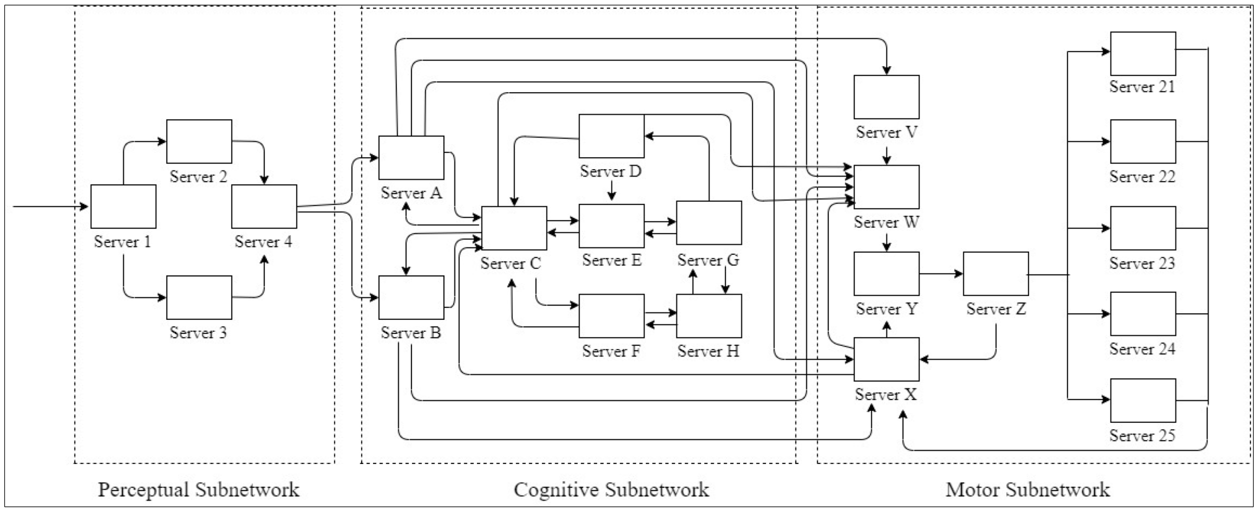


Figure 2-1: QN-MHP cognitive architecture (Liu et al., 2006)

The QN-MHP has been applied to a variety of tasks to study human performance under complex multi-tasking situations and has proved to be a versatile approach. Feyen (2002) has demonstrated the application of QN-MHP using ProModel, a commercial software for discrete event simulation, by modeling the simple reaction time, choice reaction time, visual search, and steering a driving simulator tasks. Feng (2015) has proved possibilities of QN-MHP through a series of experiments: A driver steering model, queuing network modeling of visual search task performance while driving, QN modeling of visual-manual secondary tasks while driving; all of this using MATLAB/Simulink, another commercial software that has been used for industrial simulation in a greater scale. With these two important literature and the modeling efforts as the primary inspiration for this dissertation, I have attempted to implement the QN-MHP with a simple button press task modeled using Python/SimPy with the vision to make it an open-source collaborative platform through which it could be made beneficial to researchers and designers across the globe and eventually making it an integrated library that could be used for QN-MHP simulation for different applications.

Python with an extensive support of libraries like SciPy, NumPy, SimPy, etc. has been a rapidly growing programming language that has contributed immensely towards computational studies, scientific and mathematical researches (Millman & Aivazis, 2011). Nosrati (2011) argues that Python is a fast, powerful, reliable, portable, and simple open-source programming language that supports other languages and is the best suit for real-world programming. Python has unique features supporting runtime assembly of custom user-defined code which allows flexible monitoring of the simulation process using dynamic arrays and dictionaries of variable dimensionality and also provides real-time multivariate analysis of the simulation results (Gathmann, 1998). With SimPy being a dedicated Python library for real-world simulations, it provides access to plenty of built-in modules that makes creation, running, and managing a simulation easier. Given the exceptional benefits, this dissertation project implementation is carried out using Python/SimPy to achieve the goal of creating a computation human performance model using QN-MHP in an easily accessible open-source platform.

Chapter 3 Discrete Event Simulation with SimPy

The computational human performance Model using Queuing Network-Model Human Processor (QN-MHP) cognitive architecture (Liu et al., 2006) is implemented using Python. Python is an interpreted, interactive, object-oriented programming language that incorporates modules, exceptions, dynamic typing, high-level dynamic data types, and classes (Python Software Foundation, 2021a). Python is a powerful tool for open-source projects. Python offers an extensive collection of libraries to support a variety of applications.

The main goal of this thesis project is to implement the QN-MHP model using an open-source platform rather than the previously used commercial software like ProModel (Feyen, 2002; Feyen & Liu, 2006; Liu et al., 2006) and MATLAB/Simulink (Feng et al., 2017; Feng, 2015) so that the model can be easily accessible by the broader research community with minimal financial barriers. Simulation programming in the past have had dedicated languages developed like SIMULA (Dahl & Nygaard, 1966). However, independent libraries that could be used in any programming language is the current trend (Matloff, 2008) With Python being a robust open-source programming language, SimPy is a library written and called in Python which is a package dedicated to process-oriented discrete-event simulation. According to Matloff (2008), "Instead of using threads, as is the case for most process-oriented simulation packages, SimPy makes novel use of Python's generators capability. Generators allow the programmer to specify that a function can be prematurely exited and then later re-entered at the point of its last exit, enabling coroutines, meaning functions that alternate execution with each other." This flexibility and the numerous key features of Python/SimPy make it a strong choice for this dissertation to perform discrete event simulation for the QN-MHP model. Python/SimPy project setup for this dissertation is given in Appendix H. Detailed description on the components and implementation of Python could be found in Home - Python Developer's Guide, (Python Software Foundation, 2021b); and of SimPy could be found in Overview - SimPy Documentation, (SimPy, 2021a).

3.1 Simulation Overview

Discrete event simulation (DES) is based on the idea of simulating systems that handles discrete variables instead of continuous ones. DES is a form of parallel programming where different independent events get processed simultaneously. Similarly, the QN-MHP model is based on the assumption that information entities and stimuli are handled in the brain as a queuing network consisting of a variety of servers allocated for specific functions that run in a linear or parallel fashion based on the processing logic (Liu et al., 2006). SimPy allows the developer to use generators to manage the entry or exit among the functions through the '`yield`' keyword by resuming the execution of the function immediately after the previous yield. This being the major requirement of discrete event simulation (Matloff, 2008), SimPy is considered for the complete implementation of the simulation. SimPy allows us to simulate entities traversing through a network of servers with specified capacity.

The implementation is built from scratch starting with the implementation of basic discrete event simulation with a network of two servers, which is discussed in detail in the upcoming sections. The network consists of two basic servers and an entity generator module. The structure of this dissertation is based on the similarity of the QN-MHP architecture and Discrete Event Simulation embedding parallel processing along with the '`yield`' feature of SimPy. For every complex network, there has to be a base structure or starting step on top of which building additional layers would provide a robust framework for the desired goal. In such a notion, the simple two-server network is developed as the first step for this dissertation to manipulate, execute and test the basic needs of the QN-MHP framework following which just the addition of further necessary servers would achieve the goal.

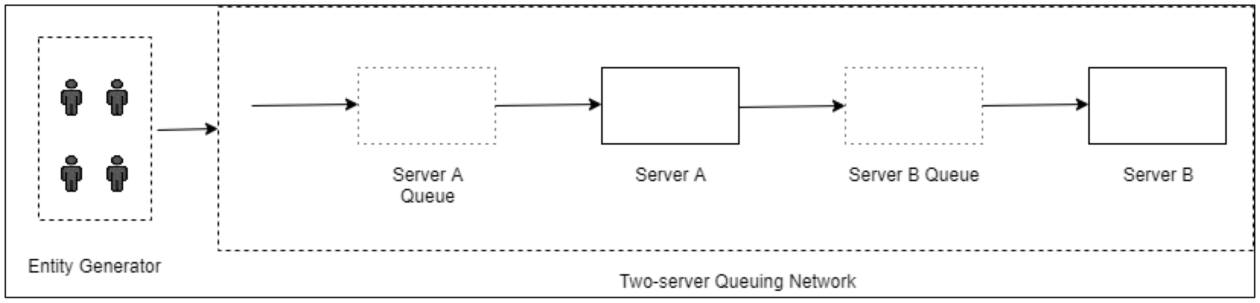


Figure 3-1: Two-server queuing network architecture for discrete event simulation

Using the SimPy library active components, in our case the entities, are modeled with process functions. These processes are controlled inside an environment to which the processes interact through events (SimPy, 2021a). An overview of this simulation architecture implemented using SimPy is shown in Figure 3-1. The simulation would consist of a two-server network and an entity generator. Here, the entity is an object created for the custom Python class `Entity` (see Appendix A) which has different attributes such as the unique ID, name, a request type which notates the server it intends to enter, and a random processing type which decides the processing time it takes on a server. A batch of four initial entities is created and sent to the network for processing which starts from Server A and proceeds to Server B. Along the course, an entity is created every 7 ms (`T_INTER` in Table 3-1) that enters the network. The entities enter the queue of each of the servers and are allowed to enter the server based on the capacity. If the capacity is full the entity waits in the queue until a space becomes available.

At the start of the simulation, the environment module is instantiated for the use of the current running simulation. Next, the instantiation of the two servers in the network is done, along with the generated first batch of 4 entities that enter Server A by default. Based on the capacity the Server A processes entities and routes them to Server B for further processing. Once the processing of an entity in Server B is completed, the entity exits the network. The detailed configuration parameters set for the simulation are given in Table 3-1.

Table 3-1: Configuration attribute fields

Configuration Attributes	Values	Description
`SIM_TIME`	50 ms	Total simulation run time
`T_INTER`	7 ms	Inter-arrival time upon which each entity is generated
`SERVER_A_CAPACITY`	2	The maximum number of entities Server A can process at a time.
`SERVER_B_CAPACITY`	2	The maximum number of entities Server B can process at a time.
`REQ_TYPES`	A list of two strings ‘A’ and ‘B’ denoting the two servers	An array that has the notations for which server the entity is routed to. [Future Implementation]
`PROCESS_TYPES`	[1,2]	An array from which a random value is set to the entity which decides how long it is processed in a server.

3.2 Queuing Network Implementation

The initial instantiation of servers is handled by a custom-built Queuing Network (QN) Module that creates two SimPy resource objects for each of the servers. A SimPy resource object is created with reference to the environment and required capacity of usage slots that can be requested by the processes, in this case, entities (SimPy, 2021b). The entities use the resource object's `request()` method to wait until the resource becomes available with a slot. The resource slot once used has to be released for the next entity to use, this is implemented using the `with` keyword while calling the `request()` method which automatically handles the release of the resource (SimPy, 2021b). Each of the servers is a SimPy resource that is set with a specific capacity, which is the number of entities it can serve at a given amount of time. Three separate data arrays [`serverA_Data, serverB_data, and network_data`] are created in the UI module to manage the data required for the visualization plots for the two servers and the entire network. A Python data array is a variable that is used to store more than one value or a list of values. These data arrays are initiated and monitored along with the resource creation and the data arrays and are updated every 0.1 milliseconds (0.1 ms is an arbitrary simulation clock time value for model verification). Utilization for a server is calculated as the percentage of occupied slots in the server at a given time. The data arrays of Server A and Server B are updated with the current timestamp and the respective server's current utilization whereas the array for the Network data is updated with the utilization of the entire network, which is the average utilization of both Server A and B. These three arrays are later used in the UI Module to generate utilization plots for the servers and the network.

Post the instantiation of the Servers in the QN module, the customer generator method in the QN module is called which generates four initial entities. Here the count four for the initial batch of entities is an arbitrary value set, just to exceed the capacity of the servers so that the queuing wait could be tested, and it can be modified to any desired value. For the sake of simplicity, the request type for all the entities is 'A' by default which means every entity enters the network through Server A and continues its processing. The entity unique IDs are consecutive integers that start from 0 and are stored locally to iterate and create the successive entities. An inter-arrival time of 7 ms (it is an arbitrary number set to test the entity generation at this simulation time) is set to the entity generator, due to which after the first set of 4 entities, the generator keeps generating one entity every 7 ms until the simulation ends. Once the entity object is created, they are routed to Server A based on the default request type.

For this basic implementation of queuing network, both the servers are given a capacity of 2, which means a server can only process up to two entities at a time, during which other entities that get in the queue are held until a spot becomes available. By default, the SimPy queue processing works based on the First Come First Serve (FCFS) rule. Hence, it maintains the order of the entities that enter the queue and allocates the available space based on the order.

As soon as an entity enters the network, it enters the queue for Server A. When an entity gets into the queue, it makes a request to the server resource for allocating a spot for processing. The resource object checks for availability in the server in accordance with its capacity. If a spot is available, then the entity is permitted inside the Server. Based on the processing time variable of the entity object, a corresponding processing time is considered. As there are no computational changes required, during the entire simulation, processing in a server is nothing more than the entity held in the server for a specified amount of time and is released. In this case, when the entity enters into the server it waits in the Server for the simulation clock to run for the specified amount of time.

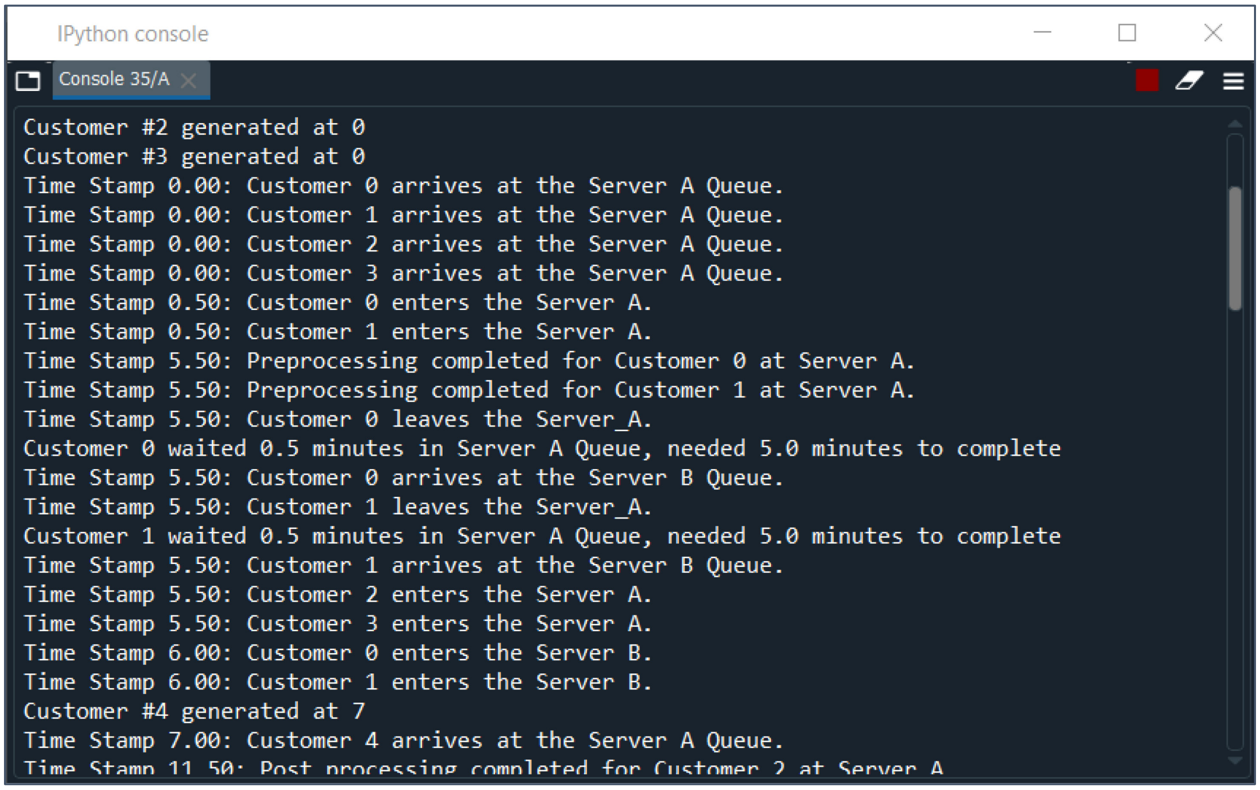
Once the processing time is complete, they exit from Server A and are routed to Server B of the network. Similar steps from Server A are followed in Server B as well but with different processing times. For testing the network behavior, the processing time in Server B is set to a different value from Server A. Hence, if two entities are already in Server B, and if other customers finished processing in Server A, they wait in the queue for the prior entities to exit after their processing. After the processing is completed in Server B the corresponding entity exits the network.

3.3 User Interface

The user interface of this simulation is split into three different parts: Simulation status logs, Queuing Network Graphics, and Utilization plot visualization. The SimPy library does not provide built-in visualization support for the simulation. Hence, three major plugins were imported into the project: Tkinter (Python Software Foundation, 2021), `matplotlib.pyplot` (Matplotlib, 2021b), and `matplotlib.backends.backend_tkagg` (Matplotlib, 2021a). Each of the user interface parts is created as separate window frames of a `Tkinter` canvas.

Simulation Status Logs: Throughout the simulation, the Python console is provided instantaneous status messages that will be printed in sync with the simulation clock (see Figure 3-2). Status messages are printed as soon as the simulation starts for every update; when an entity is generated; when an entity enters a server's queue; when an entity exits a server's queue; when an entity enters a server; when the processing starts for an entity in the server; when the processing ends for an entity in the server and finally when an entity exits the server. These status log messages help debug and verify the simulation to ensure it runs as expected.

Queuing Network Graphics: Using the Tkinter plugin a Canvas object is created at the start of the simulation to update the graphical flow of the entities in the queuing network during the simulation. This canvas help visualize the simulation workflow (see Figure 3-3). The graphics have two parts: Server figures and the Clock running. The first part has four components starting with a virtual space for Server A Queue and a rectangular box denoting Server A, then a virtual space for Server B Queue followed by a rectangular box denoting Server B. Each entity in the simulation is represented by a graphical image format of a human icon. The icon of an entity traverses through the different components based on its current location in the simulation. The rectangular boxes representing the servers have a black outline and also have an additional feature, which is that they change in color based on their current utilization. It would be filled with white when empty, dark red with the capacity is full or the alpha level of the red color increases or decreases based on the utilization value stored in the data array. The middle clock component is a small rectangle that shows the current simulation time that ticks based on the simulation environment.



```
Customer #2 generated at 0
Customer #3 generated at 0
Time Stamp 0.00: Customer 0 arrives at the Server A Queue.
Time Stamp 0.00: Customer 1 arrives at the Server A Queue.
Time Stamp 0.00: Customer 2 arrives at the Server A Queue.
Time Stamp 0.00: Customer 3 arrives at the Server A Queue.
Time Stamp 0.50: Customer 0 enters the Server A.
Time Stamp 0.50: Customer 1 enters the Server A.
Time Stamp 5.50: Preprocessing completed for Customer 0 at Server A.
Time Stamp 5.50: Preprocessing completed for Customer 1 at Server A.
Time Stamp 5.50: Customer 0 leaves the Server_A.
Customer 0 waited 0.5 minutes in Server A Queue, needed 5.0 minutes to complete
Time Stamp 5.50: Customer 0 arrives at the Server B Queue.
Time Stamp 5.50: Customer 1 leaves the Server_A.
Customer 1 waited 0.5 minutes in Server A Queue, needed 5.0 minutes to complete
Time Stamp 5.50: Customer 1 arrives at the Server B Queue.
Time Stamp 5.50: Customer 2 enters the Server A.
Time Stamp 5.50: Customer 3 enters the Server A.
Time Stamp 6.00: Customer 0 enters the Server B.
Time Stamp 6.00: Customer 1 enters the Server B.
Customer #4 generated at 7
Time Stamp 7.00: Customer 4 arrives at the Server A Queue.
Time Stamp 11.50: Post processing completed for Customer 2 at Server A
```

Figure 3-2: Simulation status logs in Python console

Visualization of server and network utilizations: This window contains the utilization plots for Server A, Server B, and the entire network (see Figure 3-3). The x-axis of the plots shows the simulation time while the y-axis shows the utilization of the respective resource. The plots are time series subplots in a grid created using `matplotlib.pyplot` plugin. The `matplotlib.backends.backend_tkagg` plugin helps to aggregate the plots created to the Canvas frame. The canvas is updated every 0.1 ms for all three interface sections.

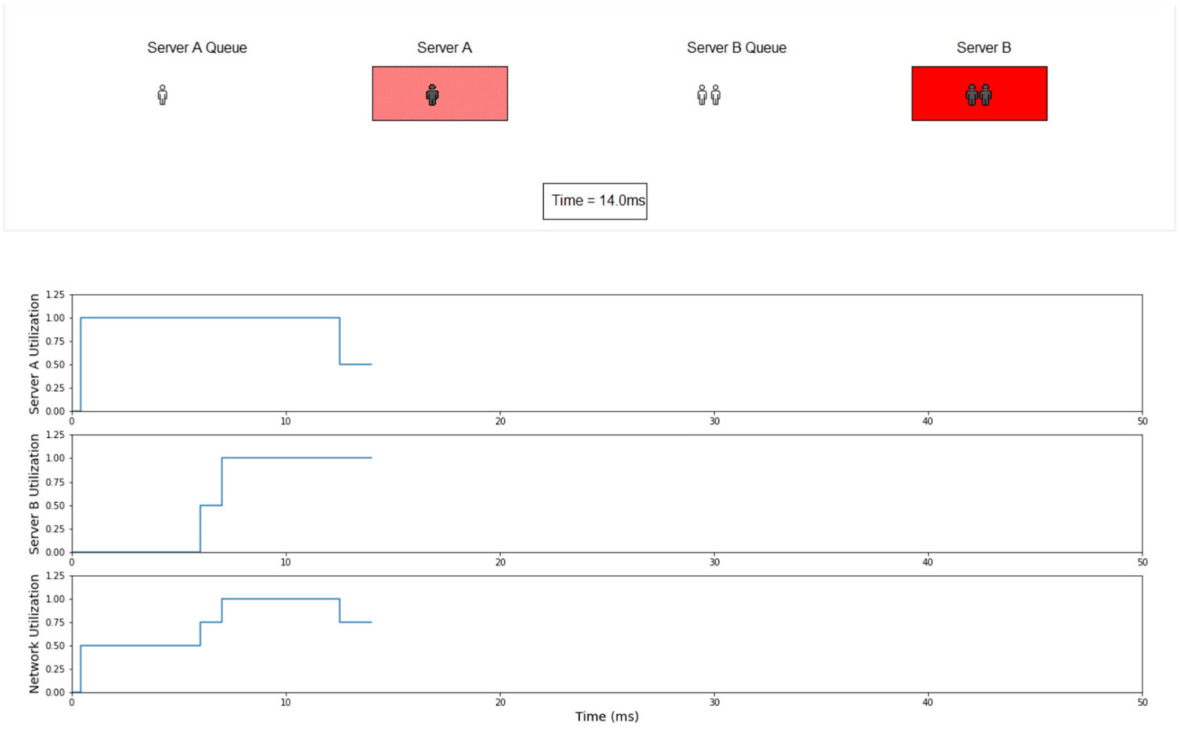


Figure 3-3: Queuing graphics and visualization plots of server and network utilization

Chapter 4 QN-MHP Model Development

4.1 Simulation Overview

Once the basic queuing network simulation in Chapter 3 was implemented and verified, it is considered as the structural base for the QN-MHP model. The QN-MHP has two major parts: The QN-MHP (here considered as a queuing network) representing the several servers of the human cognitive model; and The Modeling of the task which uses the cognitive structure to accomplish a goal. For this thesis, a simple button press task is modeled as a basic common task for computational human performance modeling. The various configuration attributes that are used in this project are listed in Appendix I based on Feng, 2015; Feyen, 2002. Apart from the configuration attributes, a memory array file is created, that maintains important global variables necessary to track the state and progress of the simulation (see Appendix J). Similar to the queuing network base implementation, the servers of the queuing network are SimPy resources (SimPy, 2021b) and follow the same process of resource allocation mechanism.

The queuing network and model implementation follows a modular pattern where most of the components like individual servers, subnetworks, user interface, modeled task, etc. are developed as reusable, separate modules using Python class. Every subnetwork, server, UI component, and evaluation model task is created as Python classes in individual files (see Figure 4-1 for the module structure of the implemented project). The modular components help the code reusability and maintenance of the project. One of the advantages of creating an open-source software is to foster collaboration. In this project, we built a human performance computational model that provides the benefit of free and open access to the public. It has the potential to be enhanced in the future through contributions from the broader research community of computational human performance modeling. Hence, the project is structured in a way using classes, adjustable global parameters, necessary comments to explain the code, and flexibility to add further features. The project is planned to be published on GitHub (GitHub Inc., 2021) for collaborative development.

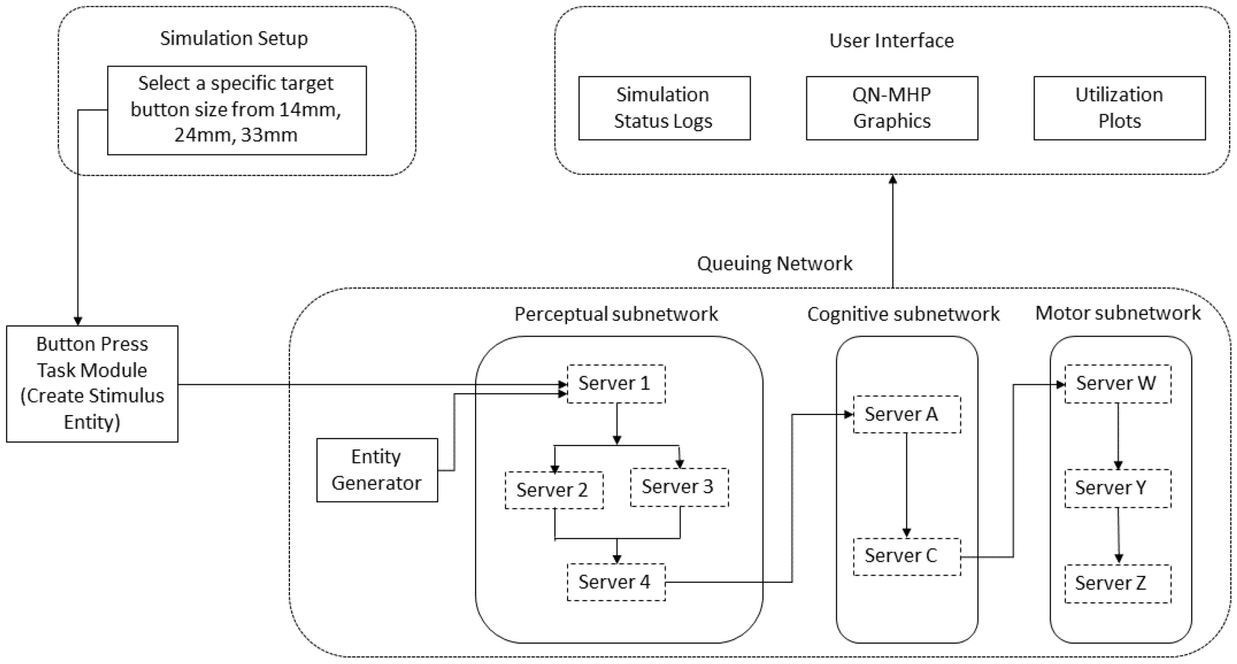


Figure 4-1: Modular architecture and components of the implemented simulation project

4.2 QN-MHP Implementation

The QN-MHP handles the entity generation and its routing through servers of the queuing network. According to Feyen (2002), every input stimulus received by any receptor such as the human eyes and ears is considered as information entities that are processed in the human brain. These entities flow through a series of subnetworks, that are composed of different servers for different processing functions and undergo processing that results in a motor action like moving an arm or touching a button. The QN-MHP cognitive architecture is considered to have three major subnetworks: Perceptual, cognitive, and motor (Liu et al., 2006). Considering the modeling of a simple button press task in this thesis, only the necessary servers, which are Visual Input Server (Server 1), Object Recognition (Server 2), Object Recognition Server (Server 3), and Visual Integration Server (Server 4), Visuospatial sketchpad (Server A), Central executive Server (Server C), Motor Element Retrieval Server (Server W), Response Tuning Server (Server X), Motor Programming Server (Server Y), Eye and Right-hand Server (Server Z), are implemented while the rest are considered for future development. See Figure 2-1 for the complete cognitive architecture of QN-MHP.

4.2.1 Information Entity Generation

Every entity is an object that carries a list of attributes (see Appendix A) about the current state, their goal, etc. The entity generator is set to create an entity every 50 ms, which is considered as the interarrival rate of entities to the network (Feng, 2015; Feyen, 2002) until either the goal is accomplished or the simulation time ends. It is also allowed to create up to a maximum capacity of 10000 entities, which is the maximum capacity of Server 1 in the perceptual network. The created entity object is updated with a `current_goal` and is appended to a global array (`ENTITY_TRACKING_ARRAY`). This array is meant to track all the entities that are created during the simulation. The initial entity is created by the task modeled which is considered to be the primary entity for the simulation. All the subsequent entities created by the entity generator method are meant to keep the network updated so that the initial entity has not timed out or gets forgotten. The created entities are routed to the queuing network for further processing. Figure 4-1 shows the information entity generator being a part of the queuing network module and sends the entities towards Server 1.

4.2.2 Perceptual Subnetwork

The Perceptual subnetwork of the QN-MHP framework consists of four servers: Visual Input Server (Server 1), Object Recognition (Server 2), Object Recognition Server (Server 3), and Visual Integration Server (Server 4). By default, an entity entering the queuing network gets into the queue of Server 1. The processing of the entities in each of the servers in the network is implemented using the "`process`" method of the Environment module in SimPy. It allows us to customize the process flow to be either sequential or parallel. Depending on whether we specify the process to "`yield`" or not, decides whether the simulation runs sequential for that process or parallel respectively.

The perceptual network module of the implementation is responsible for instantiating the servers during the start of the simulation as well as initiate the monitoring of the four servers to update the user interface. When an information entity enters the queuing network either from the modeled task or the entity generator, it is checked against the global `Tracking_Variable` from the memory arrays. An entity can route through the network if any of the following conditions are satisfied:

1. The entering entity is the initial entity from the task and the goal is not accomplished yet.
2. The entity is from the entity generator and the initial entity from the task is either put on hold or has decayed due to the processing running beyond its decay time.

Each of the entities takes a certain amount of processing time based on the server it is in. In general, one perceptual cycle is considered to take a minimum processing time of 25 ms and a mean processing time of 42 ms (Feyen, 2002 - Table 4.37). Hence, for servers that takes one perceptual cycle of processing time a random value between the minimum and the mean perceptual processing time is generated based on the exponential distribution (see Appendix F for calculation) at the server and that is used for calculating the processing time it takes in that specific server.

Visual Input Server (Server 1): The visual input server is the starting point for an entity in the queuing network. It is responsible for splitting the entered entity into two parts '(a)' and '(b)' and routing it to the Object Recognition Server (Server 2) and Object Location Server (Server 3). The maximum capacity of this server is 10000 (Feng, 2015) at a given time unit. The stimulus or information entity contains information about the goal and target specifications like the size of the button to be pressed. Due to its large capacity, the chance of an entity in this simulation to wait in the queue of Server 1 is highly unlikely. As the server only does routing, there is no processing time accounted in this server. Hence, the entity enters and exits at the same simulation time due to which it may not be logically visible in the UI to see an entity icon get in or out of this server. However, every step of the entity in the server and its queue would be recorded in the status tracking log messages.

Object Recognition Server (Server 2): This server is dedicated to decomposing the stimulus entity's characteristics such as size, shape, color, and label of the target, and update the necessary information to the entity tracking object. For this, the server takes one perceptual processing cycle. On contrary to the visual input server, this server has a limited capacity of 4 entities (R. Feng, 2015) that could be processed a given time unit. In this case, as the initial entity from the modeled task is the only most likely entity to be processed it may not have to wait in the queue. The entity is logged upon its entry in the queue and the server when it has capacity. During the processing, it copies the individual object characteristics such as label, color, shape, and size of the target to the partial entity object. Once the processing is completed in this server a corresponding flag is updated in the entity '(a)' part and it exits the server and proceeds towards the Visual Integration Server (Server 4).

Object Location Server (Server 3): This server is dedicated to obtaining the location attributes of the target producing the stimulus into the corresponding variable of the entity part. For this, the server takes one perceptual processing cycle which is calculated using the random time value following exponential distribution of the minimum and mean processing times of one perceptual cycle. Similar to the object recognition server, this server has a limited capacity of 4 (Feng, 2015) entities that could be processed in a given unit of time. The entity is logged upon its entry in the queue and the server when it has capacity. During the processing, it copies the X, Y, and Z location coordinates of the target that generates the stimulus. Once the processing is completed in this server a corresponding flag is updated in the entity '(b)' part and it exits the server and proceeds towards the Visual Integration Server (Server 4).

Visual Integration Server (Server 4): The visual integration server has two major roles: identifying the stimulus represented by the entity and storing these sensory inputs until they decay (Feyen, 2002). This server has a limited capacity of 5 entities which it can process at a given unit of time. To identify the stimulus, this server integrates and reassembles the split entities from object recognition and location servers. It is crucial to note that until the entity exits Server 1 and enters Server 4 the SimPy environment process is coded to run in parallel, whereas in the rest of the implementation they are programmed to be sequential (it waits for once process to be completed before starting the next). According to the QN-MHP model (Liu et al., 2006), processing in Server 2 and 3 starts simultaneously but could be for different processing times due to the random time generation at each of the servers. As a result, either of the entity parts could enter Server 4 first. Hence, for the visual integration server to process, whichever part of the entity gets into the queue waits until the other part finishes its processing as well and enters the server. Once both the sub-entities get into the queue, they enter the server and are made one single entity again by copying all the collected attributes into the parent. For this processing, the server takes one perceptual cycle to complete and routes the entity towards the cognitive subnetwork.

4.2.3 Cognitive Subnetwork

The full cognitive subnetwork of the QN-MHP framework consists of seven servers, which are Visuospatial sketchpad (Server A), Phonological Loop (Server B), Central executive Server (Server C), Procedural list Server (Server D), Performance Monitor Server (Server E), High-Level cognitive operators Server (Server F), and Goal prioritization Server (Server G). However, for modeling a simple button press task it only involves two servers: Visuospatial sketchpad (Server A) and Central executive Server (Server C), which are the only server implemented in the cognitive subnetwork of this thesis leaving the rest for future work. By default, an entity entering the cognitive subnetwork gets into the queue of Server A. The cognitive network module of the implementation is responsible for instantiating the servers during the start of the simulation as well as initiate the monitoring of the two servers to update the user interface.

Neurological evidence suggests that because of the larger capacities in the perceptual subnetwork the probability of an entity to be delayed is fairly less (Feyen, 2002). Whereas the cognitive server has a limited capacity which could make entities wait until the specific server has availability. In our case, since only one entity is always processed, the entity waiting in any of the servers is highly unlikely. Each of the entities takes a certain amount of processing time based on the server it is in. In general, one cognitive processing cycle is considered to take a minimum processing time of 6 ms and a mean processing time of 18 ms. Hence, for servers that takes one cognitive cycle of processing time a random value between the minimum and the mean cognitive processing time is generated based on the exponential distribution (see Appendix F for calculation) at the server and that is used for calculating the processing time it takes in that specific server.

Visuospatial Sketchpad (Server A): The visuospatial sketchpad also known as the "mind's eye", is responsible for maintaining and manipulating the visual and spatial information of the stimulus (Feyen, 2002). This server is the starting point for an entity in the cognitive subnetwork and has a limited capacity of 4 (Feng, 2015) entities that it can process at a given time unit. For processing an entity, the server takes one cognitive processing cycle. Every step of the entity in the server and its queue would be recorded in the status tracking log messages. Once the processing is completed the entity will be routed to the central executive server.

Central Executive Server (Server C): Server C is responsible for coordinating the memory and perceptual retrieval tasks while maintaining attention towards few entities that they hold (Feyen, 2002). This server processes the incoming entity from the visuospatial sketchpad with a limited capacity of 3 entities (Feng, 2015) which it can process at a given time unit. For processing an entity, the server takes one cognitive processing cycle. Every step of the entity in the server and its queue is recorded in the status tracking log messages. Once the processing is completed the entity will be routed to the motor subnetwork.

4.2.4 Motor Subnetwork

The full motor subnetwork of the QN-MHP framework consists of five servers, which are Sensorimotor Integration Server (Server V), Motor Element Retrieval Server (Server W), Response Tuning Server (Server X), Motor Programming Server (Server Y), Eye and Right-hand Server (Server Z). Similar to the cognitive subnetwork, for modeling a simple button press task, not all the servers are necessary, and it only involves three servers: Motor Element Retrieval Server (Server W), Motor Programming Server (Server Y), Eye and Right-hand Server (Server Z), which are the only servers implemented in the motor subnetwork of this thesis leaving the rest for future work. By default, an entity entering the motor subnetwork gets into the queue of Server W. The motor network module of the implementation is responsible for instantiating the servers during the start of the simulation as well as initiate the monitoring of the three servers to update the user interface.

Each of the entities takes a certain amount of processing time based on the server it is in. In general, one motor processing cycle is considered to take a minimum processing time of 10 ms and a mean processing time of 24 ms. Hence, for servers that take one motor cycle of processing time a random value between the minimum and the mean motor processing time is generated based on the exponential distribution (see Appendix F for calculation) at the server and that is used for calculating the processing time it takes in that specific server.

Motor Element Retrieval Server (Server W): This server is the starting point for an entity in the motor subnetwork and has a limited capacity of only one entity that can be processed at a given time unit (Feng, 2015). It is responsible for retrieving the set of elements from memory based on the entity and passed them to the moto programming server for assembly, sequencing, and release (Feyen, 2002). For processing an entity, the server takes one motor processing cycle. Every step of the entity in the server and its queue would be recorded in the status tracking log messages. Once the processing is completed the entity will be routed to the Motor Programming Server.

Motor Programming Server (Server Y): This server is responsible for loading the appropriate motor program associated with the entity and the goal. It has a limited capacity of 2 entities which it can process at a given time unit (Feng, 2015). For processing an entity, the server takes one motor processing cycle. Every step of the entity in the server and its queue would be recorded in the status tracking log messages. Once the processing is completed the entity will be routed to the Eye and Hand Server.

Eye and Hand Server (Server Z): Server Z consists of two parts: Actuator of the Eyes and Actuator of the Hand movement. It has a limited capacity of 5 entities which it can process at a given time unit (Feng, 2015). The actuator of the eye handles the eye movement towards the target if it is a moving object. In the case of the implementation the target is considered to be a static button in the screen and the eye is assumed to be always looking at the target. Hence, no specific processing of the eye actuator is considered or implemented currently, which could be future work. However, the actuator for the right hand has been implemented which estimates the time taken to move the right-hand index finger to reach the target based on Fitts's Law (Bi et al., 2013). Based on the size of the button in the target Fitts's law formula is applied with error coefficients and the time taken to reach the button is calculated (see Appendix G). Along with this, the time taken to tap on the target button and release the finger is taken as 280 ms - 70 ms (Feng, 2015). These calculated time together form the processing time taken by the entity at this server. Every step of the entity in the server and its queue would be recorded in the status tracking log messages. Once the processing is completed, the goal accomplished flag of the entity and global tracking variable are updated. This routes the entity out of the queuing network and the simulation ends.

4.3 User Interface

The user interface of this simulation is split into three different parts: Simulation status logs, QN-MHP Graphics, and Utilization plots. As mentioned in the User interface section of the Discrete event simulation (Chapter 3), the SimPy library does not provide a built-in user interface or visualization support that we could use for the simulation. Similar to the prior simulation, additional libraries were imported making the UI module a distinct feature for the simulation. However, given the requirements to visualize the QN-MHP along with the three major libraries (`Tkinter`, `matplotlib.pyplot`, and `matplotlib.backends.backend_tkagg`), one more important library called Pillow (PIL – Python Imaging Library) is also used for visualizing server utilization by changing the opacity of the server colors.

The UI module of QN-MHP is enhanced and implemented differently than the one done for the discrete event simulation for better usability and user experience. Here the interface has three different canvas windows created using the Tkinter library: one for the status logs, one for queuing graphics, and one for the utilization plots, rather than having them in a single canvas as separate sections in the discrete event simulation attempt. The implementation of each of the canvas parts is discussed in the upcoming sections.

4.3.1 Target Configuration Wizard

The QN-MHP model implemented opens a configuration wizard at the start of the simulation. It is a Tkinter canvas that shows the target button label of the simulation as 'USB'. It also includes three input buttons that shows the three different button sizes (14 mm, 24 mm, 33 mm) that are planned to be modeled and verified against existing literature results (see Figure 4-2). The user or the analyst has to click one among the three buttons which would pass the corresponding button size as an input parameter to the button press task modeled. The selected button size will be considered for Fitts's law calculation at Server Z. The advantage of this feature is that it is implemented as a separate Python class component which could be easily modified to accommodate complex modeling input parameters in future enhancements.

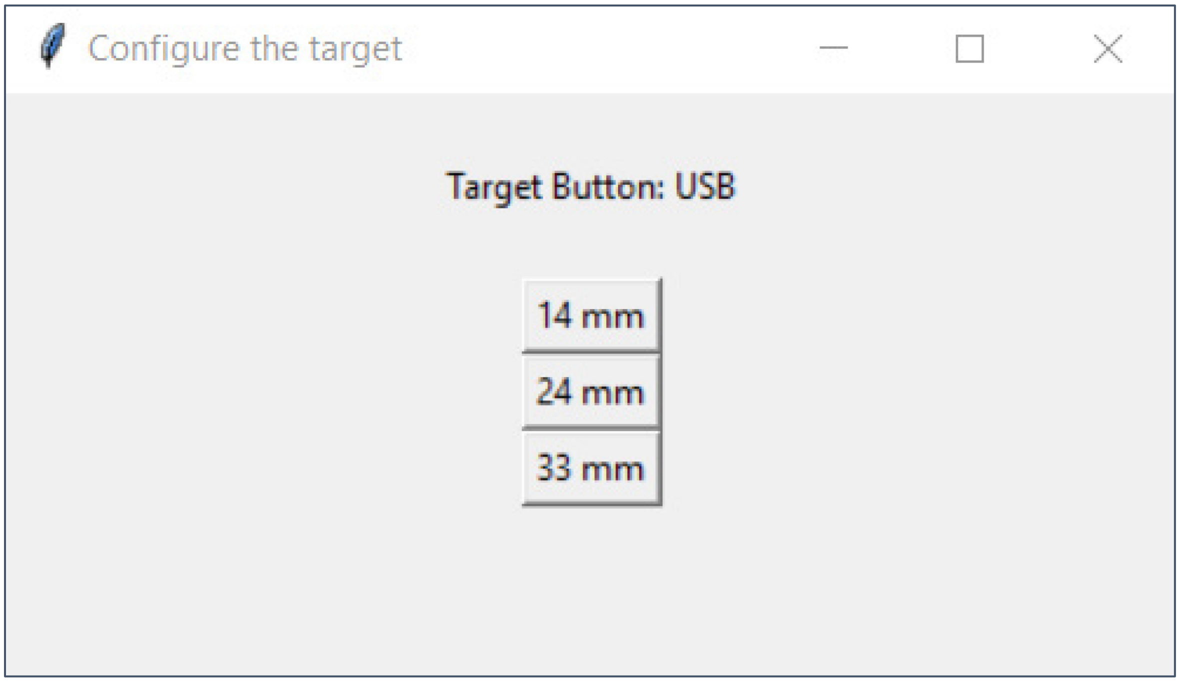
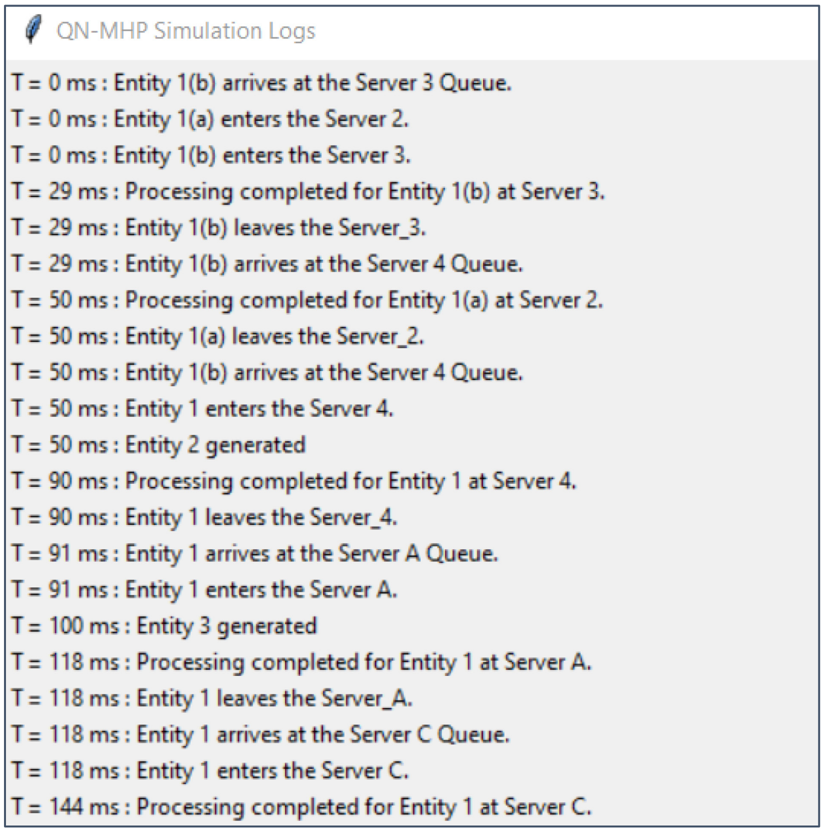
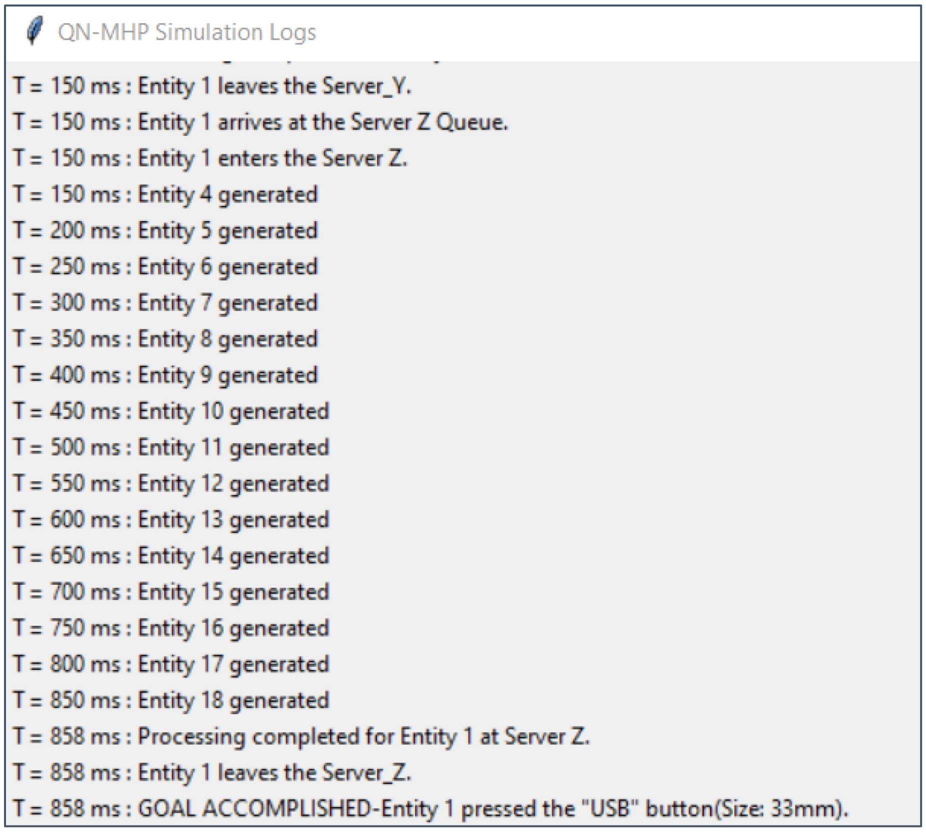


Figure 4-2: Target Configuration Wizard



```
T = 0 ms : Entity 1(b) arrives at the Server 3 Queue.
T = 0 ms : Entity 1(a) enters the Server 2.
T = 0 ms : Entity 1(b) enters the Server 3.
T = 29 ms : Processing completed for Entity 1(b) at Server 3.
T = 29 ms : Entity 1(b) leaves the Server_3.
T = 29 ms : Entity 1(b) arrives at the Server 4 Queue.
T = 50 ms : Processing completed for Entity 1(a) at Server 2.
T = 50 ms : Entity 1(a) leaves the Server_2.
T = 50 ms : Entity 1(b) arrives at the Server 4 Queue.
T = 50 ms : Entity 1 enters the Server 4.
T = 50 ms : Entity 2 generated
T = 90 ms : Processing completed for Entity 1 at Server 4.
T = 90 ms : Entity 1 leaves the Server_4.
T = 91 ms : Entity 1 arrives at the Server A Queue.
T = 91 ms : Entity 1 enters the Server A.
T = 100 ms : Entity 3 generated
T = 118 ms : Processing completed for Entity 1 at Server A.
T = 118 ms : Entity 1 leaves the Server_A.
T = 118 ms : Entity 1 arrives at the Server C Queue.
T = 118 ms : Entity 1 enters the Server C.
T = 144 ms : Processing completed for Entity 1 at Server C.
```

Figure 4-3: Simulation log window with event updates



```
QN-MHP Simulation Logs
T = 150 ms : Entity 1 leaves the Server_Y.
T = 150 ms : Entity 1 arrives at the Server Z Queue.
T = 150 ms : Entity 1 enters the Server Z.
T = 150 ms : Entity 4 generated
T = 200 ms : Entity 5 generated
T = 250 ms : Entity 6 generated
T = 300 ms : Entity 7 generated
T = 350 ms : Entity 8 generated
T = 400 ms : Entity 9 generated
T = 450 ms : Entity 10 generated
T = 500 ms : Entity 11 generated
T = 550 ms : Entity 12 generated
T = 600 ms : Entity 13 generated
T = 650 ms : Entity 14 generated
T = 700 ms : Entity 15 generated
T = 750 ms : Entity 16 generated
T = 800 ms : Entity 17 generated
T = 850 ms : Entity 18 generated
T = 858 ms : Processing completed for Entity 1 at Server Z.
T = 858 ms : Entity 1 leaves the Server_Z.
T = 858 ms : GOAL ACCOMPLISHED-Entity 1 pressed the "USB" button(Size: 33mm).
```

Figure 4-4: Simulation log window with event updates during simulation end

4.3.2 Simulation Status Logs

Simulation status logs play a major role in verifying the correctness of a computational model which could be compared with an expected task analysis. Understanding the importance of the log messages, the UI module of QN-MHP is designed to have a separate Canvas window that displays all the status log messages in addition to the Python console. This enhances the user experience in a way that when the simulation starts there is no need for the user or the programmer to jump between the console and UI Canvas. Instead, all the three UI components will be docked structurally in the screen without the need for toggling between screens. Status messages are printed in every necessary step in the simulation: as soon as the simulation starts; whenever an entity is generated; whenever an entity enters a server's queue; whenever an entity exits a server's queue; whenever an entity enters a server; when the processing starts for an entity in the server;

when the processing ends for an entity in the server; when an entity exits the server (see Figure 4-3) and finally when the expected goal is accomplished at which the simulation ends (see Figure 4-4). The log messages are created as label texts using the ttk module of the Tkinter library. As we know the log messages of a simulation could be more than a few lines it needs a scrollable window to accommodate all the log messages that are printed throughout the simulation. By default, the Tkinter does not provide a scrollable canvas frame, hence as a part of the development, a custom scrollable canvas window frame is implemented that updates itself and accommodates all the log messages of the simulation.

4.3.3 QN-MHP Graphics

The graphical representation of the QN-MHP simulation is constructed based on the different servers in the different subnetworks. Using the Tkinter plugin a Canvas object is created during the start of the simulation to update based on the simulation clock. This canvas help visualize the simulation workflow. The graphics consist of the QN-MHP framework structure and a Clock ticking at the top right corner. Every server from all the three subnetworks is represented as a rectangle box in the canvas with an unbounded section for their respective queues before them. Each entity in the simulation is represented by a graphical image format of a human icon create using the Tkinter PhotoImage module. The icon of an entity traverses through the different sections based on its current location in the simulation. The rectangular boxes of the servers change in color based on their current utilization. It would be filled with white when

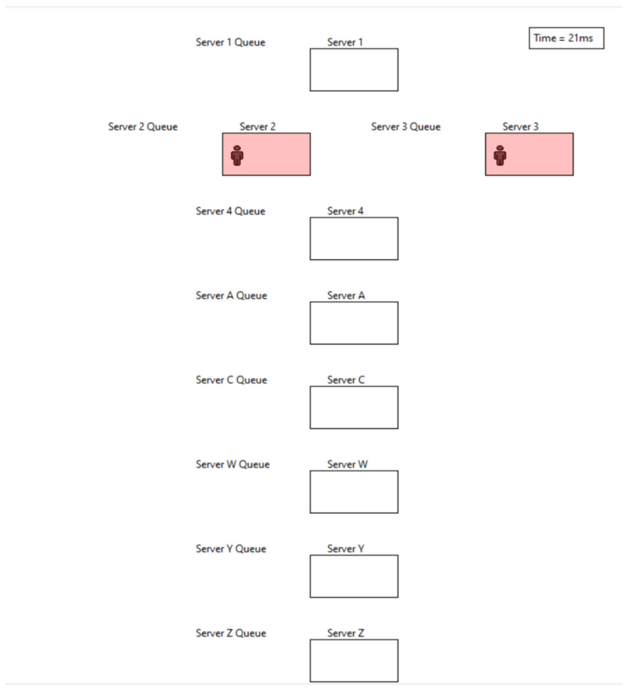


Figure 4-5: QN-MHP Graphics when the split stimulus entities are getting processed in Server 2 and 3

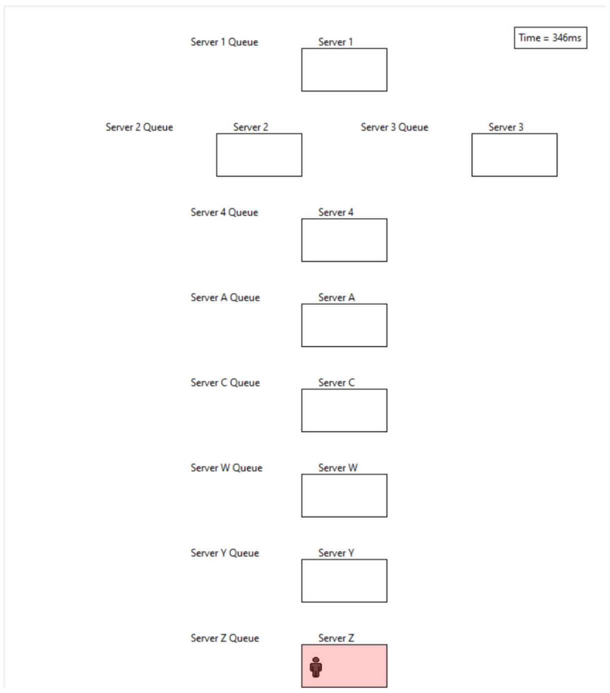


Figure 4-6: QN-MHP Graphics when the stimulus enters Server Z for processing

empty, dark red when the capacity is full or the alpha level of the red color increases or decreases based on the utilization value stored in the data array (see Figure 4-5, Figure 4-6). The top right clock is a small rectangle that shows the current simulation time that ticks based on the simulation environment. The graphics are updated for every millisecond of the simulation time. Once the simulation ends the canvas is set to be destroyed which supports the creation of new canvases for the subsequent runs of the simulation. Figure 4-5 shows a snapshot of the simulation when two split parts of an entity are getting processed in Server 2 and Server 3.

4.3.4 Utilization Plots

A separate canvas window frame shows four different time-series plots (see Figure 4-7): One for each subnetwork utilization and one for the whole Queuing network utilization. During the start of the simulation creation of the plots are automated based on the specified number of subnetworks in the global configuration. It is a reusable code that automatically creates the plot by adjusting the configuration parameter rather than manually coding the creation of every single plot. The X-axis of the plots shows the simulation time while the Y-axis shows the utilization percentage of the respective resources. The plots are created as subplots in a grid created using matplotlib.pyplot plugin. The `matplotlib.backends.backend_tkagg` plugin helps to include the plots created to the Canvas frame. The canvas is updated every millisecond. The data for each of the subnetwork plots are calculated as the mean utilization of each of the servers in the subnetwork at the specific millisecond. This calculated data is populated into a corresponding data array in the UI module with the current simulation environment timestamp. This array is then used as data fed the update of the plot. The final overall network utilization plot is created based on the network data array. The utilization values for the network data array are calculated as the mean utilization of all the three subnetworks at the specific millisecond. A server's utilization is calculated as the percentage of the total number of occupied slots for the given server capacity for a time unit. Each of the subnetwork utilization is calculated as the mean utilization of all the servers in that subnetwork. Similarly, the overall queuing network utilization is calculated as the mean utilization of all the servers in the entire network. The primary goal of plotting the server utilization is with the idea that it could be used for estimating human mental/physical workload for a given task which could be highly beneficial in studying human performance during multitasking (Wu & Liu, 2007).

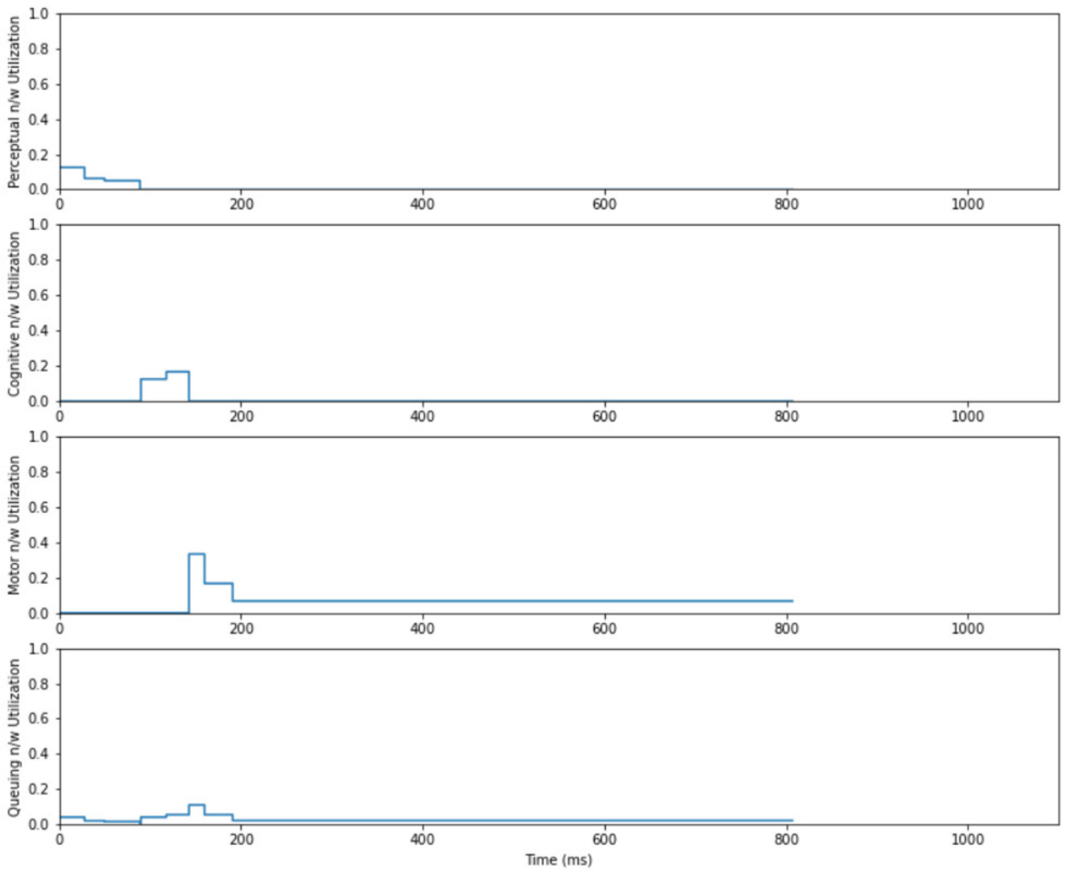


Figure 4-7: Utilization plots for a simulation to press the target ‘USB’ button of 33 mm size

Chapter 5 Model Evaluation and Validation

5.1 Evaluation - Simple Button Press Task

Evaluation of a computational model is an important step in validating the accuracy and efficiency of the model. The QN-MHP simulation in Python is developed to achieve the goal of acquiring performance measures as a result of the simulation that aligns with similar conclusions of the real-life performance of the task. It is also assumed that any changes in the QN-MHP framework impact the performance which is consistent with the real-life performance measures (Feyen, 2002). Although the QN-MHP model is not new, it was mainly implemented in commercial software (e.g., ProModel (Feyen, 2002), MATLAB/Simulink (Feng, 2015)). It would be greatly beneficial to implement it in an open-source platform so that the model is freely available to the broader research community. Hence, it is pivotal to evaluate the efficiency of the model.

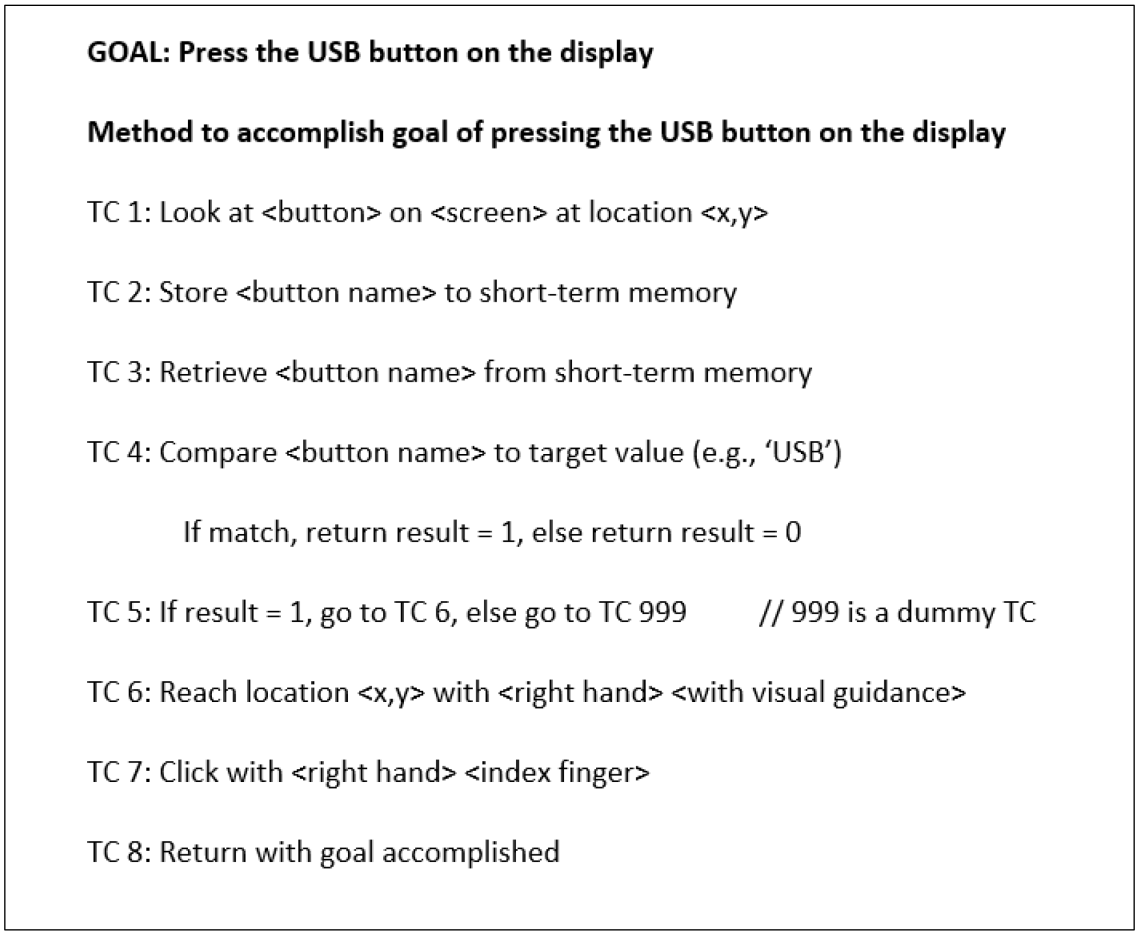
The preliminary goal of this research is to implement the QN-MHP using Python and ensure the model predicts the basic features of human performance for simple visual-manual tasks. With that intention, the initial modeling is done for a basic task modeled in most similar research, which is the effort of modeling a Simple Button Press (SBP) task. The task is modeled in such a way that a user is trying to press on a specific button, which is present in a digital screen whose location is already stored in the memory and does not change, using visual guidance. For simplicity, the motor network is programmed to always use the right-hand index finger to press the button. Once the button is pressed the simulation stops and considers the goal accomplished.

For the QN-MHP to be considered as working effectively, the performance measures of the SBP task should be similar to the results of the same task modeled in existing literature research. As the QN-MHP model is an already researched and accepted approach to modeling human performance, the evaluation of this development is primarily focused on the possibility and relativity of the implementation in Python with the existing results from works of literature for this specific task.

5.2 Task Analysis

Task analysis is a necessary step in human performance modeling to plan the procedure of actions to be performed to attain the goal. Various task analysis techniques are used in research projects. Since this research focuses on cognitive processes that are involved while performing a task, the Goals, Operators, Methods, and Selection rules (GOMS) model, specifically the Natural-GOMS-Language (NGOMSL, Kieras, 1997) variation, is used to analyze the different cognitive steps in reaching the goal. A detailed explanation of the QN-MHP along with the different server and network configurations, the creation and management of entities, the information entity flow in the network with few other complex tasks modeled could be found in Feyen (2002), Liu et al., (2006), Feng et al., (2015). The NGOMSL task analysis breaks down the task in a “top-down, breadth-first” manner into “atomic-level” Task Components (TC). Each TC is associated with a task-independent context-free QN-MHP operator from the QN-MHP operator library (Feng, 2015).

NGOMSL task analysis for the button press task is conducted with its results shown in Figure 5-1 with 8 TCs. A detailed description of each of the specific task components could be found in the visual search task section of Feng (2015).



GOAL: Press the USB button on the display

Method to accomplish goal of pressing the USB button on the display

```
TC 1: Look at <button> on <screen> at location <x,y>
TC 2: Store <button name> to short-term memory
TC 3: Retrieve <button name> from short-term memory
TC 4: Compare <button name> to target value (e.g., 'USB')
        If match, return result = 1, else return result = 0
TC 5: If result = 1, go to TC 6, else go to TC 999        // 999 is a dummy TC
TC 6: Reach location <x,y> with <right hand> <with visual guidance>
TC 7: Click with <right hand> <index finger>
TC 8: Return with goal accomplished
```

Figure 5-1: NGOMSL-style task analysis of the simple button press task

5.3 Task Assumptions

The QN-MHP implementation is based on several important assumptions in the intention to implement the basic working of the cognitive process. Firstly, it is assumed that there is a target button present in a screen that would always be present, and its location never changes. The default target button will have a label name USB which is verified in the task. It is also assumed that the eye always looks at the exact location of the target button. Hence, there is no eye movement considered in this implementation. As the eye looks by default at the target button, always a stimulus entity containing the information about the target enters the QN-MHP network from the modeled task at the start of the simulation. As soon as the stimulus entity is received in the Visual

Input Server (Server 1) it is processed through the networks. Based on the purpose of each of the servers in the QN-MHP, only specific servers as explained in Chapter 4 are implemented which are necessary for the button press task. Finally, since the first stimulus from the task always has the expected details, no specific visual search pattern is implemented.

5.4 Task Implementation and Execution

The simulation runs with the support of the SimPy library of Python. An environment object is created from the SimPy module which serves as the external environment for the entire simulation. All the different modules share the same environment object for their processing to run in alignment with the simulation. The SBP task is implemented as a Python class in a separate file to maintain modularity. The simulation starts by creating an object for the button press task module which takes the environment object as a parameter to share it across the QN-MHP. The environment process button press decides how long the entire simulation, in other words, once the goal of the button press task is accomplished it is programmed to stop the simulation rather than a specific runtime value. This allows the project to be flexible enough to build complex tasks on top of it without providing any specific runtime in the configuration.

The ButtonPressTask module in the program manages the creation of the initial stimulus entity. It creates an object of the Entity class with the unique ID value of 1. Based on the various presumptions of the modeling effort, different hard-coded configuration values are set to the entity object that consists of values for the stimulus ID of the entity, an ID for the current goal that the entity is supposed to accomplish, the size of the target button (which is one among 14mm, 24mm, 33mm, see Feng (2018)), the amplitude to calculate the movement time based on Fitt’s Law and the target label name (‘USB’) which is checked to match the value. See Appendix A for detailed Entity object variables and their description.

Two global variables are created during the initiation of the simulation: (1) an array for tracking the entities created during the simulation (`ENTITY_TRACKING_ARRAY`), for both the initial stimulus entity and the subsequent entities created by the prompt to refresh the memory, and (2) a global variable that keeps track of the initial entity, the simulation goal and tracks the goal accomplishment to complete the simulation (`TRACKING_VARIABLE`). Once the initial entity is created in the button press task method it is added to the `ENTITY_TRACKING_ARRAY` for tracking purposes and the variables of the `TRACKING_VARIABLE` are updated. The task also instantiates the clock of the UI and the entity generator method following by initiating the QN-MHP to run the series of events by passing the created initial entity. This proceeds with the QN-MHP to process the entity as explained in Chapter 4 and returns with the goal accomplished.

Once the simulation completes, the `SIM_DATAFRAME` pandas data frame (Pandas, 2021) is populated with the various simulation times, corresponding log messages, the perceptual subnetwork utilization, cognitive subnetwork utilization, and motor subnetwork utilization. A pandas data frame (Pandas, 2021) is a two-dimensional labeled data structure or simply a table that stores and handles data in Python. This data frame is then automated to be converted as an excel worksheet. With three different variations in button sizes, three excel workbooks are created one for each button size. Every time the simulation is run for a specific button size, a new worksheet is added to the workbook with the data frame values. This is considered as the results of the simulation and the final exit time, which is the task completion time, of the entity for that simulation is extracted and logged manually for validating the model.

5.5 Model Validation

The modeled button press task is validated and verified for the three different target button sizes. The validation of the model is planned to run the simulation in a random fashion selecting one of three button sizes (14 mm, 24 mm, 33mm). For a total of 60 times the simulation is run with 20 simulation runs made for each button size. The simulation results for the programmatically created excel worksheets of the three button size variations are tabulated and statistical calculations like geometric mean, standard deviation, 50th percentile, and 95th percentile are calculated (see Table 5-1, Appendix E). Statistical *t*-test analysis has been performed among the different button sizes. The observed values for the three different button groups are compared as pairs and their respective *t*-statistic, and p-values were calculated (see Table 5-2). The group comparison were done as small to medium (14mm to 24mm), medium to large (24 mm to 33mm), and small to large (14 mm to 33mm).

Table 5-1: Statistics of task completion time (all units are in seconds)

Button sizes	Geometric Mean (GM)	Std. Deviation (SD)	50th Percentile	95th Percentile
14 mm	1.109	0.303	1.092	1.137
24 mm	1.001	0.37	1.001	1.058
33 mm	0.909	0.405	0.927	0.974

The task completion times of simulations are plotted against their input target button size in Figure 5-2. The results of the simulation (see Appendix E) are compared with the results of the visual search task of different combinations of buttons in parked condition from Feng et al., (2018) considering the 25th percentile of the experiment. Though the visual search component is not implemented in this dissertation, the task components of the button press task that is modeled in this dissertation are relatively the same as the task components of the visual search task (excluding the task component for visual search) from Feng et al., (2018). The 25th percentile of task

Table 5-2: Statistical *t*-test summary of findings of task completion time

Button size	*t*-statistic	*p*-value
14 mm $\longrightarrow$ 24 mm	9.041	< .001
24 mm $\longrightarrow$ 33 mm	7.506	< .001
14 mm $\longrightarrow$ 33 mm	15.72	< .001

completion times in parked condition for the least 2x2 button combinations of Feng et al., (2018) is assumed to be possible only when the user identified the target button at their first glance (which does not need to conduct any visual search). Hence the values of Table 1 and Figure 3 from Feng et al., (2018) from the laboratory experiment are considered to validate the accuracy of theQN-MHP model in SimPy.

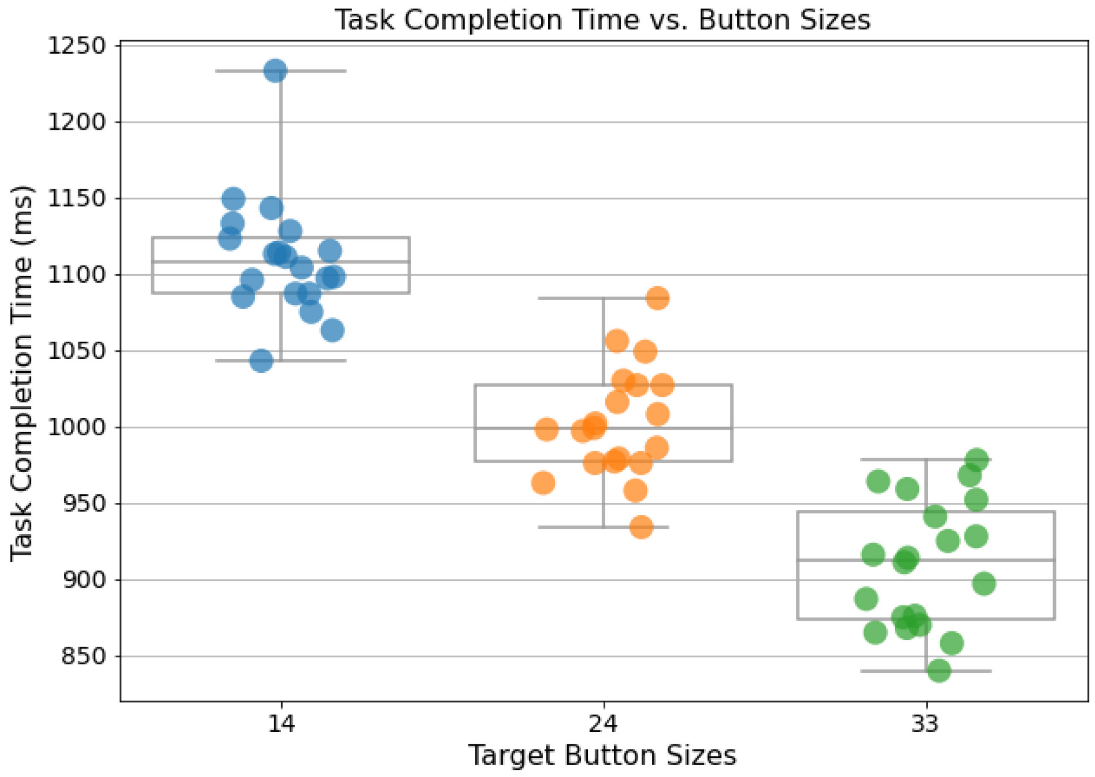


Figure 5-2: Task completion time grouped for the different button size variations

When the results of the Python/SimPy simulation are compared to the minimum task completion time in the empirical evidence, they match at a closer rate, and also the difference is assumed to be due to possible error probabilities of the set task conditions (see Figure 5-3). It is also interesting to know from Figure 5-2 and the *p*-values of Table 5-1 that the results of our simulation have shown a significant reduction in the task completion time when the size of the button increases which also verifies the evidence from the literature. Considering these results, it is indeed arguable that the implemented QN-MHP model in Python/SimPy works as expected in relevance to existing literature evidence and could still be validated for complex tasks in future enhancements.

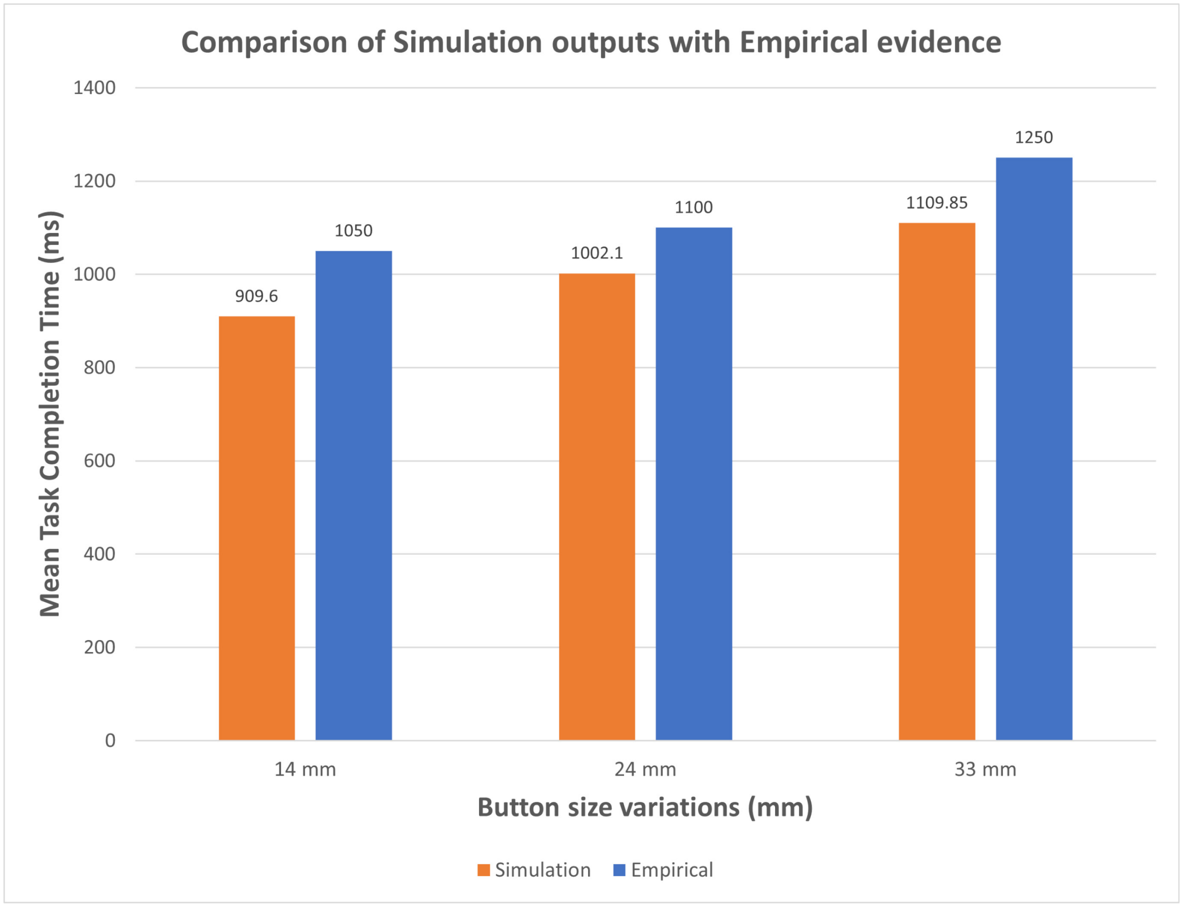


Figure 5-3: Comparison of simulation outputs with empirical evidence

Chapter 6 Discussion

6.1 Summary

This dissertation is targeted at the development of an open-source application using Python/SimPy and create a QN-MHP framework to model human performance for a simple button press task. Chapter 1 is structured to present the current state and practice of computational human performance modeling and open-source platforms. Chapter 2 discussed the existing literature that contributes towards the research of HPM's with special attention towards the QN-MHP model, Python programming language, and SimPy library and their respective benefits. In Chapter 3, the constructive step of creating a discrete event simulation using SimPy to model a two-server network has been discussed which provided the base towards the QN-MHP implementation. Through Chapter 4, the structure, steps, and logic of the QN-MHP implementation with limited servers targeted towards modeling a simple button press task have been elaborated. The implemented QN-MHP is validated and verified using empirical results from existing literature and has been proven to be effective with the total task completion time falling in the range of the expected values which was demonstrated in Chapter 5. Finally, Chapter 6 will be discussing applications of this research and the areas it could potentially benefit along with ideas and plans of future research enhancements for this project.

6.2 Applications

Plenty of industries consists of situations where a human interacts or communicates with some kind of device or machine at the same time does multitasking. It is crucial to research and understand the mechanism of human cognitive processing which handles numerous tasks with a splendid potential to handle the different interference of information to best design the User Interface of the device. Though, there is plenty of existing research and literature that talks about model human processors and their impact in research of similar applications, there are only few that provide a platform to access a QN-MHP free of cost without the dependency of any commercial license.

One of the strong motivations for this thesis is ACT-R, a famous cognitive architecture, which is implemented as an open-source software. On the other hand, though QN-MHP is celebrated in various literature, it is only available to be accessed through licensed software (e.g., Simulink in MATLAB) which is not available for everyone to access free of cost. Hence, this thesis serves as a step that tries to prove the possibility of developing such software using Python, eventually making it an open-source project, where contributors around the world can add more functionalities and use it for research plenty of applications like testing human performance when using mobile device interface, an automobile dashboard digital screen, flying cars or cockpit interface testing. These areas not only provide strong support in understanding the impact of the interface well before it is implemented for use but also play a crucial hand in keeping humankind safe from any serious fatalities.

Through this thesis project, I developed a computational model with the QN-MHP as its core structure to test a basic button press task. I also validated the performance of the developed model with existing empirical evidence, and it has proved to have a strong similarity with them. This provides the necessary confidence to say that the model if upgraded to complex tasks would be highly beneficial for user interface designers to investigate various complex design spaces and understand the usability issues that the user might face at anearly stage of the design phase eventually incurring a minimal cost and saving more time and human subject involvement.

Apart from these, this project could create a significant impact towards using the SimPy library of Python for simulating model human processors which is the rare or even the first application of SimPy in this area of research so far. Through, the implementation of the graphical user interface for the simulation, I was able to showcase the possibility of demonstrating a complete simulation package which when improved could match or even exceed the expectations from the existing commercial software.

6.3 Future Research

- Currently, the QN-MHP implement in Python using SimPy only simulates a basic button press task. The modeled button press task has a preset single target button which is the only stimulus that the network receives. Future developments of adding additional complexities like adding multiple buttons on the screen as well as introducing a visual search pattern for the model to follow to navigate between buttons.
- A more defined user interface that shows a graphical representation of the button being clicked could add more value to visualize the simulation better. It is also planned as a future effort to include a specific window in UI to let the designer input various customizable configuration parameters rather than having them as static values in a configuration file. This would provide more flexibility to the application by making it suitable for plenty of applications. This is to be a value add to provide a more realistic simulation to the designer.
- This thesis project has selective servers implemented in the cognitive and motor subnetwork that was necessary to model the planned button press task. Future more investigation and implementation of the rest of the servers from the actual QN-MHP

version by Liu et al., (2006) could contribute towards multimodal interactions and serves several task components and cognitive operators to test multiple scenarios.

Appendices

Appendix A

Attributes of custom Python class `Entity`

Attributes	Description
`id`	A unique id number set for any new entity created
`sub_id`	An integer field that is used to track when the entity is split or cloned into two parts (a) and (b) at Server 2.
`name`	A string field used in log messages to denote the respective entity. (e.g., Entity 1, Entity 2, etc.)
`current_goal`	An integer that possesses the unique id number of the goal with which the entity is associated.
`server_2_process_completed`	A boolean field that is set to *True* when the processing of the entity is completed in Server 2.
`server_3_process_completed`	A boolean field that is set to *True* when the processing of the entity is completed in Server 3. Only when the server_2_process_completed and server_3_process_completed attributes of an entity are *True* the entity will be eligible to enter Server 4. Until which either of the parts wait in Server 4 queue.
`server_4_process_completed`	A boolean field that is set to *True* when the processing of the entity is completed in Server 4.
`label`	A string field that holds the label of the target button (in this case 'USB')
`shape`	A string field that holds the shape of the target button (Default: 'square')
`size`	An integer field that holds the size of the target button.
`color`	A string field that holds the shape of the target button (Default: 'gray')
`x_coordinate`	An integer field that would hold the x-coordinate of the target button's location.
`y_coordinate`	An integer field that would hold the y-coordinate of the target button's location.
`z_coordinate`	An integer field that would hold the z-coordinate of the target button's location.
`goal_accomplished`	A boolean field that is set to *True* when the goal associated with the entity is accomplished successfully, after which the simulation ends.

Appendix B

Attributes of custom Python class `TrackingVariable`

Attributes	Description
`inital_entity`	An integer field that holds the id of the initial entity. It is used to filter any other entity for the same associated goal from entering the network if the initial entity is already being processed.
`Label`	A string field that holds the label of the target stimulus (in this case 'USB')
`Shape`	A string field that holds the shape of the target stimulus (Default: 'square')
`Size`	An integer field that holds the size of the target stimulus. This is set from the button selected in the Target configuration wizard during the start of the simulation.
`Color`	A string field that holds the shape of the target stimulus (Default: 'gray')
`x_coordinate`	An integer field that would hold the x-coordinate of the target stimulus's location. (Default: 100)
`y_coordinate`	An integer field that would hold the y-coordinate of the target stimulus's location. (Default: 100)
`z_coordinate`	An integer field that would hold the z-coordinate of the target stimulus's location. (Default: 100)

Appendix C

Attributes of custom Python class `Stimulus`

Attributes	Description
`stimulusID`	An integer field that holds the id of the initial entity. It is used to filter any other entity for the same associated goal from entering the network if the initial entity is already being processed.
`entity_goal`	An integer field that holds the id of the goal associated with the initial stimulus entity that entered the network from the modeled task.
`goal_accomplished`	A boolean field that is set to *True* when the goal associated with the initial entity is accomplished successfully, after which the simulation ends.

Appendix D

Sample outputs of results for simulation with target button size selected as ’14 mm’

Simulation_Time	Log	PN_Utilization	CN_Utilization	MN_Utilization	QN_Utilization
1		0.125	0	0	0.041667
2		0.125	0	0	0.041667
3		0.125	0	0	0.041667
4		0.125	0	0	0.041667
5		0.125	0	0	0.041667
.		.	.	.	.
.		.	.	.	.
.		.	.	.	.
1083		0	0	0.066667	0.022222
1084		0	0	0.066667	0.022222
1085		0	0	0.066667	0.022222
1086		0	0	0.066667	0.022222
1087	GOAL ACCOMPLISHED- Entity 1 pressed the "USB" button(Size: 14mm).	0	0	0.066667	0.022222

Sample outputs of results for simulation with target button size selected as ’24 mm’

Simulation_Time	Log	PN_Utilization	CN_Utilization	MN_Utilization	QN_Utilization
1		0.125	0	0	0.041667
2		0.125	0	0	0.041667
3		0.125	0	0	0.041667
4		0.125	0	0	0.041667
5		0.125	0	0	0.041667
.		.	.	.	.
.		.	.	.	.
.		.	.	.	.
1004		0	0	0.066667	0.022222
1005		0	0	0.066667	0.022222
1006		0	0	0.066667	0.022222
1007		0	0	0.066667	0.022222
1008	GOAL ACCOMPLISHED- Entity 1 pressed the "USB" button(Size: 24mm).	0	0	0.066667	0.022222

Sample outputs of results for simulation with target button size selected as ’33 mm’

Simulation_Time	Log	PN_Utilization	CN_Utilization	MN_Utilization	QN_Utilization
1		0.125	0	0	0.041667
2		0.125	0	0	0.041667
3		0.125	0	0	0.041667
4		0.125	0	0	0.041667
5		0.125	0	0	0.041667
.		.	.	.	.
.		.	.	.	.
.		.	.	.	.
948		0	0	0.066667	0.022222
949		0	0	0.066667	0.022222
950	Entity 20 generated	0	0	0.066667	0.022222
951		0	0	0.066667	0.022222
952	GOAL ACCOMPLISHED- Entity 1 pressed the "USB" button(Size: 33mm).	0	0	0.066667	0.022222

Appendix E

Simulation results for 20 times for each of the button size variations and its statistical test values (all units are in milliseconds)

Trials	14 mm	24 mm	33 mm
1	1087	1084	928
2	1111	958	952
3	1085	976	916
4	1096	977	968
5	1114	963	925
6	1123	1027	959
7	1063	1016	978
8	1087	1002	868
9	1043	999	876
10	1149	1008	875
11	1113	998	964
12	1075	1027	897
13	1104	976	865
14	1097	1030	941
15	1133	1049	840
16	1128	979	914
17	1233	1056	870
18	1115	986	911
19	1143	997	887
20	1098	934	858
Mean	1109.85	1002.1	909.6
S.D	39.0428577	36.2794708	41.4860788
Min	1233	1084	978
Max	1043	934	840
50th Percentile	1107.5	998.5	912.5
95th Percentile	1153.2	1057.4	968.5

Appendix F

Server processing time calculation based on exponential distribution of corresponding minimum and average subnetwork cycle time

1. Perceptual processing time calculation

```
# Wait for one perceptual processing cycle (adjustable parameter, PPT) while
object characteristics are copied into the entity tracking memory
mean = config.Avg_PPT - config.Min_PPT;
shift = config.Min_PPT;
np.random.default_rng(config.RANDOM_SEED)
seed = np.random.rand()
process_time = (-1) * mean * np.log(seed) + shift;
yield self.env.timeout(process_time)
```

2. Cognitive processing time calculation

```
# Wait for one cognitive processing cycle (adjustable parameter, CPT)
mean = config.Avg_CPT - config.Min_CPT;
shift = config.Min_CPT;
np.random.default_rng(config.RANDOM_SEED)
seed = np.random.rand()
process_time = (-1) * mean * np.log(seed) + shift;
yield self.env.timeout(process_time)
```

3. Motor processing time calculation

```
# Wait for one motor processing cycle (adjustable parameter, MPT)

mean = config.Avg_MPT - config.Min_MPT;

shift = config.Min_MPT;

np.random.default_rng(config.RANDOM_SEED)

seed = np.random.rand()

process_time = (-1) * mean * np.log(seed) + shift;

yield self.env.timeout(process_time)


# config.Avg_PPT, config.Min_PPT , config.Avg_CPT, config.Min_CPT

config.Avg_MPT, config.Min_MPT, config.RANDOM_SEED (refer to Appendix I)
```

Appendix G

Calculation of movement time for right-hand to reach the target button based on Fitts's Law in Server Z

```
def hand_movement(self, entity):
"""Calculation of time taken for the right-hand index finger
to reach the button based on Fitts’s Law"""

# Calculations are based on values from Feng, 2015.

a = 230 # Empirical constraint in msec
b = 166 # Difficulty index in msec/bit

error_rate_fitts = 0.04 # Based on z_score_table of error rate
zScore_fitts = 2.0537 if error_rate_fitts <= 0.04 else 1.6449

error_rate = 0.04 # Based on z_score_table of error rate
zScore = 2.0537 if error_rate <= 0.04 else 1.6449

# The implementation is based on the assumption of the target button to be a
square.
# Hence one side of the square is considered as size
button_size = entity.size

effective_size = button_size * zScore_fitts / zScore

ID = np.log2(entity.amplitude / effective_size + 1)

mouse_clicking_time = 260

motor_prep_time = 70 # motor preperation time from MHP (motor cycle typical
value)

#According to Fitt's Law MovementTime = a + b * ID - mouseClickingTime
movement_time = a + b * ID - mouse_clicking_time - motor_prep_time

return movement_time
```

Appendix H

System and project setup instructions

```
# queuing-network-sim
Description: Implementation of basic queuing network simulation

#System Details
OS: Windows/Mac

#Development Setup
1. Install anaconda3
https://www.anaconda.com/products/individual#windows
(for more details on anaconda visit https://docs.anaconda.com/anaconda/)
** Installed Python version 3.8.5**
2. Open Anaconda Navigator from Start Menu
3. Launch Spyder (For this project I am using Spyder. Please feel free to choose the
IDE of your choice)
4. Open Anaconda Prompt from Start Menu and execute the following commands:
conda update anaconda
conda install spyder=4

#Library Installation
1. Simpy - (version- 4.0.1)
In command prompt/anaconda prompt execute the following command
to install Simpy library:
pip install simpy
       (Should have pip already installed which comes by default with Python)
```

Appendix I

QN-MHP Configuration attributes and their values

Configuration Attributes	Values	Description
`SIM_TIME`	1100 ms	Arbitrary maximum simulation run time value for the x-axis of plots
`T_INTER`	50 ms	Interarrival rate between entities in the stimulus streams [50 ms]
`Min_PPT`	25 ms	The minimum exponentially distributed perceptual processing time for one memory cycle
`Avg_PPT`	42 ms	The average exponentially distributed perceptual processing time for one memory cycle
`Min_CPT`	6 ms	The minimum exponentially distributed cognitive processing time for one memory cycle
`Avg_CPT`	18 ms	The average exponentially distributed cognitive processing time for one memory cycle
`Min_MPT`	10 ms	The minimum exponentially distributed motor processing time for one memory cycle
`Avg_MPT`	24 ms	The average exponentially distributed motor processing time for one memory cycle
`PERCEPTUAL_SERVERS`	A list of four integers [1, 2, 3, 4] denoting the four servers	An array that includes the notations for implemented servers in the perceptual subnetwork. [Future implementation: can be updated with other servers implemented for utilization calculation]
`COGNITIVE_SERVERS`	A list of two strings 'A' and 'C' denoting the two servers	An array that includes the notations for implemented servers in the cognitive subnetwork. [Future implementation: can be updated with other servers implemented for utilization calculation]
`MOTOR_SERVERS`	A list of three strings 'W', 'Y' and 'Z' denoting the three servers	An array that includes the notations for implemented servers in the motor subnetwork. [Future implementation: can be updated with other servers implemented for utilization calculation]
`SERVER_1_CAPACITY`	10000	Maximum number of entities allowed to process in Server 1 at a given time
`SERVER_2_CAPACITY`	4	Maximum number of entities allowed to process in Server 2 at a given time
`SERVER_3_CAPACITY`	4	Maximum number of entities allowed to process in Server 3 at a given time
`SERVER_4_CAPACITY`	5	Maximum number of entities allowed to process in Server 4 at a given time

`SERVER_A_CAPACITY`	4	Maximum number of entities allowed to process in Server A at a given time
`SERVER_C_CAPACITY`	3	Maximum number of entities allowed to process in Server B at a given time
`SERVER_W_CAPACITY`	1	Maximum number of entities allowed to process in Server W at a given time
`SERVER_Y_CAPACITY`	2	Maximum number of entities allowed to process in Server Y at a given time
`SERVER_Z_CAPACITY`	5	Maximum number of entities allowed to process in Server Z at a given time
`RANDOM_SEED`	42	Arbitrary random seed value set to the exponential distribution calculation of all the server processing time. It is set to help reproduce the results of the simulation.
`TARGET_BUTTON_SIZES`	A list of three integers [14, 24, 33] denoting the size in mm.	The different sizes of the target button that are considered for validation in the simulation.

Appendix J

QN-MHP Global variables

Configuration Attributes	Description
`ENTITY_TRACKING_ARRAY`	A Python data array that collects a list of objects created for the custom Python class *Entity* (see Appendix A) throughout the simulation
`TRACKING_VARIABLE`	An object of the *TrackingVariable* (see Appendix B) custom Python class that tracks the global state of the simulation
`SIM_DATAFRAME`	A pandas data frame (Pandas, 2021), which is a form of a database table, that is used to store the simulation time, logs, utilization of all the subnetworks, and the entire queuing network. This data frame is finally used to create an output excel file to the results folder. (See sample outputs for each button size in Appendix D)
`LOGS`	An array of objects that stores the simulation time and log message that is printed for every update in the simulation.
`STIMULUS`	An object of the custom Python *Stimulus* (see Appendix C) class that stores the values of the target stimulus of the simulation

References

ACT-R. (2021). *ACT-R*. http://act-r.psy.cmu.edu/

Bi, X., Li, Y., & Zhai, S. (2013). FFitts law: Modeling finger touch with Fitts' law. *Conference on Human Factors in Computing Systems - Proceedings*, 1363–1372.

Card, S. K., Moran, T. P., & Newell, A. (1986). The model human processor: An engineering model of human performance. In K. R. Boff, L. Kaufman, & J. P. Thomas (Eds.), *Handbook of Perception and Human Performance, Vol. 2. Cognitive Processes and Performance* (pp. 1–35). John Wiley & Sons. (Reprinted with revisions from "The Psychology of Human-Computer Interaction," Hillsdale, NJ: Lawrence Erlbaum Associates, 1983)

Dagkakis, G., Heavey, C., Robin, S., & Perrin, J. (2013). ManPy: An open-source layer of DES manufacturing objects implemented in SimPy. *2013 8th EUROSIM Congress on Modelling and Simulation*, 357–363. https://doi.org/10.1109/EUROSIM.2013.70

Dahl, O.-J., & Nygaard, K. (1966). SIMULA: An ALGOL-based simulation language. *Communications of the ACM*, *9*(9), 671–678. https://doi.org/10.1145/365813.365819

Downey, A. B. (2017). *Modeling and Simulation in Python: Version 3.4.3*. Green Tea Press.

Feng, F., Liu, Y., & Chen, Y. (2017). A computer-aided usability testing tool for in-vehicle infotainment systems. *Computers & Industrial Engineering*, *109*, 313–324. https://doi.org/10.1016/j.cie.2017.05.019

Feng, F., Liu, Y., & Chen, Y. (2018). Effects of quantity and size of buttons of in-vehicle touch screen on drivers' eye glance behavior. *International Journal of Human–Computer Interaction*, *34*(12), 1105–1118. https://doi.org/10.1080/10447318.2017.1415688

Feng, R. (2015). Queuing Network Modeling of Human Multitask Performance and its Application to Usability Testing of In-Vehicle Infotainment Systems. Doctoral Dissertation. Department of Industrial and Operations Engineering. University of Michigan.

Feyen, R. G. (2002). Modeling Human Performance using the Queuing Network-Model Human Processor (QN-MHP). Doctoral Dissertation. Department of Industrial and Operations Engineering. University of Michigan. https://hdl.handle.net/2027.42/129422

Feyen, R. G., & Liu, Y. (2006). The Queuing Network Model Human Processor (QNMHP): An engineering approach for modeling cognitive performance. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. https://doi.org/10.1177/154193120104502407

Gathmann, F. O. (1998). Python as a Discrete Event Simulation environment. Doctoral Dissertation. Department of Zoology. University of Toronto. 51-62. https://tspace.library.utoronto.ca/bitstream/1807/13743/1/NQ56700.pdf

GitHub Inc. (2021). *GitHub features: The right tools for the job*. GitHub. https://github.com/features

Kieras, D. (1997). A guide to GOMS model usability evaluation using NGOMSL. In M. G. Helander, T. K. Landauer, & P. V. Prabhu (Eds.), *Handbook of Human-Computer Interaction (Second Edition)*. North-Holland. 733–766. https://doi.org/10.1016/B978-044481862-1.50097-2

Liu, Y., Feyen, R., & Tsimhoni, O. (2006). Queueing Network-Model Human Processor (QN-MHP): A computational architecture for multitask performance in human-machine systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *13*(1), 34. https://doi.org/10.1145/1143518.1143520

Matloff, N. (2008). Introduction to discrete-event simulation and the SimPy language. Davis, CA. Dept of Computer Science. 2. https://web.cs.ucdavis.edu/~matloff/matloff/public_html/156/PLN/DESimIntro.pdf

Matplotlib. (2021a). *Matplotlib-backend_tkagg*. Matplotlib.Backends.Backend_tkagg — Matplotlib 3.3.4 Documentation. https://matplotlib.org/3.3.4/api/backend_tkagg_api.html

Matplotlib. (2021b). *Matplotlib-pyplot*. Matplotlib.Pyplot — Matplotlib 3.4.2 Documentation. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html

Mehlhase, A. (2014). A Python framework to create and simulate models with variable structure in common simulation environments. *Mathematical and Computer Modelling of Dynamical Systems*, *20*(6), 566–583. https://doi.org/10.1080/13873954.2013.861854

Millman, K. J., & Aivazis, M. (2011). Python for scientists and engineers. *Computing in Science Engineering*, *13*(2), 9–12. https://doi.org/10.1109/MCSE.2011.36

Pandas. (2021). *Pandas*. User Guide - Pandas. https://pandas.pydata.org/docs/user_guide/index.html

Python Software Foundation. (2021). *Tkinter—Python interface to Tcl/Tk—Python 3.9.6 documentation*. https://docs.python.org/3/library/tkinter.html

Python Software Foundation. (2021a). *Python*. General Python FAQ — Python 3.9.6 Documentation. https://docs.python.org/3/faq/general.html#general-information

Python Software Foundation. (2021b). *Python Developer's Guide*. https://devguide.python.org/#quick-reference

Simon, H. A., & Newell, A. (1971). Human problem solving: The state of the theory in 1970. *American Psychologist*, *26*(2), 145–159. https://doi.org/10.1037/h0030806

SimPy. (2021b). *Shared Resources—SimPy 4.0.2.dev1+g2973dbe documentation*. https://simpy.readthedocs.io/en/latest/simpy_intro/shared_resources.html

SimPy. (2021a). *SimPy*. Overview — SimPy 4.0.2.Dev1+g2973dbe Documentation. https://simpy.readthedocs.io/en/latest/index.html

Tsimhoni, O. (2004). Visual sampling of in -vehicle displays while driving: Empirical findings and a queueing network cognitive model. Doctoral Dissertation. Department of Industrial and Operations Engineering. University of Michigan. https://hdl.handle.net/2027.42/124322

Tsimhoni, O., & Liu, Y. (2003). Steering a driving simulator using the Queuing Network-Model Human Processor (QN-MHP). 81-85. 10.17077/drivingassessment.1100.

Wu, C. (2007). Queueing network modeling of human performance and mental workload in perceptual-motor tasks. Doctoral Dissertation. Department of Industrial and Operations Engineering. University of Michigan. https://hdl.handle.net/2027.42/55678

Wu, C., & Liu, Y. (2007). Queuing network modeling of driver workload and performance. *IEEE Transactions on Intelligent Transportation Systems*, *8*(3), 528–537. https://doi.org/10.1109/TITS.2007.903443