Leibniz Universität Hannover
Wirtschaftswissenschaftliche Fakultät
Institut für Produktionswirtschaft
Prof. Dr. Stefan Helber

Hausarbeit
im Rahmen des Moduls Scientific Computing II am Institut für Produktionswirtschaft
im SS 2022
(Veranstaltungs-Nr. 470011)

Thema Nr. I:

Simulation von Kassensystemen in Supermärkten

Mirco Oliver Höhne
[redacted]

Abgabedatum: 20.07.2022

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Abstract

Unternehmen müssen effiziente Entscheidungen treffen. Damit Entscheidungen effizient sind, müssen diese auf Grundlage von Daten getroffen werden. Die Erhebung von Daten und das Beobachten von Systemen ist sehr zeit- und kostenintensiv. Daher ist es sinnvoll, Systeme der echten Welt in Simulationsmodelle zu übertragen, um schneller durch diese iterieren zu können und so Zeit und Kosten zu sparen. In dieser Seminararbeit wurde ein Modul/Programm erstellt, das einen Supermarkt simulieren soll. Hierfür wurden die grundlegenden Elemente einer ereignisdiskreten Simulation modular implementiert, um die Verwendung des Programms und der einzelnen Klassen auch außerhalb einer speziellen Probleminstanz zu ermöglichen.

1.2 Grundlagen der ereignisdiskreten Simulation

Eine Simulation ist eine vereinfachte Nachbildung des Verhaltens eines geplanten Systems oder eines Systems aus der realen Welt.[1] Mithilfe einer Simulation soll der Prozess oder das System beobachtet werden, um die zugrundeliegenden Charakteristika des Systems, das repräsentiert wird, zu verstehen.[2] Durch das Verständnis der Charakteristika werden weitere Analysen ermöglicht, die zum Beispiel bei Entscheidungen über die Erweiterung eines Systems oder Optimierung eines Prozesses helfen können.[3]
Eine ereignisdiskrete Simulation ist eine Simulation, bei welcher sich der Zustand des Systems (hier in Form von Zustandsvariablen, z.B. Zeit) nur zu diskreten Zeitpunkten,

[1] Banks o. D., S. 663.
[2] Banks o. D., S. 663.
[3] Banks o. D., S. 663.

an denen Ereignisse stattfinden, verändern.[4] Die ereignisdiskrete Simulation besteht aus mehreren Komponenten:

- Ereignisse
- Zustandsvariablen
- Instanzen und deren Attribute
- Ressourcen
- Listen
- Aktivitäten und Verzögerungen

Ereignisse können in interne und externe Ereignisse eingeteilt werden. Interne Ereignisse sind Ereignisse, die innerhalb des Systems passieren, also beispielsweise der Prozess des Kassierens, nachdem in der Schlange gewartet wurde.[5] Externe Ereignisse sind solche, die außerhalb des Systems vorkommen, wie zum Beispiel die Ankunft eines neuen Kunden.[6]
Zustandsvariablen sind die Variablen des Modells, die alle benötigten Informationen über den Zustand des Modells beinhalten, um ausreichend zu erklären, was im Modell passiert.[7]
Instanzen sind Objekte, die einer expliziten Definition bedürfen.[8] Instanzen können sowohl dynamisch als auch statisch sein.[9] Im Kontext dieser Arbeit wäre eine dynamische Instanz ein Kunde, der sich durch das System „bewegt“ und eine statische Instanz wäre ein Kassierer, da dieser andere Instanzen bedient, die sich durch das System „bewegen“. Jede Instanz kann Attribute haben, die diese spezielle Instanz beschreiben.[10] In dieser Arbeit wären solche Attribute beispielsweise die Anzahl der Waren, die der Kunde einkaufen möchte.
Ressourcen sind Instanzen, die dynamische Instanzen bedienen.[11] Eine dynamische Instanz kann eine oder mehrere Ressourcen anfragen.[12] In dem Fall dieser Arbeit wäre das der Kunde, der als Ressource einen Kassierer an einer handelsüblichen Kasse anfordert. Für die Ressource gibt es mindestens zwei Zustände: „in Nutzung“ und „ungenutzt“.[13]

[4] Banks o. D., S. 663.
[5] Banks o. D., S. 665.
[6] Banks o. D., S. 665.
[7] Banks o. D., S. 665.
[8] Banks o. D., S. 665.
[9] Banks o. D., S. 665.
[10] Banks o. D., S. 665.
[11] Banks o. D., S. 666.
[12] Banks o. D., S. 666.
[13] Banks o. D., S. 666.

Listen werden genutzt, um die Ereignisse zu speichern, welche im System vorkommen und zu den entsprechenden Zeitpunkten den Instanzen die benötigten Ressourcen zur Verfügung zu stellen.[14] Listen werden für eine Modellierung von Warteschlangen genutzt.[15] Hier gibt es unterschiedliche Methoden die Listen zu modellieren.[16] Eine Methode wäre First in first out (FIFO), bei der die Instanz, die zuerst da war auch zu erst bedient wird.[17] Eine weitere Möglichkeit wäre Last in first out, wo jeweils die Instanz bedient wird, die zuletzt angekommen ist.[18] Neben diesen zwei Möglichkeiten gibt es noch weitere, die hier aber nicht genauer betrachtet werden.
Eine **Aktivität** ist eine Zeitspanne, die schon vor Beginn der Aktivität bekannt ist.[19] Somit kann das Ende der Aktivität geplant werden, wenn die Aktivität gestartet wird.[20] Die Zeitspanne kann eine Konstante, ein randomisierter Wert aus einer statistischen Verteilung oder auch ein berechneter Wert sein.[21] **Verzögerungen** sind undefinierte Zeitspannen, die durch eine Kombination von Zuständen im System entstehen.[22] Wenn sich eine Instanz in eine Schlange stellt, dann ist möglicherweise unbekannt, wie lange die Instanz in der Schlange verbringt, da dies von anderen Ereignissen im System abhängt.[23] Der Anfang und das Ende einer Aktivität oder Verzögerung ist ein Ereignis.[24]

Mithilfe dieser Komponenten kann eine ereignisdiskrete Simulation erstellt werden. In der ereignisdiskreten Simulation verändern sich Zustandsvariablen nur an den diskreten Zeitpunkten, an denen Ereignisse stattfinden.[25] Ereignisse entstehen als eine Konsequenz aus Aktivitätszeitspannen und Verzögerungen.[26] Instanzen konkurrieren untereinander um Ressourcen und stellen sich gegebenenfalls in Warteschlangen und warten auf freigegebene Ressourcen.[27] Eine ereignisdiskrete Simulation läuft über Zeit mit einem Mechanismus, der die Zeit voranschreiten lässt.[28] Bei jedem Ereignis werden die Zustandsvariablen aktualisiert und gegebenenfalls Ressourcen allokiert beziehungsweise freigegeben.[29]

[14] Banks o. D., S. 666.
[15] Banks o. D., S. 666.
[16] Banks o. D., S. 666.
[17] Banks o. D., S. 666.
[18] Banks o. D., S. 666.
[19] Banks o. D., S. 666.
[20] Banks o. D., S. 666.
[21] Banks o. D., S. 666.
[22] Banks o. D., S. 666.
[23] Banks o. D., S. 666.
[24] Banks o. D., S. 666.
[25] Banks o. D., S. 666.
[26] Banks o. D., S. 666.
[27] Banks o. D., S. 666.
[28] Banks o. D., S. 666.
[29] Banks o. D., S. 666.

1.3 Modellierung der Problemstellung

Es soll ein Supermarkt modelliert werden. Hierfür ist es notwendig, die vorher definierten Bestandteile des Modells für dieses Problem zu definieren.
In diesem Modell gibt es genau zwei Ereignisse. Ein Ankunftsereignis und ein Weggangsereignis. Dynamische Instanzen sind die Kunden und statische Instanzen sind die Kassen mit der Ressource der Kassierer beziehungsweise dem Selbstbedienungsterminal an Selbstbedienungskassen.
Um die Warteschlangen zu simulieren wird für jede einzelne Warteschlange eine sortierte Liste geführt, die eine entsprechende Schlange vor der Kasse darstellt. Zusätzlich gibt es eine Liste für die Bedienung des Kunden, in die diese transferiert werden, sobald sie an der Reihe sind. Dies ist wichtig um einen Selbstbedienungskassenblock mit mehreren Kassen zu simulieren, da dort eine parallele Bedienung der Kunden stattfindet, wenn man einen solchen Kassenblock als eine Instanz sieht. Schlussendlich gibt es noch eine geordnete Liste, die Ereignisse nach der Zeit des Auftretens sortiert und mithilfe derer die Ereignisse nacheinander abgearbeitet werden können und somit ein Mechanismus implementiert werden kann, der die Liste durchgeht und Ereignisse abarbeitet und die Zeit voranschreiten lässt. Die Warteschlangen werden, wie im Supermarkt üblich, nach dem FIFO, also first in first out Prinzip bearbeitet, welches in Kapitel 1.2 erklärt wurde.
In dem Modell werden folgende Annahmen getroffen:

- Finite Population, das heißt die Ankunftsrate hängt nicht von der Anzahl der wartenden Kunden ab
- Zwischenankunftszeiten sind exponential verteilt
- Kunden wählen Kasse, welche die kürzeste Warteschlange hat und bei gleicher Länge wird zufällig gewählt
- Die Bearbeitungszeit pro Artikel wird für jeden Kunden aus der Exponentialverteilung oder ermittelten Verteilungen aus den POS Daten[30] generiert.
- unbegrenzte Warteschlangenkapazität
- kein Balking, also keine Kunden, die sich nicht an eine Kasse anstellen
- kein Reneging, also kein Verlassen der Schlange bevor der Kunde bedient wurde
- kein Jockeying, also kein Wechseln der Warteschlange durch den Kunden
- Die Anzahl der Kassen bleibt während der Simulation gleich

[30] Antczak und Weron 2019.

Das Modell des Supermarktes ist Abbildung 1 zu entnehmen.

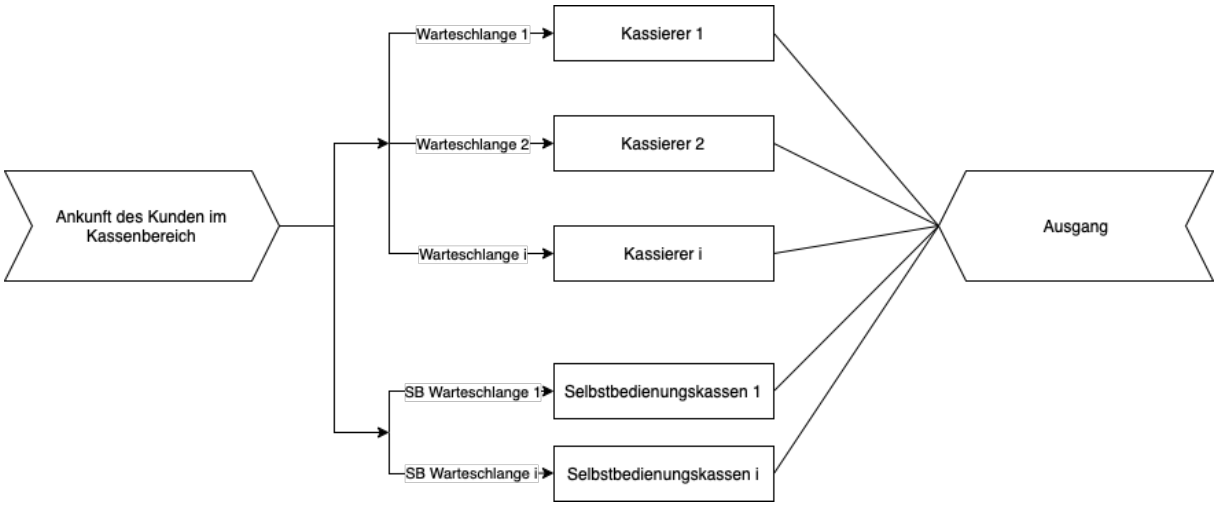

Abbildung 1 Visualisierung des Modells

2 Dokumentation

2.1 Design

2.1.1 Grundlegende Entscheidungen

Um die Zusammenarbeit und Erweiterung des Programms auch im internationalen Kontext zu ermöglichen, wurden die Variablennamen im Programm und auch die Kommentare und DocStrings auf Englisch geschrieben.
Für das Programm wurde die rein objektorientierte Programmierung als Programmierparadigma gewählt. Dies hat Vor- und Nachteile. Vorteil ist, dass dieses Paradigma intuitiv zu diesem Problem passt. Die Simulation wird als Objekt dargestellt, dessen Zustände sich durch Ereignisse verändern. Großer Nachteil ist hier allerdings, dass das Programm, wenn es um viele Module erweitert wird, schwer nachzuvollziehen wird und es zu Nebeneffekten kommen kann.[31]
Es wurden einige hilfreiche Module für die Erstellung des Programms genutzt. Diese werden im folgenden aufgeführt. Um schnell mathematische Berechnungen durchzuführen wurde das Modul Numpy verwendet. Für die Datenanalyse wurde das Pandas Modul genutzt. Um Visualisierungen zu erstellen, wurde die Matplotlib Implementierung von Pandas und Seaborn genutzt. Um Verteilungen aus Daten abzuleiten, wurde das FITTER Modul in Python genutzt. Für längere Simulationen wurde mittels dem tqdm Modul auch eine Fortschrittsanzeige implementiert. Als IDE für die Erstellung des

[31] Stefik et al. 1986, S. 10.

Programms wurde PyCharm genutzt und für die Formatierung des Programmcodes der Code Formatierer Black.
Grundlage für das Programmieren sind die Datenstrukturen. Im folgenden wird auf die Datenstrukturen eingegangen, die im Programm genutzt wurden. Allerdings werden nicht die standardmäßig vorhandenen Datenstrukturen in Python (Lists, Sets, Tuples, Dictionaries) behandelt, sondern nur jene, die von diesen abweichen und es wird begründet, warum diese genutzt wurden.

2.1.2 Heap Queue

Während der Simulation werden dynamisch Ereignisse generiert, die in eine bestehende Liste mit Ereignissen nach der Zeit einsortiert werden müssen, damit diese in der richtigen Reihenfolge abgearbeitet werden können. Hier bietet es sich an eine Liste zu nutzen, die generierten Ereignisse anzuhängen und dann mit der Methode .sort() zu sortieren. Python nutzt hier den Timsort Algorithmus.[32] Eine performantere Alternative für die Implementierung ist hier ein Binärer Heap anstelle einer Liste. Beim Einfügen wird das Objekt direkt anhand eines Wertes einsortiert. Der Trade-off ist hier, dass der binäre Heap nicht stabil sortiert. Um eine stabile Sortierung zu gewährleisten wurde ein zweiter Parameter, nämlich die einzigartige Identifikationsnummer, eingefügt. Mit dem heapq Modul von Python ist der Heap direkt so implementiert, dass nach dem zweiten Wert sortiert wird, sollte der erste gleich sein. Somit ist eine stabile Sortierung garantiert. Im Programm sind alle Listen mithilfe des heapq Moduls von Python als Heap implementiert. Heapq erstellt einen Minimum Heap, das heißt der kleinste Wert steht an erster Stelle der Liste. Sortiert werden die Listen nach Zeit und wenn diese bei zwei Objekten identisch ist, dann wird nach dem zweiten Wert, der Identifikationsnummer, sortiert.

2.1.3 Slots

Slots ist eine Klassenvariable. Diese Klassenvariable sorgt dafür, dass Datenelemente explizit deklariert werden und verhindern die Erstellung von Dictionaries. Dies führt dazu, dass der Speicher für die entsprechende Klasse nicht dynamisch allokiert wird. Somit wird weniger Speicher verbraucht und auch die Geschwindigkeit der Abfrage von Attributen dieser Klasse kann signifikant verringert werden.[33]

2.1.4 Dataclasses

Um das Programm so abstrakt wie möglich zu halten, wurden die Prozesse abstrahiert und in Klassen verpackt. Dies garantiert, dass entsprechende Klassen auch außerhalb

[32] Peters 2002.
[33] *3. Data model — Python 3.10.5 documentation* 2022.

des Programms für andere Probleminstanzen genutzt werden können. Es wurden Klassen für Ereignisse, Kunden, Kassen und für die Simulation erstellt. Für die Klassen Event, Customer und Checkout, die nur Daten speichern sollen, wurden Dataclasses verwendet. Dataclasses sind eine in Python 3.7 eingeführte Möglichkeit, das Erstellen von Klassen, die nur Daten speichern, zu automatisieren. Im Hintergrund werden Methoden (wie zum Beispiel die __repr__ Methode) automatisch generiert und müssen nicht explizit im Code geschrieben werden.[34] Dies erhöht die Übersichtlichkeit und sorgt für deutlich weniger Code.

2.2 Implementierte Datenklassen

2.2.1 Event

Diese Klasse beschreibt die einzelnen Ereignisse. Nachdem die Ereignisse erstellt wurden, können die Attribute nicht verändert werden (Frozen=True). Jedes Ereignis hat fünf Attribute.
Als erstes bekommt jedes Ereignis eine einzigartige Identifikationsnummer zugewiesen. Diese ist für eine sichere Sortierung in der Ereignisliste notwendig, damit bei gleicher Zeit das Ereignis gewählt wird, das zuerst generiert wurde. Darüber hinaus speichert die Klasse Event, zu welcher Zeit das Ereignis stattfindet und um welchen Typ von Ereignis es sich handelt. Implementiert sind hier die Typen Ankunftsereignisse und Weggangsereignisse. Schlussendlich werden in dieser Klasse auch der Kunde an sich (Objekt der Klasse Customer) und die Identifikationsnummer der Kasse (Objekt der Klasse Checkout), an der das Ereignis stattfindet, gespeichert.
Der Code für die Klasse Event findet sich im Anhang unter Listing 1 .

2.2.2 Customer

Die Customer Klasse beschreibt die einzelnen Kunden und speichert fünf Attribute.
Auch hier hat jedes Objekt der Klasse, also jeder einzelne Kunde, eine einzigartige Identifikationsnummer um die stabile Sortierung zu garantieren. Des weiteren werden die Ankunftszeit des Kunden an der Schlange einer Kasse, die Bearbeitungszeit an der Kasse und die Zeit des Weggangs in dieser Klasse gespeichert. Zusätzlich werden für jeden Kunden auch die Anzahl an Waren, die gekauft werden und die aus der Verteilung gezogene Bearbeitungszeit pro Artikel des Kunden hier gespeichert. Schlussendlich wird hier auch die Identifikationsnummer der Kasse gespeichert, an der der Kunde steht.
Der Code für die Klasse Customer befindet sich im Anhang unter Listing 2 .

[34] *dataclasses — Data Classes — Python 3.10.5 documentation* 2022.

2.2.3 Checkout

Die Checkout Klasse beschreibt die einzelnen Instanzen der Kassen und speichert sieben Attribute.
Auch die Objekte der Klasse Kasse haben eine einzigartige Identifikationsnummer. Des weiteren wird der Typ einer Kasse gespeichert. Implementiert sind die Typen Selbstbedienungskasse und Kasse mit Kassierer, also die handelsübliche Kasse. Natürlich wird auch die Anzahl der Kunden, die an einer Kasse gleichzeitig bedient werden können, gespeichert. Diese Anzahl kann für Selbstbedienungskassen und handelsübliche Kassen gesondert eingestellt werden. Zusätzlich wird auch der Status einer Kasse gespeichert, der anzeigt ob eine Kasse belegt oder frei ist. Des weiteren wird hier noch eine Schlange als Liste gespeichert in der sich die Objekte der Klasse Customer befinden, die aktuell in der Schlange stehen. Zu guter Letzt gibt es noch eine weitere Liste, in der sich die Objekte befinden, die aktuell an einer Kasse bedient werden. Dies ist zum einen für die Selbstbedienungskassen nötig, ermöglicht aber auch Situationen zu modellieren, in denen zwei Kassen direkt hintereinander liegen und eine gemeinsame Warteschlange bedienen können.
Der Programmcode findet sich im Anhang unter Listing 3 .

2.3 Simulationsklasse

2.3.1 Überblick

Die Klasse Simulation ist die Implementierung einer Simulation als Klasse. Diese Klasse hat sehr viele Parameter, da hier auch viele Methoden implementiert sind. Insgesamt wird diese Klasse mit 14 Variablen initialisiert. Ausführlichere Beschreibungen jeder einzelnen Variable finden sich im Programmcode.
Da diese Klasse ein Simulationsmodell und dessen Verhalten widerspiegelt, hat diese Klasse einige Methoden, welche die Zustände des Modells verändern.

2.3.2 __init__(...)

In der Initialisierungsfunktion werden die Zustandsvariablen des Systems übergeben. Hier wird ein Seed für die Erzeugung von pseudo-randomisierten Werten übergeben um Reproduzierbarkeit zu gewährleisten. Darüber hinaus wird hier die maximale Dauer der Simulation übergeben und sich für Verteilungen entschieden (entweder die aus POS Daten generierten, siehe 3.1 , oder stattdessen Exponentialverteilungen für generelle Anwendung). Für alle Verteilungen können hier die Werte der Parameter einzeln übergeben werden und die Anzahl an Selbstbedienungs- beziehungsweise herkömmlichen Kassen können hier ebenfalls übergeben werden. Darüber hinaus können hier auch die Kapazitäten der Kassentypen übergeben werden.

Mithilfe dieser übergebenen Werte wird dann die Simulation initialisiert. Die Zeit wird auf null gesetzt, es werden die entsprechenden Listen und Logdateien erstellt und es wird ein Dictionary mit den Kassenobjekten, für einen leichteren Zugriff auf diese, generiert.

2.3.3 update_ql(self)

Diese Methode aktualisiert die Liste der Warteschlangenlängen und schreibt diese in die Logdatei für die Warteschlangenlänge.

2.3.4 get_arrival(self)

Diese Methode generiert eine Ankunftszeit, einen neuen Kunden und die Anzahl an Waren des Kunden. Wenn der Kunde erstellt ist wird die kürzeste Warteschlange gesucht und diese dem Kunden zugewiesen. Dann wird ein Ankunftsereingnis erstellt und dies wird der Ereignis-Liste hinzugefügt.

2.3.5 arrival(self)

Die Ankunftsmethode zieht sich das nächste Ereignis, welches ein Ankunftsereignis ist, aus der Ereignis-Liste und schreibt es zuerst in die Logdatei für Ereignisse und update_ql() wird aufgerufen. Wenn die Warteschlange leer ist und Ressourcen vorhanden sind, so wird der Kunde direkt bedient und in die Bearbeitungsliste hinzugefügt. Die Bearbeitungszeit pro Artikel, die Gesamtbearbeitungszeit und Zeit des Weggangs werden berechnet. Ein entsprechendes Weggangsereignis wird der Ereignisliste hinzugefügt. Wenn die Warteschlange nicht leer ist wird der Kunde in die Warteschlangenliste hinzugefügt. Am Ende der Methode wird ein Ankunftsereignis mit get_arrival() aufgerufen.

2.3.6 departure(self)

Die Weggangsmethode zieht sich das nächste Ereignis, welches ein Weggangsereignis ist, aus der Liste der Kunden, die an dieser Kasse gerade bearbeitet werden und schreibt es in die Logdatei für Ereignisse und setzt den Status der Ressource auf „ungenutzt“.
Wenn es Kunden in der Warteschlange gibt, wechselt der erste in der Warteschlangenliste in die Bearbeitungsliste und wird somit bedient. Die Bearbeitungszeit pro Artikel, gesamte Bearbeitungszeit und die Zeit des Weggangs wird berechnet. Es wird ein entsprechendes Weggangsereignis der Ereignis Liste hinzugefügt. Danach wird überprüft ob die Kapazität der Ressource erschöpft ist und gegebenenfalls der Status auf „in Benutzung“ gesetzt. Zum Schluss wird ein Ankunftsereignis mit get_arrival() generiert.

2.3.7 next_action(self)

Diese Funktion überprüft um welchen Typ es sich bei dem nächsten Ereignis der Simulation handelt und ruft die entsprechende Funktion, also entweder arrival() oder departure(), auf.

2.3.8 simulate(self)

Die Simulationsmethode bringt die vorher genannten Methoden zusammen. Als erstes wird get_arrival() aufgerufen um ein initiales Ankunftsereignis zu erzeugen. In einer while Schleife wird dann wiederholt die Methode next_action() aufgerufen, bis die maximale Zeit erreicht ist. Am Ende werden die Logdateien so bearbeitet, dass sie möglichst einfach verarbeitet werden können und von der Methode ausgegeben. Es gibt drei Logs. Den customer_log, den event_log und den queue_log.

Der **customer_log** speichert für jeden Kunden die Identifikationsnummer, Ankunftszeit, Weggangszeit, Bearbeitungszeit, die Zeit vom Anstellen in der Warteschlange bis zum Weggang und die Identifikationsnummer der Kasse, an der er bedient wurde.
Der **event_log** speichert für jedes Ereignis die Identifikationsnummer, die Zeit zu der es Auftritt, die Identifikationsnummer des Kunden, der bedient wird und die Identifikationsnummer der Kasse.
Der **queue_log** speichert für jeden Zeitpunkt, an dem ein Ereignis auftritt, die Länge der Warteschlange jeder einzelnen Kasse.
Die Logdateien werden für die einfache Weiterverarbeitung alle als CSV Datei ausgegeben.

3 Simulationsstudie

3.1 Daten für das Modell

Die Daten, die für die Erstellung eines Modells genutzt werden stammen aus der Veröffentlichung von POS Daten,[35] Aus diesen Daten wurden die benötigten Daten extrahiert, um die zugrundeliegenden Verteilungen zu identifizieren und deren Parameter zu bestimmen (Der vollständige Extraktionsprozess ist im entsprechenden Jupyter Notebook, parameter_extraction.ipynb, zu finden). Um die Verteilungen zu bestimmen, wurde das FITTER Modul in das Programm importiert und genutzt. FITTER testet 80 Verteilungen aus SciPy mit unterschiedlichen Fehlermetriken, plottet diese und gibt die beste Verteilung samt den entsprechenden Parametern aus.[36] FITTER wurde genutzt,

[35] Antczak und Weron 2019.
[36] *FITTER documentation — fitter 1.4.0 documentation* 2022.

um aus den Daten die Verteilungen für die Anzahl der Artikel des Kunden, Transaktionszeit pro Artikel bei Selbstbedienungskassen und handelsüblichen Kassen zu bestimmen. Als Fehlermetrik habe ich die mittlere quadratische Abweichung oder auch Mean Squared Error (MSE) gewählt. Die mittlere quadratische Abweichung gibt in diesem Fall an, wie groß die durchschnittliche Summe der quadratischen Distanz jedes geschätzten Punktes zum tatsächlichen Punkt ist. Dies wird in folgender Formel beschrieben:

$$MSE = \frac{1}{n} * \sum_{i} (x_i - \tilde{x}_i)^2$$

$$n = \text{Anzahl der Punkte}$$

$$x_i = \text{tatsächlicher Wert des Punktes}$$

$$\tilde{x}_i = \text{geschätzter Wert des Punktes}$$

Mithilfe dieser Methode wurden folgende Ergebnisse erzielt (sämtliche Parameter sind der Übersicht halber im entsprechenden Jupyter Notebook, experiment_analysis.ipynb , zu finden): Die Anzahl der Waren eines Kunden wird am besten durch eine Exponentialverteilung beschrieben, siehe Abbildung 2 .

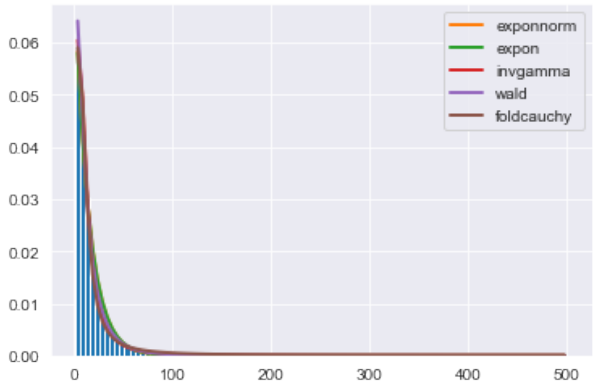

Abbildung 2 Verteilung der Anzahl der Waren
(x: Anzahl Waren, y: Wahrscheinlichkeit)

Die Bearbeitungszeit pro Artikel an handelsüblichen Kassen kann am besten mit der Laplace Verteilung beschrieben werden, siehe Abbildung 3 .
Die Bearbeitungszeit pro Artikel der Selbstbedienungskassen folgt einer rechtsschrägen Gumbel Verteilung, siehe Abbildung 4 .
Anmerkung: Die Verteilungen erlauben teilweise negative Werte. In der Implementierung des Programms ist dieses Problem so gelöst, dass so lange neue Werte generiert werden, bis ein positiver Wert ungleich null erzielt wird.
Neben den Verteilungen wurden aus den Daten zusätzlich die Anzahl der handelsüblichen Kassen und Selbstbedienungskassen ermittelt. In dem untersuchten Markt gab es 16 handelsübliche und sechs Selbstbedienungskassen.

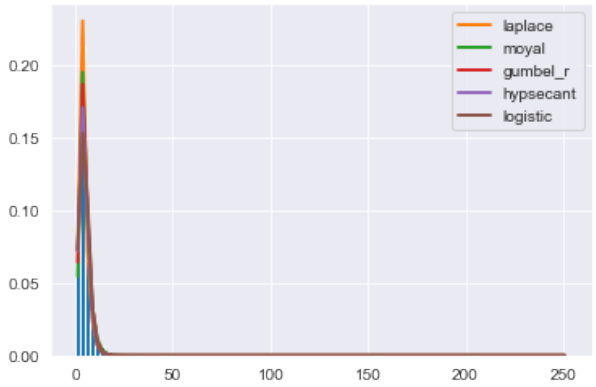

Abbildung 3 Verteilung der Bearbeitungszeit pro Artikel für handelsübliche Kassen (x: Bearbeitungszeit pro Artikel, y: Wahrscheinlichkeit)

.

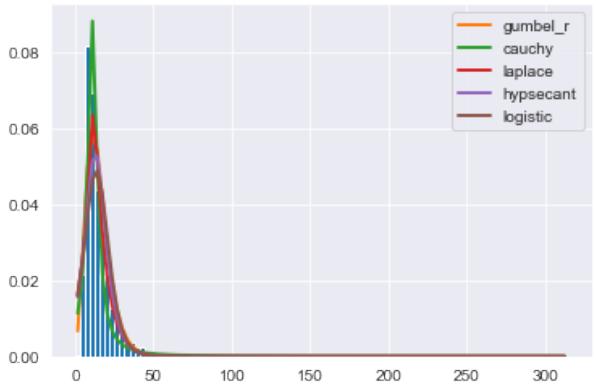

Abbildung 4 Verteilung der Bearbeitungszeit pro Artikel für handelsübliche Kassen (x: Bearbeitungszeit pro Artikel, y: Wahrscheinlichkeit)

.

3.2 Szenarioanalyse

Um die Funktionsweise des Programms zu zeigen, wurde mithilfe der analysierten Daten ein Modell erstellt, welches den Supermarkt, in dem die Daten erhoben wurden, möglichst gut repräsentiert. Das Modell wurde mithilfe der Klassen, die in Kapitel 2.2 und 2.3 vorgestellt wurden, implementiert. Es sollen unterschiedliche Szenarien durchgespielt werden. Ausgangspunkt ist ein großer Supermarkt, dessen Manager entscheiden möchte, ob es sinnvoller ist, Personal für sechs unbesetzte handelsübliche Kassen mit jeweils eigener Warteschlange einzustellen, oder diese Kassen zu sechs Selbstbedienungskassen mit einer Warteschlange umzubauen. Hierfür wurden drei Experimente durchgeführt, um die Key-Performance-Indikatoren zu vergleichen. Key-Performance-Indikatoren sind in diesem Falle die durchschnittliche Wartezeit der einzelnen Kunden und die durchschnittliche Länge der Warteschlange.

Experiment 1:
Die Parameter und Verteilungen, welche aus den Daten des Papers[37] extrahiert wurden, werden hier übernommen. Somit werden die Anzahl an Selbstbedienungskassen und handelsüblichen Kassen übernommen und auch entsprechende Verteilungen. Die genauen Daten wurden in 3.1 vorgestellt. Die gewählten Parameter im einzelnen sind dem Programmcode der Experimente zu entnehmen, welche im Anhang unter 4 zu finden sind.
Experiment 2:
Experiment 2 ist vom Aufbau analog zu Experiment 1, allerdings werden hier sechs handelsübliche Kassen mit jeweils eigener Warteschlange hinzugefügt, um zu beobachten, wie sich die Key-Performance-Indikatoren verändern.
Experiment 3:
Experiment 3 ist auch analog zu Experiment 1, allerdings wird hier ein Block mit sechs Selbstbedienungskassen erschaffen, die sich eine Warteschlange teilen.

Der Programmcode für die Implementierung der Experimente findet sich im Anhang unter Listing 4 .

3.3 Ergebnisse

Für die einzelnen Simulationen erstellt das Programm Logdateien, welche unter anderem die Key-Performance-Indikatoren speichern (siehe Abbildungen 5 , 6 und 7).

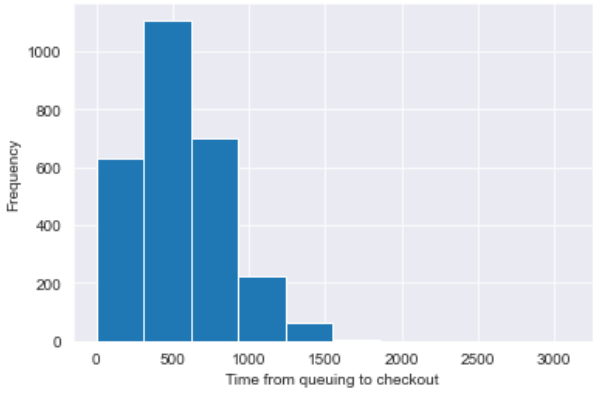

Abbildung 5 Histogramm der Wartezeit für Experiment 1

In den Histogrammen ist zu sehen, dass die Wartezeit in den Experimenten 2 und 3 im Vergleich zu Experiment 1 deutlich abnimmt. Auch im Vergleich zwischen Experiment 2 und 3 sind Unterschiede zu sehen. Um diese Unterschiede genauer zu untersuchen, wurden der Mittelwert und der Median der Wartezeit und der Warteschlangenlänge un-

[37] Antczak und Weron 2019.

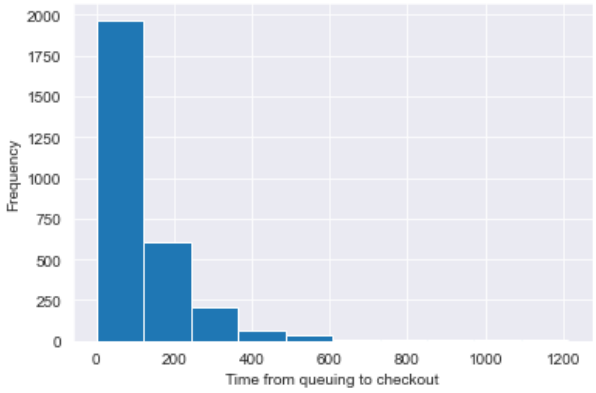

Abbildung 6 Histogramm der Wartezeit für Experiment 2

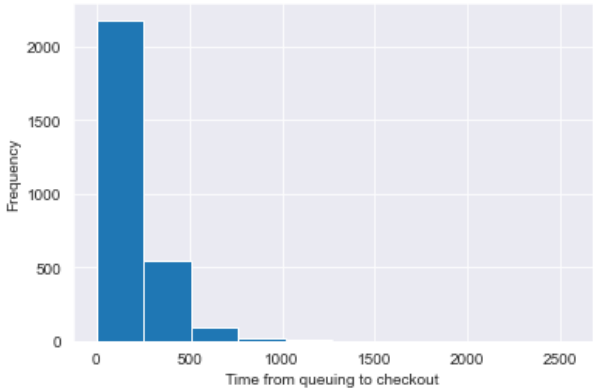

Abbildung 7 Histogramm der Wartezeit für Experiment 3

tersucht. Diese sind in Tabellen 1 und 2 zu finden.

Experiment Nr.	Mittelwert	Median
1	9,07 Minuten	8,38 Minuten
2	1,92 Minuten	1,26 Minuten
3	3,06 Minuten	2,29 Minuten

Tabelle 1 Wartezeiten (Zeit zwischen Ankunft in Schlange und Weggang von der Kasse)

Experiment Nr.	Mittelwert	Median
1	8 Kunden	8 Kunden
2	1 Kunde	1 Kunde
3	2 Kunden	2 Kunden

Tabelle 2 Warteschlangenlänge

Aus den Werten kann abgelesen werden, dass Experiment 2 und 3 eine deutlich kürzere Wartezeit und Warteschlangenlänge haben. Allerdings ist auch zu sehen, dass der Unterschied zwischen Experiment 2 und 3 nicht sehr weit auseinander liegt. Bei der Entscheidung ob Selbstbedienungskassen oder herkömmliche Kassen zu verwenden sind, ist also die Frage wichtig, wie viel ein Umbau kosten würde und auf welche Beträge sich die laufenden Personalkosten belaufen. Während an jeder handelsüblichen Kasse auch ein Kassierer sitzen muss, ist dies für Selbstbedienungskassen nicht nötig. Hier kann auch ein Mitarbeiter mehrere Kassen betreuen, da nicht jede Kasse die Hilfe eines Mitarbeiters benötigt.

4 Schlussbemerkungen

4.1 Limitationen

Das vorgestellte Szenario ist natürlich stark von den zugrundeliegenden Daten abhängig. Eine große Limitation stellen hier die Annahmen dar, die gemacht wurden. Möchte man genauere beziehungsweise realitätsnähere Daten, so müsste man entsprechende Daten selber erheben. Bei meiner Recherche konnte ich keinen Datensatz finden, der es ermöglicht hätte, alle Parameter diesen Daten anzupassen. Wenn diese Daten zur Verfügung stehen, so kann mit dem geschriebenen Programm eine realitätsnahe Auswertung erfolgen. Das Programm folgt der reinen Objektorientierung als Paradigma. Dies ist für das Schreiben des Programms sehr intuitiv, allerdings kann dies schnell unübersichtlich werden, wenn man das Programm um weitere Module erweitert. Das Programm könnte optimiert werden, indem man die Objektorientierung beispielsweise mit der funktionalen Programmierung mischt, um die Gefahr für Nebeneffekte zu verkleinern. Darüber hinaus ließen sich noch die Arbeitsschichten der einzelnen Kassierer modellieren, die hier außer Acht gelassen wurden. Man könnte auch jedem Mitarbeiter eine eigene Verteilung für die Transaktionszeit zuweisen, je nachdem wie erfahren dieser ist. Auch eine

Limitierung der Warteschlangenlängen wäre denkbar. Die Möglichkeiten sind hier unzählig.
Außerdem orientiert sich diese Arbeit an dem Gliederungsschema von Disterer.[38] Abweichend von dieser Vorgabe wurde die Gliederung nur möglichst gleichmäßig gestaltet. Eine komplette Gleichmäßigkeit wurde jedoch absichtlich nicht implementiert, denn das hätte eine Anpassung des Programmcodes erfordert, sodass dieser in eine gleichmäßige Gliederung passt. Dies hätte das Programm in der Funktionalität beschnitten. Deshalb wurde im Kapitel 2 von der Richtlinie abgewichen um die Funktionalität des Programms nicht zu beschneiden.

4.2 Ausblick

Die Klassen dieses Programms können frei genutzt werden und stehen auch einzeln außerhalb des Gesamtprogramms zur Verfügung. Es ergeben sich unzählige Möglichkeiten das System zu erweitern. Beispielsweise könnte eine Tendenz der Kunden, an eine Selbstbedienungskasse zu gehen, eingefügt werden. Auch Kunden die von einer Warteschlange zur nächsten springen, weil diese kürzer ist, könnten implementiert werden. Es könnten auch noch mehr unterschiedliche Verteilungen implementiert werden. Eine weitere Möglichkeit solch ein Programm zu nutzen, wäre die Simulation einer Umgebung in die ein Agent gesetzt wird, welcher mithilfe der Machine-Learning-Methode Reinforcement Learning (bestärkendes Lernen) eine optimale Strategie für den Kunden findet, anhand der gegebenen Parameter die Wartezeit zu minimieren, indem er mit der Umgebung agiert und für gute Aktionen eine Belohnung bekommt und für schlechte eine Bestrafung erhält.

4.3 Fazit

Im Zuge dieser Seminararbeit wurden die Grundlagen einer ereignisdiskreten Simulation erklärt und ein Modul und Programm erschaffen, welches die grundlegenden Prinzipien dieser Simulationen implementiert. Mithilfe von echten Daten konnten zugrundeliegende Verteilungen für ein Modell extrahiert werden und dann anhand eines Beispiels mit diesen Verteilungen eine Situation in der echten Welt für eine Entscheidungsfindung simuliert werden.

[38] Disterer 2019, S. 120.

Literatur

3. Data model — Python 3.10.5 documentation (2022). URL: https://docs.python.org/3/reference/datamodel.html#slots (besucht am 17. 07. 2022).

Antczak, T. und R. Weron (Juni 2019). "Point of Sale (POS) Data from a Supermarket: Transactions and Cashier Operations". en. In: *Data* 4.2. Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, S. 67. URL: https://www.mdpi.com/2306-5729/4/2/67 (besucht am 23. 04. 2022).

Banks, J. (o. D.). "Discrete Event Simulation". en. In: *Discrete Event Simulation* (), S. 9.

dataclasses — Data Classes — Python 3.10.5 documentation (2022). URL: https://docs.python.org/3/library/dataclasses.html (besucht am 11. 07. 2022).

Disterer, G. (2019). *Studien- und Abschlussarbeiten schreiben: Seminar-, Bachelor- und Masterarbeiten in den Wirtschaftswissenschaften*. de. Berlin, Heidelberg: Springer Berlin Heidelberg. URL: http://link.springer.com/10.1007/978-3-662-59042-3 (besucht am 19. 07. 2022).

FITTER documentation — fitter 1.4.0 documentation (2022). URL: https://fitter.readthedocs.io/en/latest/ (besucht am 17. 07. 2022).

Peters, T. (Juli 2002). *[Python-Dev] Sorting*. URL: https://mail.python.org/pipermail/python-dev/2002-July/026837.html (besucht am 27. 06. 2022).

Stefik, M., D. Bobrow und K. Kahn (Feb. 1986). "Integrating Access-Oriented Programming into a Multiparadigm Environment". In: *Software, IEEE* 3, S. 10–18.

A Programmcode

Listing 1 Event code

```
@dataclass(slots=True, frozen=True)
class Event:
"""class for keeping track of events"""

# initialize variables for object
# init is set to false, so the counter doesn't reset every time
ev_id: int = field(default_factory=count(start=1).__next__,
    init=False)
""" generate a new id when a class object is created, used for
    sorting purposes in heapq """
time: float
""" time at which event occurs """
kind: str
""" kind of event ("arr": arrival and "dep": departure) """
customer: object
""" customer that is handled """
c_id: int
""" id of checkout where event occurs """
```

Listing 2 Customer code

```
@dataclass(slots=True)
class Customer:
"""class to keep track of customers"""

# initialize Variables for Object
t_arr: float = None
""" arrival time of the customer """
cust_id: int = field(default_factory=count(start=1).__next__,
    init=False)
""" generate an unique id for every customer """
t_dep: float = None
""" initialize departure time """
t_proc: float = None
""" initialize processing time """
num_items: int = None
""" number of items the customer wants to buy"""
proc_rate_per_item: int = None
```

```
""" processing rate per item """
c_id: int = None
"""id of the checkout the customer queues at"""
```

Listing 3 Checkout code

```
@dataclass(slots=True)
class Checkout:
"""Class to keep track of Checkouts"""

# initialize variables
c_id: int
""" id of the checkout """
c_type: str
""" type of the checkout {'sc': self-checkout, 'cc': cashier
    checkout}"""
c_status: int = 0
""" status of the cashier/self-checkout {0: free, 1: Busy}"""
c_quant: int = 1
""" number of cashiers """
sc_quant: int = 6
""" number of self checkouts """
queue: List[Tuple[float, int, Customer]] =
    field(default_factory=list)
""" list of costumers in queue """
processing: List[Tuple[float, int, Customer]] =
    field(default_factory=list)
""" list of customers in processing """

# heapify the queue and processing list and make sure c_status is
    always 0,
def __post_init__(self):
heapq.heapify(self.queue)
heapq.heapify(self.processing)
```

Listing 4 Code für Experimente

```
def main():
"""
Experiment 1:
"""
my_sim = Simulation(
```

```
num_cc=16,
num_sc=1, # six self checkouts with one queue
t_max=10000
)
event_log, customer_log, queue_log = my_sim.simulate()

event_log.to_csv("event_log1.csv", index=False)
customer_log.to_csv("customer_log1.csv", index=False)
queue_log.to_csv("queue_log1.csv", index=False)

"""
Experiment 2:
"""
my_sim = Simulation(
num_cc=22,
num_sc=1, # six self checkouts with one queue
t_max=10000
)
event_log, customer_log, queue_log = my_sim.simulate()

event_log.to_csv("event_log2.csv", index=False)
customer_log.to_csv("customer_log2.csv", index=False)
queue_log.to_csv("queue_log2.csv", index=False)

"""
Experiment 3:
"""
my_sim = Simulation(
num_cc=16,
num_sc=2, # six self checkouts with one queue
t_max=10000
)
event_log, customer_log, queue_log = my_sim.simulate()

event_log.to_csv("event_log3.csv", index=False)
customer_log.to_csv("customer_log3.csv", index=False)
queue_log.to_csv("queue_log3.csv", index=False)
```