DIPLOMARBEIT

Modellierung von Warteschlangensystemen mit dynamisch erzeugten Markov-Ketten

Ausgeführt am

Institut für Breitbandkommunikation
der Technischen Universität Wien

unter Anleitung von

o. Univ.-Prof. Dr.-Ing. Harmen R. van As

durch

Markus Sommereder Bakk.techn.

Wien, 25. April 2007

Kurzfassung

Warteschlangensysteme treten in vielen Bereichen unseres Lebens auf. Eine wichtige Möglichkeit zur Untersuchung von Warteschlangensystemen ist die Modellierung. Unter Modellierung versteht man das Nachbilden eines realen Systems durch ein Modell, das ausschließlich die interessierenden Eigenschaften des Systems beinhaltet, und das anschließende Untersuchen des Verhaltens des Modells. Aus dem Verhalten des Modells können dann Rückschlüsse auf das Verhalten des realen Systems gezogen werden. Ziel dieser Arbeit ist es zu zeigen, wie Warteschlangensysteme unter Verwendung von dynamisch erzeugten Markov-Ketten als Modell untersucht werden können. Insbesondere werden Warteschlangensysteme, die in Telekommunikationseinrichtungen und -netzen auftreten, betrachtet.

Markov-Ketten sind spezielle stochastische Prozesse, die die sehr angenehme Eigenschaft der Gedächtnislosigkeit haben. Es gibt zeitdiskrete und zeitkontinuierliche Markov-Ketten; in dieser Arbeit werden ausschließlich zeitkontinuierliche Markov-Ketten betrachtet. Bei der Untersuchung von zeitkontinuierlichen Markov-Ketten spielen die Chapman-Kolmogorov-Gleichungen eine große Rolle. Sie besagen, dass die Zustandsübergangswahrscheinlichkeiten von zeitkontinuierlichen Markov-Ketten durch ein System von gewöhnlichen Differentialgleichungen 1. Ordnung beschrieben werden können.

Es können Markov-Ketten für verschiedene Fragestellungen über ein Warteschlangensystem (Systemzustand, Durchflusszeit, ...) erstellt werden. Anhand der Eigenschaften dieser Markov-Ketten sind Aussagen über das zugrundeliegende Warteschlangensystem möglich. Da das händische Erzeugen der Markov-Ketten bei komplexen Warteschlangensystemen sehr aufwendig ist, ist es oft hilfreich, stattdessen die Struktur der Markov-Ketten zu beschreiben und die Markov-Ketten so weit wie möglich vom Computer erzeugen zu lassen. Mit Markov-Ketten können nur solche Prozesse nachgebildet werden, bei denen die Zwischenereigniszeiten exponentialverteilt sind. Die Wahrscheinlichkeitsverteilungen von nicht-exponentialverteilten Zwischenereigniszeiten können jedoch oft durch eine Kombination von mehreren Exponentialverteilungen dargestellt oder angenähert werden.

Zum Berechnen der stationären Zustandswahrscheinlichkeiten von zeitkontinuierlichen Markov-Ketten müssen lineare Gleichungssysteme gelöst werden. Da diese Gleichungssysteme oft sehr groß sind, eignen sich für ihre Lösung iterative Verfahren im allgemeinen besser als direkte Verfahren. Um transiente Zustandswahrscheinlichkeiten und Durchflusszeiten zu berechnen, muss man gewöhnliche Differentialgleichungssysteme 1. Ordnung lösen. Hierzu sind beispielsweise Runge-Kutta-Verfahren mit adaptiver Schrittweitensteuerung gut geeignet.

Abstract

Queueing systems appear in many areas of life. An important way to investigate queueing systems is modelling. By modelling one understands the reconstruction of a real system in a model which contains only the interesting properties of the system and the subsequent examination of the behaviour of the model. From the behaviour of the model conclusions on the behaviour of the real system can be drawn. The aim of this work is to show how queueing systems can be investigated using dynamically generated Markov chains as a model. Especially, queueing systems which appear in telecommunication systems and networks are considered.

Markov chains are a special kind of stochastic processes. There are discrete time und continuous time Markov chains; in this work exclusively continuous time Markov chains are considered. For the investigation of continuous time Markov chains the Chapman Kolmogorov equations play an important role. They say that the state transition probabilities of continuous time Markov chains can be described by a system of ordinary first-order differential equations.

Markov chains can be constructed for different questions about a queueing system (system state, flow time, ...). Based on the properties of these Markov chains statements about the underlying queueing system can be made. The manual generation of the Markov chains of complex queueing systems is often very time consuming, complicated and expensive. Therefore, it is helpful to describe the structure of the Markov chains and to let the Markov chains be generated by the computer. With Markov chains only processes with exponential distributed interevent times can be modelled. But often it is possible to approximate the probability distributions of not exponential-distributed interevent times with a combination of several exponential distributions.

In order to calculate the stationary state probabilities of continuous time Markov chains linear equation systems must be solved. Because these equation systems often are very big, iterative methods are suited for their solution in the general better than direct methods. To calculate the transient state probabilities and the flow times, one must solve ordinary first-order differential equation systems. For this purpose, for example, Runge-Kutta methods with adaptive step size control are particularly suitable.

Inhaltsverzeichnis

1. Einleitung

Warteschlangensysteme

Warteschlangensysteme treten in vielen Bereichen unseres Lebens auf. Einige Beispiele dafür sind:

- Anrufer, die in der Warteschleife eines Call-Centers warten, bis jemand ihren Anruf entgegen nimmt
- Datenpakete, die im Puffer eines Internet-Routers warten, bis sie weitergeleitet werden
- Prozesse, die in der Prozesswarteschlange darauf warten, dass ihnen die CPU zugeteilt wird

Warteschlangensysteme haben daher eine große Bedeutung.

Warteschlangensysteme bestehen grundsätzlich aus einem Wartebereich mit einer oder mehreren Warteschlangen, einem Bedienbereich mit einer oder mehreren Bedieneinheiten und aus Anforderungen, die das System durchlaufen. Am System ankommende Anforderungen werden zunächst in den Wartebereich eingereiht. Die Bedieneinheiten entnehmen Anforderungen aus dem Wartebereich und bedienen sie. Nachdem eine Anforderung bedient wurde, verläßt sie das System. Die Zeit, die zwischen zwei Ankünften von Anforderungen vergeht, wird Zwischenankunftszeit genannt. Sie ist im allgemeinen nicht konstant, sondern eine stochastische Größe. Das Gleiche gilt für die Bedienzeit, also die Zeit, die die Bedienung einer Anforderung benötigt. Aus diesem Grund ist der Zustand des Warteschlangensystems (Anzahl der wartenden Anforderungen, ...) nicht genau vorhersagbar, sondern ebenfalls eine stochastische Größe.

Bei der Gestaltung von Warteschlangensystemen müssen einander widersprechende Anforderungen berücksichtigt werden: Einerseits soll die Wartezeit für die Anforderungen möglichst klein sein, andererseits sollen die Bedieneinheiten gut ausgelastet sein. Daher werden Verfahren benötigt, um verschiedene Fragen zum Verhalten von Warteschlangensystemen beantworten zu können. Ein sehr wichtiges dieser Verfahren ist die Modellierung.

Modellierung

Die Modellierung eines Systems (Abbildung 1.1) verläuft in mehreren Schritten: Zunächst wird ein Modell erstellt, das das reale System nachbildet, aber nur mehr dessen interessierende Eigenschaften enthält. Anschließend wird das Modell untersucht. Das kann mit analytischen Methoden oder durch Simulation erfolgen. Aus dem Verhalten des Modells (Modelllösung) können dann Rückschlüsse auf das Verhalten des realen

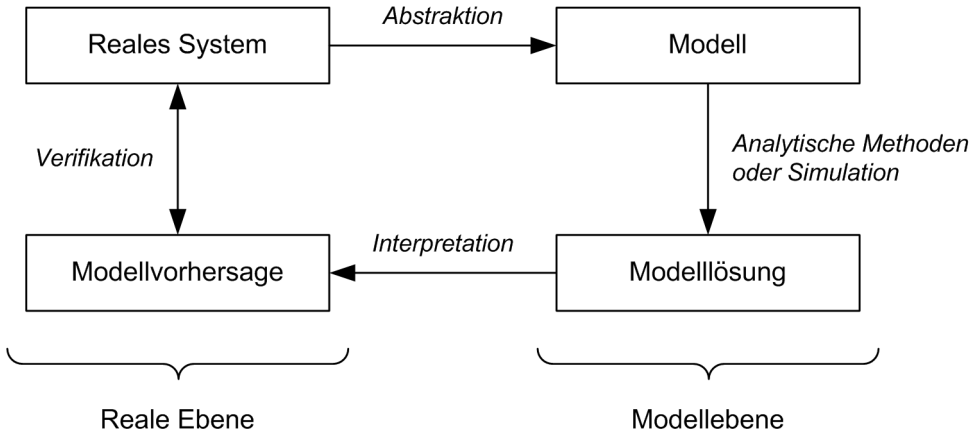


Abbildung 1.1.: Ablauf der Modellierung.

Systems (Modellvorhersage) gezogen werden. Ein Vergleich zwischen Modellvorhersage und tatsächlichem Verhalten des realen Systems erlaubt eine Beurteilung der Qualität des Modells.

Ziel dieser Diplomarbeit ist es zu zeigen, wie Warteschlangensysteme unter Verwendung von sogenannten zeitkontinuierlichen Markov-Ketten als Modell untersucht werden können.

Aufbau der Arbeit

Die Arbeit ist folgendermaßen aufgebaut:

In Kapitel 2 werden die theoretischen Grundlagen von Markov-Ketten erklärt. Obwohl es in dieser Arbeit um zeitkontinuierliche Markov-Ketten geht, werden auch zeitdiskrete Markov-Ketten besprochen.

In Kapitel 3 wird gezeigt, wie die Modellbildung für verschiedene Fragestellungen (Systemzustand, Durchflusszeit, ...) vor sich geht und wie anhand der Eigenschaften der Markov-Kette Rückschlüsse auf das Verhalten des zugrundeliegenden Warteschlangensystems gezogen werden können. Da das händische Erzeugen der Markov-Ketten bei größeren Warteschlangensystemen sehr aufwendig ist, wird auch beschrieben, wie diese Aufgabe effizient am Computer gelöst werden kann.

In Kapitel 4 werden mathematische Verfahren gezeigt, die zum Berechnen der Eigenschaften der Markov-Ketten verwendet werden können.

Im Anhang befinden sich die Quelltexte zu einigen Programmen, die für die Erstellung dieser Arbeit verwendet wurden.

2. Theorie der Markov-Ketten

2.1. Stochastische Prozesse

2.1.1. Stochastische Prozesse

Unter einem stochastischen Prozess versteht man eine Familie von stochastischen Größen $X_t,\ t \in \mathcal{T}$.

Die geordnete, unendliche Menge $\mathcal{T}$ heißt Indexmenge, wobei im folgenden stets angenommen wird, dass es sich dabei um eine Menge von Zeitpunkten handelt. Ist die Indexmenge abzählbar (zum Beispiel $\mathcal{T} = \mathbb{N}$), spricht man von einem zeitdiskreten stochastischen Prozess, andernfalls (zum Beispiel $\mathcal{T} \in \mathbb{R}_0^+$) von einem zeitkontinuierlichen stochastischen Prozess.

Die Werte, die die stochastischen Größen annehmen können, werden die **Zustände**, der Merkmalraum wird der **Zustandsraum** des stochastischen Prozesses genannt. Der Zustandsraum kann endlich, abzählbar oder überabzählbar sein. Bei endlichen Zustandsräumen wird im folgenden mit N die Größe des Zustandsraums bezeichnet.

Ist ein stochastischer Prozess invariant gegenüber Zeitverschiebungen, das heißt unabhängig von der absoluten Zeit, dann wird der Prozess homogen oder **zeitinvariant** genannt, andernfalls inhomogen oder **zeitvariant**. Im folgenden werden ausschließlich homogene Prozesse betrachtet.

2.1.2. Markov-Prozesse

Ein Markov-Prozess (benannt nach dem russischen Mathematiker Andrej A. MARKOV, 1856-1922) ist ein stochastischer Prozess, bei dem die weitere Entwicklung des Prozesses nur vom aktuellen Zustand abhängt und nicht von der Vorgeschichte, das heißt auf welchem Weg der aktuelle Zustand erreicht wurde oder wann er erreicht wurde. Dies wird auch als **Markov-Eigenschaft** oder **Gedächtnislosigkeit** bezeichnet.

Markov-Prozesse sind deutlich einfacher zu untersuchen als allgemeine stochastische Prozesse. Es ist in vielen Fällen möglich, allgemeine stochastische Prozesse zu Markov-Prozessen umzuformen.

2.1.3. Markov-Ketten

Markov-Prozesse mit einem abzählbaren Zustandsraum werden **Markov-Ketten** genannt. Üblicherweise wird als Zustandsraum die Menge der natürlichen Zahlen $\mathbb{N}$ oder eine Teilmenge davon gewählt. Analog zu den stochastischen Prozessen wird von zeitdiskreten und von zeitkontinuierlichen Markov-Ketten gesprochen.

2.2. Zeitdiskrete Markov-Ketten

2.2.1. Übergangswahrscheinlichkeiten

Zeitdiskrete Markov-Ketten können ihren Zustand zu diskreten Zeitpunkten ändern. Die Wahrscheinlichkeit für eine solche Änderung wird durch die Übergangswahrscheinlichkeiten bestimmt.

Die Übergangswahrscheinlichkeit p_{ij} gibt die Wahrscheinlichkeit an, dass sich die Markov-Kette zum Zeitpunkt $n+1$ im Zustand j befindet, unter der Voraussetzung, dass sie zum Zeitpunkt n im Zustand i ist:

$$p_{ij} = W\{X_{n+1} = j \mid X_n = i\} \tag{2.1}$$

Es ist üblich, die Übergangswahrscheinlichkeiten p_{ij} in Form einer Matrix $\mathcal{P}$ zu schreiben:

$$\mathcal{P} = (p_{ij}) = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots \\ p_{2,1} & p_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \tag{2.2}$$

Da $\sum_{j=0}^{n} p_{ij} = 1$ gilt, sind die Zeilensummen der Matrix $\mathcal{P}$ alle gleich 1. Außerdem gilt $0 \leq p_{ij} \leq 1$. Eine solche Matrix wird **stochastische Matrix** genannt.

2.2.2. Mehrschritt-Übergangswahrscheinlichkeiten

Die Wahrscheinlichkeit, dass sich die Markov-Kette zum Zeitpunkt $n+m$ $(m \geq 2)$ im Zustand j befindet, unter der Voraussetzung, dass sie zum Zeitpunkt n im Zustand t ist, also

$$p_{ij}^{(m)} = W\{X_{n+m} = j \mid X_n = i\} \tag{2.3}$$

kann mit Hilfe folgender Überlegung ermittelt werden: Um in m Schritten vom Zustand i zum Zustand j zu gelangen, muss zuerst in l $(0 < l < m)$ Schritten ein Zwischenzustand k erreicht werden (Wahrscheinlichkeit $p_{ik}^{(l)}$). Anschließend muss von diesem

Zwischenzustand in $m - l$ Schritten der Zustand j erreicht werden (Wahrscheinlichkeit $p_{kj}^{(m-l)}$). Als Zwischenzustände kommen alle Zustände der Markov-Kette in Frage:

$$p_{ij}^{(1)} = p_{ij} \tag{2.4}$$

$$p_{ij}^{(m)} = \sum_{\text{alle } k} p_{ik}^{(l)} \; p_{kj}^{(m-k)}, \quad \text{für } 0 < l < m \tag{2.5}$$

Oder in Matrix-Schreibweise:

$$\mathcal{P}^{(1)} = \mathcal{P} \tag{2.6}$$

$$\mathcal{P}^{(m)} = \mathcal{P}^{(l)} \; \mathcal{P}^{(m-l)}, \quad \text{für } 0 < l < m \tag{2.7}$$

Daraus folgt (mit $l = 1$):

$$\mathcal{P}^{(m)} = \mathcal{P} \; \mathcal{P}^{(m-1)} = \mathcal{P} \; \mathcal{P} \; \mathcal{P}^{(m-2)} = \cdots = \mathcal{P}^m \tag{2.8}$$

2.2.3. Darstellung durch Zustandsdiagramme

Zeitdiskrete Markov-Ketten können graphisch durch Zustandsdiagramme dargestellt werden. Dabei werden die einzelnen Zustände durch Kreise repräsentiert und die möglichen Übergänge zwischen den Zuständen als Pfeile, die mit den entsprechenden Übergangswahrscheinlichkeiten beschriftet sind.

2.2.4. Klassifikation von zeitdiskreten Markov-Ketten

Ein Zustand j ist von einem Zustand i aus **erreichbar** ($i \rightsquigarrow j$), wenn j von i aus in beliebig vielen Schritten erreicht werden kann:

$$i \rightsquigarrow j \Leftrightarrow \exists \, m \geq 0 : p_{ij}^{(m)} > 0 \tag{2.9}$$

Eine Menge von Zuständen wird **geschlossen** oder **abgeschlossen** genannt, wenn kein Zustand außerhalb der Menge von einem Zustand in der Menge erreicht werden kann. Besteht eine geschlossene Menge aus genau einem Zustand, dann wird dieser Zustand **absorbierend** genannt.

Zwei Zustände i und j **kommunizieren** miteinander ($i \leftrightsquigarrow j$), wenn sie jeweils vom anderen Zustand aus erreichbar sind:

$$i \leftrightsquigarrow j \Leftrightarrow i \rightsquigarrow j \wedge j \rightsquigarrow i \tag{2.10}$$

Faßt man alle miteinander kommunizierenden Zustände zusammen, entstehen **Kommunikationsklassen**.

Eine Markov-Kette ist **irreduzibel**, wenn jeder Zustand von jedem anderen Zustand aus erreicht werden kann, das heißt, die einzige geschlossene Menge der Markov-Kette ist die Menge aller Zustände der Markov-Kette.

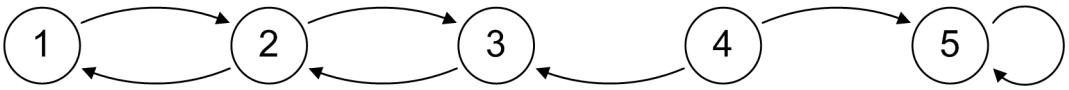


Abbildung 2.1.: Eine zeitdiskrete Markov-Kette mit drei Kommunikationsklassen.

Die Zustände einer nicht geschlossenen Kommunikationsklasse werden **transient** genannt. Transiente Zustände können nur endlich oft eingeommen werden, denn wenn die Kommunikationsklasse des betreffenden Zustands einmal verlassen wurde, kann sie – und somit auch der Zustand – nie wieder erreicht werden.

In der in Abbildung 2.1 gezeigten Markov-Kette gibt es drei Kommunikationsklassen: $\{1, 2, 3\}$, $\{4\}$ und $\{5\}$. Die Mengen $\{1, 2, 3\}$ und $\{5\}$ sind abgeschlossen, der Zustand 5 ist absorbierend. Der Zustand 4 ist transient.

Zustände, die nicht transient sind, sind **rekurrent**. Wenn ein rekurrenter Zustand einmal eingenommen wurde, wird er unendlich oft eingenommen.

Mit $f_i^{(n)}$ wird die Wahrscheinlichkeit bezeichnet, dass die Markov-Kette nach dem Verlassen eines Zustands i nach genau n Schritten zu ihm zurückkehrt. Die Wahrscheinlichkeit, dass die Markov-Kette jemals (das heißt in beliebig vielen Schritten) zu einem Zustand i zurückkehrt, ist

$$f_i = \sum_{m=1}^{\infty} f_i^{(m)} \tag{2.11}$$

Das heißt, für einen rekurrenten Zustand i muss $f_i = 1$ sein.

Der Erwartungswert der Anzahl der Zeitschritte zwischen zwei aufeinanderfolgenden Aufenthalten in einem rekurrenten Zustand j (auch **mittlere Rekurrenzzeit** genannt), ist

$$M_i = \sum_{n=1}^{\infty} n\, f_i^{(n)} \tag{2.12}$$

Ist die mittlere Rekurrenzzeit endlich ($M_i < \infty$), wird der Zustand i **positiv-rekurrent** genannt, andernfalls **null-rekurrent**. Null-rekurrente Zustände treten nur in Markov-Ketten mit unendlichem Zustandsraum auf.

Die Zustände einer irreduziblen Markov-Kette sind alle entweder positiv-rekurrent, null-rekurrent oder transient.

Die Zustände einer endlichen, irreduziblen Markov-Kette sind alle positiv-rekurrent.

Ein Zustand j wird **periodisch** mit der Periode p genannt, wenn nach dem Verlassen des Zustands eine Rückkehr nur mit einer Anzahl von Schritten, die ein Vielfaches von p ist, möglich ist. Das heißt, p ist der größte gemeinsame Teiler aller Zahlen r, für die $p_{jj}^{(r)} > 0$ ist. Ein Zustand mit der Periode $p = 1$ wird **aperiodisch** genannt. Ist in einer irreduziblen Markov-Kette ein Zustand aperiodisch, dann sind alle anderen Zustände auch aperiodisch.

Eine zeitdiskrete Markov-Kette ist **ergodisch**, wenn sie irreduzibel, aperiodisch und positiv-rekurrent ist.

2.2.5. Zustandswahrscheinlichkeiten

Die Wahrscheinlichkeit, dass sich die Markov-Kette zum Zeitpunkt n im Zustand i befindet (**Zustandswahrscheinlichkeit**), wird mit $\pi_i(n)$ bezeichnet:

$$\pi_i(n) = W\{X_n = i\} \tag{2.13}$$

Die Wahrscheinlichkeitsverteilung der Markov-Kette (also die Gesamtheit aller Zustandswahrscheinlichkeiten) kann als Vektor geschrieben werden:

$$\pi(n) = (\pi_0(n),\ \pi_1(n),\ \dots) \tag{2.14}$$

Die Zustandswahrscheinlichkeiten können berechnet werden, wenn die Anfangswahrscheinlichkeitsverteilung $\pi(0)$ bekannt ist:

$$\pi_i(n) = \sum_{\text{alle } k} p_{ki}^{(n)}\ \pi_k(0) \tag{2.15}$$

Oder in Matrix-Schreibweise:

$$\pi(n) = \pi(0)\ \mathcal{P}^{(n)} = \pi(0)\ \mathcal{P}^n \tag{2.16}$$

Von einer **stationären Wahrscheinlichkeitsverteilung** $\pi = (\pi_1,\ \pi_2,\ \dots)$ spricht man, wenn

$$\pi\ \mathcal{P} = \pi \tag{2.17}$$

gilt. Das heißt, wenn die Markov-Kette eine stationäre Wahrscheinlichkeitsverteilung eingenommen hat, wird diese für immer beibehalten.

Ist eine bestimmte Anfangswahrscheinlichkeitsverteilung $\pi(0)$ gegeben und existiert der Grenzwert $\tilde{\pi} = \lim\limits_{n\to\infty} \pi(n)$, so wird $\tilde{\pi}$ **Grenzwahrscheinlichkeitsverteilung** genannt.

Für die Existenz und Eindeutigkeit der beiden Wahrscheinlichkeitsverteilungen gilt:

- In einer aperiodischen Markov-Kette existiert zu jeder Anfangswahrscheinlichkeitsverteilung die Grenzwahrscheinlichkeitsverteilung. In einer periodischen Markov-Kette ist das nicht der Fall, was zum Beispiel durch eine Markov-Kette mit $\mathcal{P} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ und $\pi(0) = (1, 0)$ leicht ersichtlich ist.
- In einer aperiodischen und irreduziblen Markov-Kette existiert eine eindeutige Grenzwahrscheinlichkeitsverteilung (die unabhängig von der Anfangswahrscheinlichkeitsverteilung ist). In einer reduziblen Markov-Kette ist das nicht der Fall, zum Beispiel ist bei einer Markov-Kette mit $\mathcal{P} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ jede Anfangswahrscheinlichkeitsverteilung ihre Grenzwahrscheinlichkeitsverteilung.
- In einer irreduziblen, positiv-rekurrenten Markov-Kette existiert eine eindeutige stationäre Wahrscheinlichkeitsverteilung. In einer reduziblen Markov-Kette ist das nicht der Fall, zum Beispiel ist bei einer Markov-Kette mit $\mathcal{P} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ jede Wahrscheinlichkeitsverteilung stationär.

- In einer aperiodischen, irreduziblen und positiv-rekurrenten (das heißt ergodischen) Markov-Kette existiert eine eindeutige Grenzwahrscheinlichkeitsverteilung, die gleich der eindeutigen stationären Wahrscheinlichkeitsverteilung ist. Diese Wahrscheinlichkeitsverteilung kann durch Lösen der Gleichung (2.17) berechnet werden, wobei als Nebenbedingung $\sum_i \pi_i = 1$ gelten muss. Alternativ kann π aus der Beziehung

$$\lim_{n\to\infty} \mathcal{P}^n = (\pi,\ \pi,\ \dots)^T \tag{2.18}$$

ermittelt werden.

2.2.6. Verweilzeiten

Die Zeit, die eine Markov-Kette in einem bestimmten Zustand i verbringt, wird **Verweilzeit** genannt und mit R_i bezeichnet. Aufgrund der Gedächtnislosigkeit von zeitdiskreten Markov-Ketten sind die Verweilzeiten geometrisch verteilt:

$$W\{R_i = k\} = p_{ii}^{k-1} \sum_{j\neq i} p_{ij} \tag{2.19}$$

Der Erwartungswert der Verweilzeit ist daher

$$\mathrm{E}[R_i] = \frac{1}{1-p_{ii}} \tag{2.20}$$

und die Varianz ist

$$\mathrm{Var}[R_i] = \frac{p_{ii}}{(1-p_{ii})^2} \tag{2.21}$$

2.3. Zeitkontinuierliche Markov-Ketten

2.3.1. Übergangsraten

Zeitkontinuierliche Markov-Ketten können ihren Zustand zu jedem beliebigen Zeitpunkt ändern. Daher sind hier die Übergangswahrscheinlichkeiten vom betrachteten Intervall abhängig:

Die Übergangswahrscheinlichkeit $p_{ij}(\tau)$ $(\tau \geq 0)$ gibt die Wahrscheinlichkeit an, dass sich die Markov-Kette zum Zeitpunkt $t+\tau$ im Zustand j befindet, unter der Voraussetzung, dass sie zum Zeitpunkt t im Zustand i ist:

$$p_{ij}(\tau) = W\{X_{t+\tau} = j \mid X_t = i\} \tag{2.22}$$

Je kleiner das Intervall τ ist, desto kleiner ist die Wahrscheinlichkeit für einen Übergang in einen anderen Zustand. Für $\tau = 0$ gilt schließlich

$$p_{ij}(0) = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{2.23}$$

Mit steigender Länge des Intervalls wird die Wahrscheinlichkeit für einen Übergang immer größer, wobei dann auch Mehrfachübergänge (das heißt, die Markov-Kette wechselt vom Zustand i zu einem Zwischenzustand k und von diesem zum Zustand j) auftreten können und berücksichtigt werden müssen. Um nur einfache Übergänge zu beschreiben, muss τ daher möglichst klein gewählt werden.

Die **Übergangsrate** q_{ij} ist nun definiert als Ableitung der Übergangswahrscheinlichkeit $p_{ij}(\tau)$ zum Zeitpunkt $\tau = 0$:

$$q_{ij}(\tau) = p'_{ij}(0) = \lim_{\tau \to 0} \frac{p_{ij}(\tau) - p_{ij}(0)}{\tau - 0} \tag{2.24}$$

Für $j \neq i$ ergibt sich

$$q_{ij}(\tau) = \lim_{\tau \to 0} \frac{p_{ij}(\tau)}{\tau} \tag{2.25}$$

Für $j = i$ ergibt sich

$$q_{ii}(\tau) = \lim_{\tau \to 0} \frac{p_{ii}(\tau) - 1}{\tau} \tag{2.26}$$

Mit $p_{ii}(\tau) = 1 - \sum_j p_{ij}(\tau)$ folgt

$$q_{ii}(\tau) = \lim_{\tau \to 0} \frac{1 - \sum_j p_{ij}(\tau) - 1}{\tau} = -\sum_j \lim_{\tau \to 0} \frac{p_{ij}(\tau)}{\tau} \tag{2.27}$$

$$q_{ii}(\tau) = -\sum_j q_{ij}(\tau) \tag{2.28}$$

Die Übergangsraten können auch als Matrix geschrieben werden:

$$\mathcal{Q} = (q_{ij}) = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots \\ q_{2,1} & q_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \tag{2.29}$$

2.3.2. Darstellung durch Zustandsdiagramme

Zeitkontinuierliche Markov-Ketten können graphisch durch Zustandsdiagramme dargestellt werden. Dabei werden die einzelnen Zustände durch Kreise repräsentiert und die möglichen Übergänge zwischen den Zuständen als Pfeile, die mit den entsprechenden Übergangsraten beschriftet sind.

2.3.3. Die Chapman-Kolmogorov-Gleichungen

Die Übergangswahrscheinlichkeit zwischen zwei Zuständen (unter Berücksichtigung von Mehrfachübergängen) kann mit Hilfe folgender Überlegung ermittelt werden:

Die Markov-Kette sei zum Zeitpunkt t im Zustand i. Wenn sie zum Zeitpunkt $t+\tau+\alpha$ im Zustand j ist, dann muss sie zum Zeitpunkt $t+\tau$ in einem Zwischenzustand k gewesen sein. Die Wahrscheinlichkeit dafür ist $p_{ik}(\tau)$. Anschließend muss in α Zeiteinheiten ein Übergang von diesem Zwischenzustand k nach j stattgefunden haben. Die Wahrscheinlichkeit dafür ist $p_{kj}(\alpha)$. Da als Zwischenzustände alle Zustände der Markov-Kette in Frage kommen (auch i und j selbst), gilt

$$p_{ij}(\tau+\alpha) = \sum_k p_{ik}(\tau)\; p_{kj}(\alpha) \tag{2.30}$$

Wegen $\tau+\alpha = \alpha+\tau$ gilt weiters

$$p_{ij}(\tau+\alpha) = \sum_k p_{ik}(\alpha)\; p_{kj}(\tau) \tag{2.31}$$

Daraus folgt

$$\begin{aligned} p_{ij}(\tau+\alpha) - p_{ij}(\tau) &= \sum_{k\neq j} p_{ik}(\tau)\; p_{kj}(\alpha) + p_{ij}(\tau)\; p_{jj}(\alpha) - p_{ij}(\tau) \\ &= \sum_{k\neq j} p_{ik}(\tau)\; p_{kj}(\alpha) + p_{ij}(\tau)\;\; (p_{jj}(\alpha) - 1) \end{aligned}$$

und

$$\begin{aligned} p_{ij}(\tau+\alpha) - p_{ij}(\tau) &= \sum_{k\neq i} p_{ik}(\alpha)\; p_{kj}(\tau) + p_{ii}(\alpha)\; p_{ij}(\tau) - p_{ij}(\tau) \\ &= \sum_{k\neq i} p_{ik}(\alpha)\; p_{kj}(\tau) + p_{ij}(\tau)\;\; (p_{ii}(\alpha) - 1) \end{aligned}$$

Mit Gleichung (2.23) folgt

$$p_{ij}(\tau+\alpha) - p_{ij}(\tau) = \sum_{k\neq j} p_{ik}(\tau)\;\; (p_{kj}(\alpha) - p_{kj}(0)) + p_{ij}(\tau)\;\; (p_{jj}(\alpha) - p_{jj}(0)) \tag{2.32}$$

und

$$p_{ij}(\tau+\alpha) - p_{ij}(\tau) = \sum_{k\neq i} p_{kj}(\tau)\;\; (p_{ik}(\alpha) - p_{ik}(0)) + p_{ij}(\tau)\;\; (p_{ii}(\alpha) - p_{ii}(0)) \tag{2.33}$$

Dividieren beider Seiten durch α und bilden des Grenzwertes $\alpha \to 0$ ergibt

$$\begin{aligned} \lim_{\alpha\to 0} \underbrace{\frac{p_{ij}(\tau+\alpha) - p_{ij}(\tau)}{\alpha}}_{\to p'_{ij}(\tau)} = \\ \lim_{\alpha\to 0} \sum_{k\neq j} p_{ik}(\tau) \underbrace{\frac{p_{kj}(\alpha) - p_{kj}(0)}{\alpha}}_{\to p'_{kj}(0) = q_{kj}} + \lim_{\alpha\to 0} p_{ij}(\tau) \underbrace{\frac{p_{jj}(\alpha) - p_{jj}(0)}{\alpha}}_{\to p'_{jj}(0) = q_{jj}} \end{aligned} \tag{2.34}$$

und

$$\lim_{\alpha\to 0} \underbrace{\frac{p_{ij}(\tau+\alpha)-p_{ij}(\tau)}{\alpha}}_{\to p'_{ij}(\tau)} = \lim_{\alpha\to 0}\sum_{k\neq i} p_{kj}(\tau)\underbrace{\frac{p_{ik}(\alpha)-p_{ik}(0)}{\alpha}}_{\to p'_{ik}(0)=q_{ik}} + \lim_{\alpha\to 0} p_{ij}(\tau)\underbrace{\frac{p_{ii}(\alpha)-p_{ii}(0)}{\alpha}}_{\to p'_{ii}(0)=q_{ii}} \tag{2.35}$$

Die erste Gleichung wird Chapman-Kolmogorov-Vorwärts-Gleichung genannt:

$$\boxed{p'_{ij}(\tau) = \sum_k p_{ik}(\tau)\; q_{kj}} \tag{2.36}$$

Die zweite Gleichung ist die Chapman-Kolmogorov-Rückwärts-Gleichung:

$$\boxed{p'_{ij}(\tau) = \sum_k q_{ik}\; p_{kj}(\tau)} \tag{2.37}$$

2.3.4. Klassifikation von Zuständen

Ein Zustand j ist von einem Zustand i aus **erreichbar** ($i \rightsquigarrow j$), wenn j von i aus in beliebiger Zeit erreicht werden kann:

$$i \rightsquigarrow j \Leftrightarrow \exists\, t : p_{ij}(t) > 0 \tag{2.38}$$

Eine Menge von Zuständen wird **geschlossen** oder **abgeschlossen** genannt, wenn kein Zustand außerhalb der Menge von einem Zustand in der Menge erreicht werden kann. Besteht eine geschlossene Menge aus genau einem Zustand, dann wird dieser Zustand **absorbierend** genannt.

Zwei Zustände i und j **kommunizieren** miteinander ($i \leftrightsquigarrow j$), wenn sie jeweils vom anderen Zustand aus erreichbar sind:

$$i \leftrightsquigarrow j \Leftrightarrow i \rightsquigarrow j \wedge j \rightsquigarrow i \tag{2.39}$$

Faßt man alle miteinander kommunizierenden Zustände zusammen, entstehen **Kommunikationsklassen**.

Eine Markov-Kette ist **irreduzibel**, wenn jeder Zustand von jedem anderen Zustand aus erreicht werden kann, das heißt, die einzige geschlossene Menge der Markov-Kette ist die Menge aller Zustände der Markov-Kette.

Die Zustände einer nicht geschlossenen Kommunikationsklasse werden **transient** genannt. Transiente Zustände können nur endlich oft eingeommen werden, denn wenn die Kommunikationsklasse des betreffenden Zustands einmal verlassen wurde, kann sie – und somit auch der Zustand – nie wieder erreicht werden.

Zustände, die nicht transient sind, sind **rekurrent**. Rekurrente Zustände werden unendlich oft eingenommen.

Zustände einer zeitkontinuierlichen Markov-Kette können nicht periodisch sein.

Eine zeitkontinuierliche Markov-Kette ist **ergodisch**, wenn sie irreduzibel und positiv-rekurrent ist.

2.3.5. Zustandswahrscheinlichkeiten

Die Wahrscheinlichkeit $\pi_j(\tau)$, dass sich die Markov-Kette zum Zeitpunkt τ im Zustand i befindet, kann mit Hilfe der Chapman-Kolmogorov-Vorwärts-Gleichung (2.36) berechnet werden.

Es gilt

$$\pi_j(\tau) = \sum_i \pi_i(0)\ p_{ij}(\tau) \tag{2.40}$$

Ableiten beider Seiten ergibt

$$\pi'_j(\tau) = \sum_i \pi_i(0)\ p'_{ij}(\tau) = \sum_i \pi_i(0) \sum_k p_{ik}(\tau)\ q_{kj} = \sum_k \underbrace{\sum_i \pi_i(0)\ p_{ik}(\tau)}_{\pi_k(\tau)}\ q_{kj} \tag{2.41}$$

$$\boxed{\pi'_j(\tau) = \sum_k \pi_k(\tau)\ q_{kj}} \tag{2.42}$$

Diese Gleichung kann auch in Matrix-Form geschrieben werden:

$$\boxed{\pi'(\tau) = \pi(\tau)\ \mathcal{Q}} \tag{2.43}$$

Von einer **stationären Wahrscheinlichkeitsverteilung** $\pi = (\pi_1,\ \pi_2,\ \dots)$ spricht man, wenn

$$\pi' = \pi\ \mathcal{Q} = 0 \tag{2.44}$$

gilt. Das heißt, wenn die Markov-Kette eine stationäre Wahrscheinlichkeitsverteilung eingenommen hat, ist die Änderung der Wahrscheinlichkeitsverteilung 0 und die Wahrscheinlichkeitsverteilung wird für immer beibehalten.

Ist eine bestimmte Anfangswahrscheinlichkeitsverteilung $\pi(0)$ gegeben und existiert der Grenzwert $\tilde{\pi} = \lim_{t\to\infty} \pi(t)$, so wird $\tilde{\pi}$ **Grenzwahrscheinlichkeitsverteilung** genannt.

Für die Existenz und Eindeutigkeit der beiden Wahrscheinlichkeitsverteilungen gilt:

- In einer irreduziblen Markov-Kette existiert eine eindeutige Grenzwahrscheinlichkeitsverteilung (die unabhängig von der Anfangswahrscheinlichkeitsverteilung ist).
- In einer irreduziblen, endlichen, zeitkontinuierlichen Markov-Kette existiert eine eindeutige stationäre Wahrscheinlichkeitsverteilung. Diese Wahrscheinlichkeitsverteilung kann durch Lösen der Gleichung (2.44) berechnet werden, wobei als Nebenbedingung $\sum_i \pi_i = 1$ gelten muss.

2.3.6. Verweilzeiten

Die Zeit, die eine Markov-Kette in einem bestimmten Zustand i verbringt, wird **Verweilzeit** genannt und mit R_i bezeichnet. Aufgrund der Gedächtnislosigkeit von zeitkontinuierlichen Markov-Ketten sind die Verweilzeiten exponentialverteilt (mit dem Parameter $-q_{ii}$):

$$W\{R_i \leq t\} = 1 - e^{q_{ii}t} \tag{2.45}$$

Der Erwartungswert der Verweilzeit ist daher

$$\mathrm{E}[R_i] = -\frac{1}{q_{ii}} \tag{2.46}$$

und die Varianz ist

$$\mathrm{Var}[R_i] = \frac{1}{{q_{ii}}^2} \tag{2.47}$$

Manchmal ist es erforderlich zu wissen, wie lange sich die Markov-Kette außerhalb einer bestimmten Teilmenge $\mathcal{H}$ des Zustandsraums befindet. Dies kann mit Hilfe der Chapman-Kolmogorov-Rückwärts-Gleichung berechnet werden:

Sei $f_i(\tau)$ die Wahrscheinlichkeit, dass die Markov-Kette nach dem Intervall τ in einem Zustand außerhalb der Teilmenge $\mathcal{H}$ ist, wenn sie zum Zeitpunkt t im Zustand i ist:

$$f_i(t+\tau) = W\{X(t+\tau) \notin \mathcal{H} \mid X(t) = i\} \tag{2.48}$$

Dann gilt

$$f_i(\tau) = \sum_{j \notin \mathcal{H}} p_{ij}(\tau) \tag{2.49}$$

Ableiten beider Seiten ergibt

$$f_i'(\tau) = \sum_{j \notin \mathcal{H}} p_{ij}'(\tau) = \sum_{j \notin \mathcal{H}} \sum_k q_{ik}\, p_{kj}(\tau) = \sum_k q_{ik} \underbrace{\sum_{j \notin \mathcal{H}} p_{kj}(\tau)}_{f_k(\tau)} \tag{2.50}$$

$$\boxed{f_i'(\tau) = \sum_k q_{ik}\, f_k(\tau)} \tag{2.51}$$

Mit $f(\tau) = (f_1(\tau), f_2(\tau), \ldots, f_N(\tau))^T$ kann diese Gleichung ebenfalls in Matrix-Form geschrieben werden:

$$\boxed{f'(\tau) = \mathcal{Q}\, f(\tau)} \tag{2.52}$$

Für die Anfangsbedingungen $f_i'(0)$ gilt

$$f_i(0) = 1 \quad \forall i \notin \mathcal{H} \qquad f_i(0) = 0 \quad \forall i \in \mathcal{H} \tag{2.53}$$

3. Modellierung von Warteschlangensystemen mit Markov-Ketten

In diesem Kapitel wird der Vorgang der Modellierung anhand einiger einfacher Warteschlangensysteme erläutert und es werden wichtige Ergebnisse gezeigt. Da es in dieser Arbeit nicht um die Eigenschaften von Warteschlangensystemen sondern um den Vorgang der Modellierung geht, wird auf die Ergebnisse nicht näher eingegangen. Eine ausführliche Diskussion der Eigenschaften der hier gezeigten Warteschlangensysteme und der Ergebnisse ist in [van As 1984] zu finden.

Die meisten der gezeigten Warteschlangensysteme wurden mit einem im Rahmen dieser Arbeit entstandenen C++-Programm untersucht, dessen Quellcode im Anhang A abgedruckt ist. Manche Systeme wurden mit MATLAB untersucht, die entsprechenden Programme befinden sich im Anhang B.

Noch ein Hinweis zur Notation: Die Zustände der Markov-Ketten werden im Folgenden je nach Bedarf entweder durch ihre Nummer (0, 1, 2, ...) oder durch ihre Beschriftung (Idle, 0/1/0, ...) bezeichnet.

3.1. Allgemeines Vorgehen

Die Modellierung von Warteschlangensystemen mit Markov-Ketten erfolgt in drei Schritten:

Zuerst wird eine zur Fragestellung passende Markov-Kette erstellt. Dabei ist darauf zu achten, dass die Markov-Kette alle benötigten Eigenschaften des Warteschlangensystems enthält. Gleichzeitig darf sie aber nicht zu viele Details enthalten, da sonst der Aufwand zum Berechnen der Eigenschaften der Markov-Kette unnötig groß ist.

Anschließend werden die Eigenschaften der Markov-Kette (zum Beispiel die transienten und stationären Zustandswahrscheinlichkeiten) berechnet. Dazu benötigte mathematische Verfahren werden in Kapitel 4 beschrieben.

Zuletzt können anhand dieser Eigenschaften die gesuchten Werte im realen Warteschlangensystem berechnet werden. Die transienten Zustandswahrscheinlichkeiten korrespondieren dabei mit den transienten Vorgängen im Warteschlangensystem, die sta-

tionären Zustandswahrscheinlichkeiten korrespondieren mit dem eingeschwungenen Zustand[1] des Warteschlangensystems.

Aufgrund der Gedächtnislosigkeit von Markov-Ketten können nur solche Prozesse direkt nachgebildet werden, deren Zwischenereigniszeiten exponentialverteilt sind. Wie allgemeine Prozesse nachgebildet werden können, ist in Abschnitt 3.6 beschrieben.

Außerdem ist es nicht möglich, Anforderungen unterschiedlicher Länge zu unterscheiden. Es kann nur die Anzahl der Anforderungen betrachtet werden.

3.2. Effizientes Erstellen der Markov-Ketten am Computer

Mit steigender Komplexität eines Warteschlangensystems wächst die Größe und damit der Aufwand zum Erstellen der benötigten Markov-Kette sehr rasch an. Hier ist es oft hilfreich, die Struktur der Markov-Kette zu beschreiben und die Markov-Kette so weit wie möglich vom Computer erzeugen zu lassen.

Die Beschreibung der Struktur besteht aus zwei Teilen:

- Beschreibung, wie ein Zustand grundsätzlich aussieht und festlegen von Regeln, wann ein Zustand tatsächlich in der Markov-Kette vorkommt. In der Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems[2] mit mehreren Klassen von Anforderungen würde ein Zustand zum Beispiel für jede Klasse die Anzahl der Anforderungen im System sowie den Zustand der Bedieneinheit beinhalten. Ein Zustand kommt aber (unter anderem) nur dann in der Markov-Kette vor, wenn die Summe der Anzahlen der Anforderungen aller Klassen nicht größer ist als die Gesamtgröße des Warteschlangensystems.
- Erstellen von Regeln, ob und mit welcher Rate ein Übergang zwischen zwei Zuständen stattfinden kann. Beim Erstellen dieser Regeln überlegt man sich, welche Übergänge im System auftreten können und wie man diese Übergänge durch Vergleich der Werte von Anfangs- und Endzustand erkennen kann. Für die Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems würde eine Regel beispielsweise lauten: Wenn in einem Zustand A die Anzahl der Anforderungen um eins höher ist als in einem Zustand B, dann ist die Übergangsrate von A nach B gleich der Bedienrate.

Zur Verdeutlichung werden bei einigen der im Folgenden gezeigten Beispiele die verwendeten Regeln in Text-Form und als Code-Ausschnitte angegeben.

Das Erstellen der Markov-Kette geschieht nun wie folgt:

1. Bereitstellen einer Datenstruktur, die die Beschreibung eines Zustands enthält.

[1] Unter dem eingeschwungenen Zustand eines Systems versteht man jenen Zustand, der nach dem Abklingen aller Ausgleichsvorgänge erreicht wird.

[2] M/M/1/S bezeichnet in der allgemein verwendeten Kendall-Notation ein Warteschlangensystem mit exponentialverteilten Zwischenankunftszeiten, exponentialverteilten Bedienzeiten, einer Bedieneinheit und Systemgröße S. (Mit Systemgröße ist die Anzahl der Warteplätze plus die Anzahl der Bedienplätze gemeint. Manchmal wird mit S auch nur die Anzahl der Warteplätze bezeichnet.)

2. Bereitstellen einer Funktion, die die Gültigkeit eines Zustands prüft.

3. Bereitstellen einer Funktion, die die Übergangsrate zwischen jeweils zwei Zuständen berechnet. Das Berechnen der Übergangsrate erfolgt hauptsächlich durch das Angeben von Regeln. Bei speziellen Zuständen, die durch Regeln nicht effektiv erfaßt werden können (zum Beispiel Leerlauf, volles System, ...), können die Übergangsraten auch explizit angegeben werden.

4. Erstellen einer Liste mit allen Zuständen der Markov-Kette. Dazu wird für jeden darstellbaren Zustand geprüft, ob er gültig ist. Ist das der Fall, wird er in die Liste aufgenommen.

5. Erstellen der Matrix mit den Zustandsübergangsraten. Dazu wird für jedes Paar von verschiedenen Zuständen aus der erstellten Liste die Übergangsrate ermittelt und gespeichert. Die Übergangsrate von einem Zustand zu sich selbst wird zum Schluss berechnet. Sie ist gleich der negativen Summe aller Übergangsraten von ihm zu anderen Zuständen. Die entstehende Matrix ist meist dünn besetzt und kann daher bei Verwendung geeigneter Datenstrukturen sehr platzsparend gespeichert werden.

3.3. Modellierung des Systemzustands

Für die meisten Fragestellungen ist es notwendig, den Systemzustand zu modellieren. Dabei bilden die Zustände der Markov-Kette jeden möglichen Belegungszustand des Warteschlangensystems ab. Die Übergangsraten zwischen den Zuständen sind die entsprechenden Raten im Warteschlangensystem (also zum Beispiel Ankunfts- und Bedienraten).

3.3.1. M/M/1/S-Warteschlangensystem

Zunächst wird ein M/M/1/S-Warteschlangensystem ($S = 5$) mit Ankunftsrate λ und Bedienrate μ untersucht.

Modell

Die Markov-Kette für den Systemzustand dieses Systems ist in Abbildung 3.1 gezeigt.

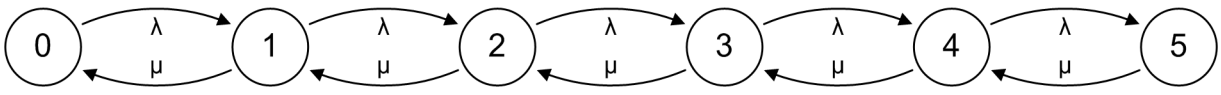


Abbildung 3.1.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems. $S = 5$. Die Nummer der Zustände entspricht der Anzahl der Anforderungen im System.

Ergebnisse

Nachdem die Zustandswahrscheinlichkeiten der Markov-Kette ermittelt wurden, können zum Beispiel folgende Werte berechnet werden:

Der Erwartungswert der Anzahl der Anforderungen im System im eingeschwungenen Zustand ist

$$\mathrm{E}[X] = \sum_{k=0}^{5} k\ \pi_k \tag{3.1}$$

Abbildung 3.2 zeigt die Anzahl der Anforderungen im System in Abhängigkeit von der Ankunftsrate λ.

Die Wahrscheinlichkeit für ein volles System (Blockierwahrscheinlichkeit) im eingeschwungenen Zustand ist

$$p_{\text{blocking}} = \pi_5 \tag{3.2}$$

In Abbildung 3.3 ist die Blockierwahrscheinlichkeit in Abhängigkeit von der Ankunftsrate λ zu sehen.

Der Erwartungswert der Anzahl der Anforderungen im System während eines Einschaltvorgangs (das System befindet sich im Leerlauf und es beginnt ein Ankunftsprozess) ist

$$\mathrm{E}[X](t) = \sum_{k=0}^{5} k\ \pi_k(t) \qquad \pi(0) = (1, 0, 0, 0, 0, 0) \tag{3.3}$$

Die Ergebnisse für verschiedene Werte von λ sind in Abbildung 3.4 zu sehen.

Die Auslastung der Bedieneinheit ist gleich der Wahrscheinlichkeit, dass das System nicht im Leerlauf ist, also

$$\rho = 1 - \pi_0 \tag{3.4}$$

Die für die Untersuchung dieses Warteschlangensystems verwendeten MATLAB-Programme *M_M_1_S_stationaer.m* und *M_M_1_S_transient.m* sind im Anhang B zu finden (Listings B.1 und B.2).

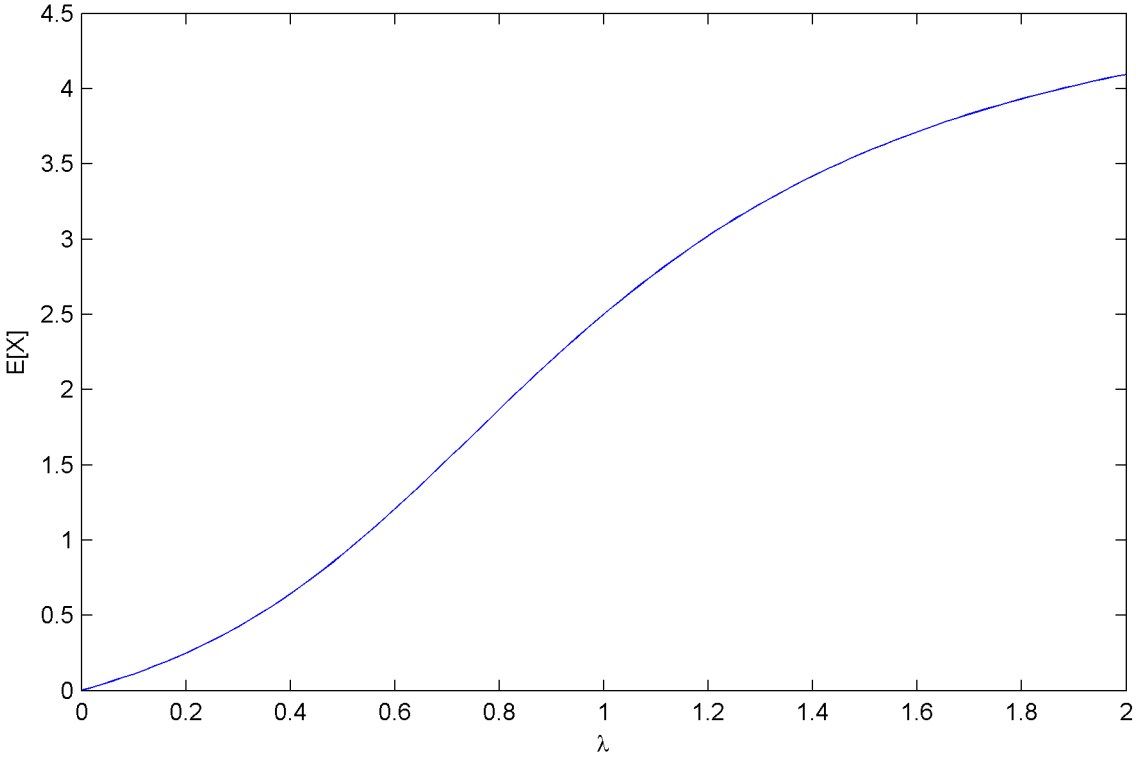


Abbildung 3.2.: M/M/1/S-Warteschlangensystem: Erwartungswert der Anzahl der Anforderungen im System im eingeschwungenen Zustand in Abhängigkeit von der Ankunftsrate.

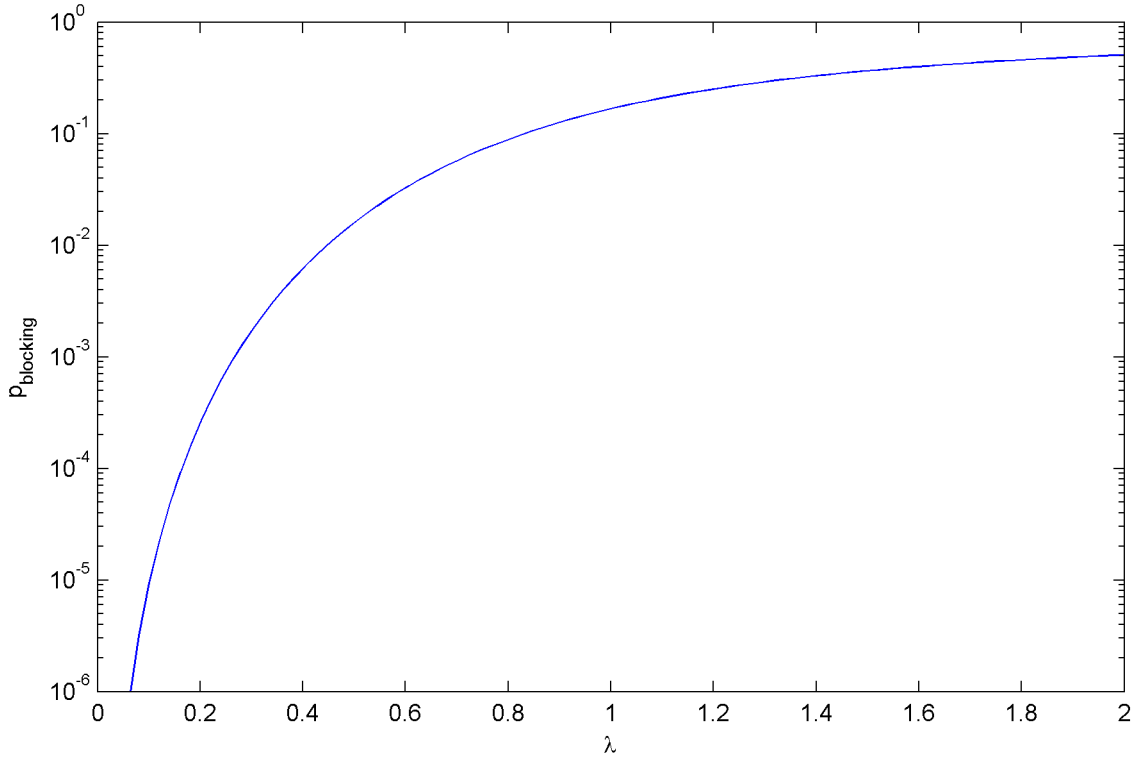


Abbildung 3.3.: M/M/1/S-Warteschlangensystem: Wahrscheinlichkeit für ein volles System im eingeschwungenen Zustand in Abhängigkeit von der Ankunftsrate.

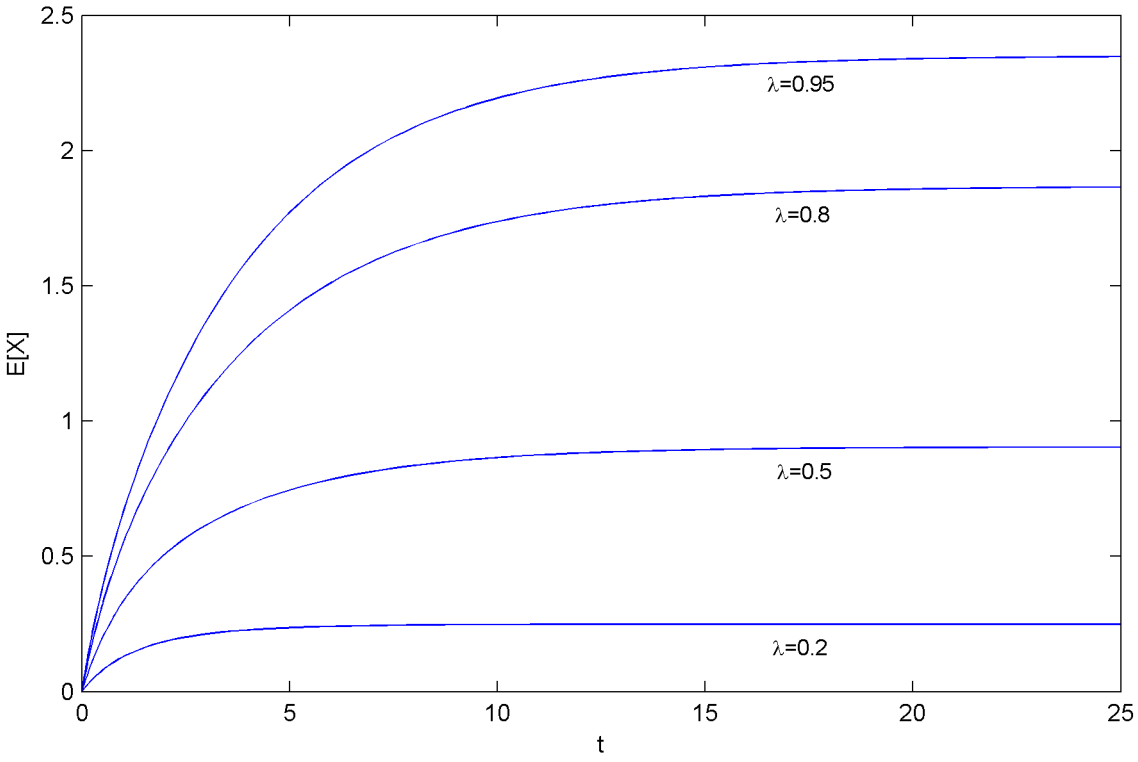


Abbildung 3.4.: M/M/1/S-Warteschlangensystem: Erwartungswert der Anzahl der Anforderungen im System während eines Einschaltvorgangs.

3.3.2. M/M/1/S-Warteschlangensystem mit mehreren Klassen von Anforderungen

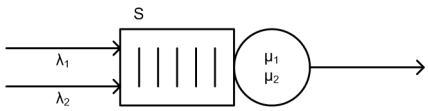


Abbildung 3.5.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen.

Bei diesem System gehört jede ankommende Anforderung zu einer bestimmten Klasse $(1, 2, \ldots, K)$. Die entsprechenden Ankunftsraten sind $\lambda_1, \lambda_2, \ldots, \lambda_K$, die Bedienraten sind $\mu_1, \mu_2, \ldots, \mu_K$. Die verwendete **Abfertigungsdisziplin** bestimmt, wie die einzelnen Klassen bei der Bedienung behandelt werden. Der verfügbare Platz in der Warteschlange kann von allen Klassen uneingeschränkt verwendet werden *(common buffer)*.

Abfertigungsdisziplinen

Bei den hier verwendeten Abfertigungsdisziplinen hängt die Entscheidung, welche Klasse als nächstes bedient wird, ausschließlich von den Anzahlen $n_1, \ldots, n_K$ der Anforderungen pro Klasse ab (und nicht von der Vorgeschichte, also zum Beispiel welche Anforderung zuletzt bearbeitet wurde).

Bei der *Abfertigung in zufälliger Reihenfolge* wird aus der Menge der wartenden Anforderungen zufällig eine ausgewählt. Die Wahrscheinlichkeit, dass eine Anforderung der Klasse k gewählt wird, entspricht dem Anteil von Klasse k-Anforderungen in der Gesamtheit der wartenden Anforderungen:

$$W\{\text{Auswahl Klasse } k \mid n_1 \ldots n_K\} = \frac{n_k}{\sum_{i=1}^{K} n_i} \tag{3.5}$$

Bei der Abfertigungsdisziplin *Nichtunterbrechende Prioritäten* werden den Klassen Prioritäten zugeordnet: Die Klasse 1 hat höchste Priorität, die Klasse 2 nächstniedrigere Priorität und so weiter. Es wird immer eine Anforderung der höchsten vorhandenen Prioritätsklasse für die Bearbeitung ausgewählt:

$$W\{\text{Auswahl Klasse } k \mid n_1 \ldots n_K\} = \begin{cases} 1 & \text{wenn } \sum_{i=1}^{k-1} n_i = 0 \text{ und } n_k > 0 \\ 0 & \text{sonst} \end{cases} \tag{3.6}$$

Die Bedienung von Anforderungen wird jedoch nicht unterbrochen, wenn eine Anforderung mit höherer Priorität eintrifft.

Modell

Abbildung 3.6 zeigt die Markov-Kette für den Systemzustand bei der Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*, Abbildung 3.7 zeigt die Markov-Kette für den Systemzustand bei der Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge*. Unterschiede zwischen den beiden Markov-Ketten gibt es bei den Zuständen 1/1/1, 1/1/2,

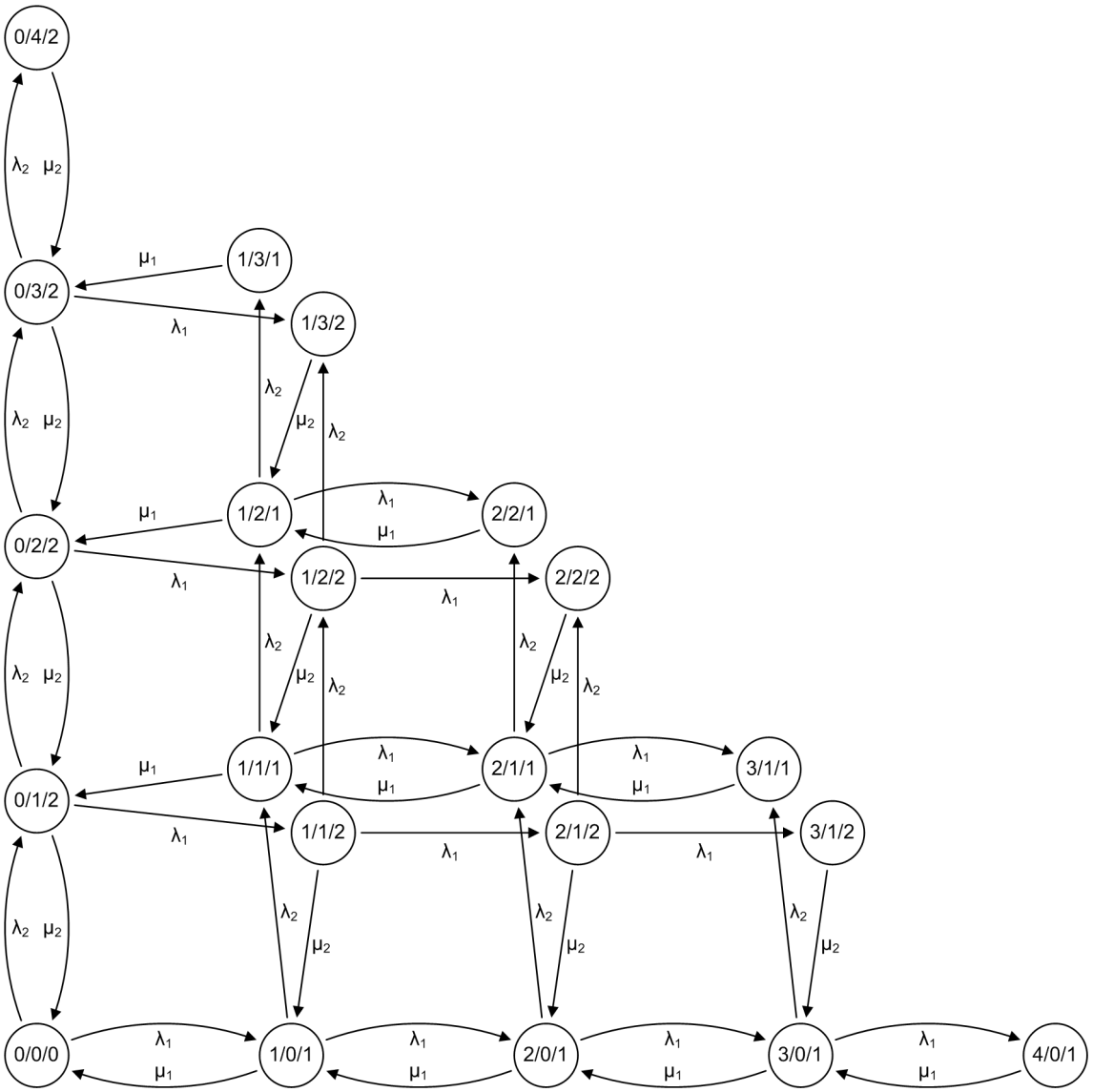


Abbildung 3.6.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit zwei Klassen von Anforderungen, Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*. Beschriftung der Zustände: Anzahl der Klasse 1-Anforderungen im System / Anzahl der Klasse 2-Anforderungen im System / Zustand der Bedieneinheit.

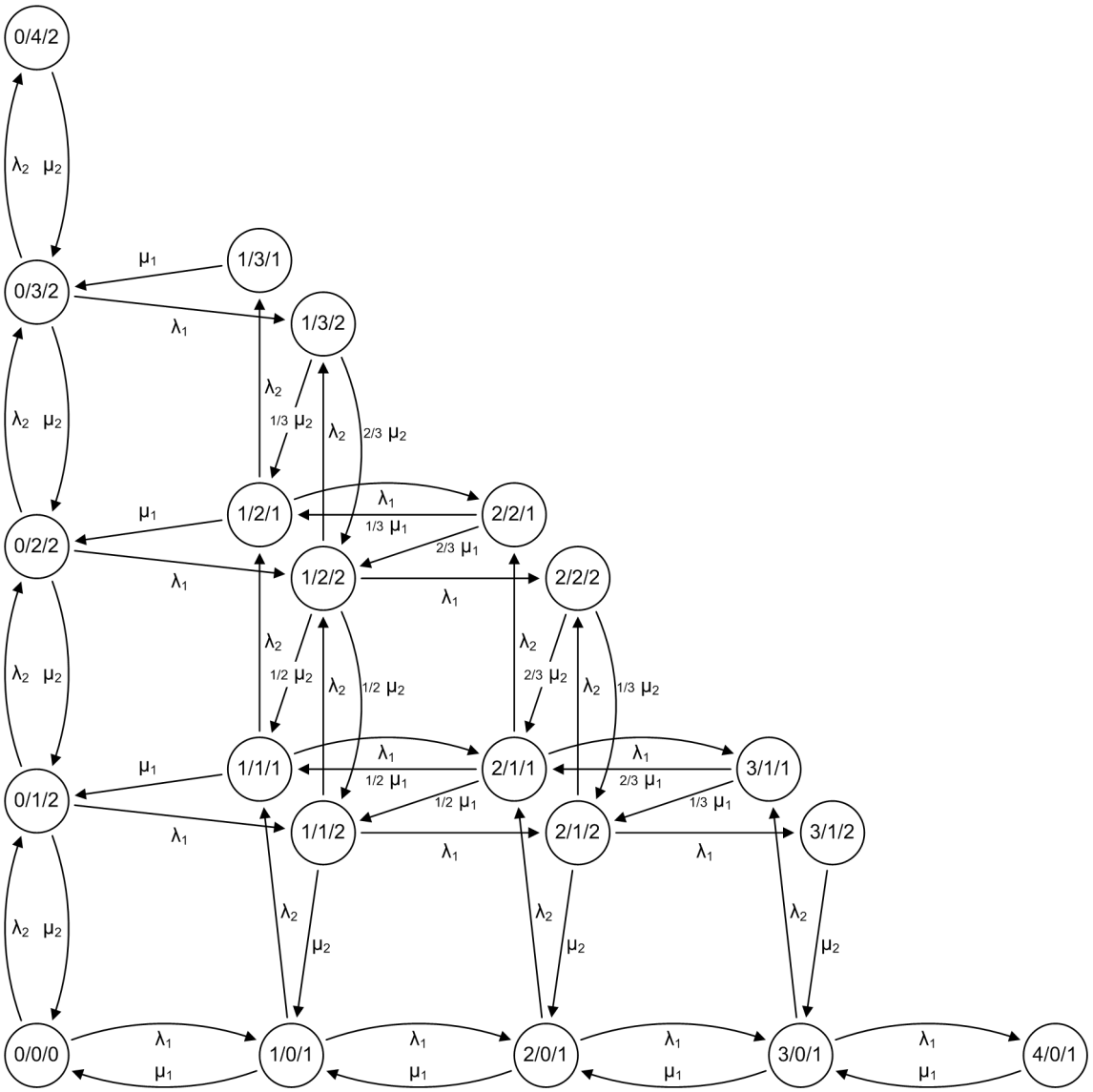


Abbildung 3.7.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit zwei Klassen von Anforderungen, Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge*. Beschriftung der Zustände: Anzahl der Klasse 1-Anforderungen im System / Anzahl der Klasse 2-Anforderungen im System / Zustand der Bedieneinheit.

1/2/1, 1/2/2, 2/1/1 und 2/1/2. Während bei der Abfertigungsdisziplin *Nichtunterbrechende Prioritäten* von diesen Zuständen nur die Zustände 1/1/1, 1/2/1 und 2/1/1 durch den Abschluss der Bedienung einer Anforderung erreicht werden können, kann bei der Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge* jeder dieser Zustände erreicht werden.

Modellbildung am Computer

Ein Zustand ist definiert durch die Anzahl der sich im System befindenden Anforderungen pro Klasse sowie durch den Zustand der Bedieneinheit (das heißt welche Anforderungklasse gerade bedient wird):

Listing 3.1: M/M/1/S-Warteschlangensystem mit mehreren Klassen von Anforderungen: Datenstruktur für einen Zustand

```
#define MAX_CLASS 10         // maximum number of classes

struct TSystemState
{
    unsigned int n[MAX_CLASS + 1];   // number of customers for each class.
    unsigned int k;                  // state of the server
};
```

Damit ein Zustand gültig ist, müssen folgende Bedingungen erfüllt sein:

- Die Anzahl der Anforderungen im System darf nicht größer sein als die Größe des Systems.
- Wenn keine Anforderung der Klasse j im System ist, darf die Bedieneinheit nicht eine Anforderung der Klasse j bedienen.
- Die Bedieneinheit darf nur dann im Leerlauf sein, wenn das System leer ist.

Listing 3.2: M/M/1/S-Warteschlangensystem mit mehreren Klassen von Anforderungen: Erzeugen der Liste der gültigen Zustände

```
std::vector<TSystemState> vecSystemStates;

void M_M_1_S_Prio::InitStateSpace()
{
    TSystemState state;

    for (unsigned int i = 0; i <= MAX_CLASS; state.n[i++] = 0);
    state.k = 0;  // idle state
    vecSystemStates.push_back(state);
    do
    {
        for (state.k = 1; state.k <= iNumberOfClasses; state.k++)
            if (SystemStateIsValid(state))
                vecSystemStates.push_back(state);
        state.n[iNumberOfClasses] ++;
        for (unsigned int j = iNumberOfClasses; j > 1; j--)
            if (state.n[j] > iSystemSize)
            {
                state.n[j] = 0;
                state.n[j - 1] ++;
            }
    }
    while (state.n[1] <= iSystemSize);
    //...
}

bool M_M_1_S_Prio_CommonBuffer_Prio::SystemStateIsValid(const TSystemState & state)
{
    // the number of customers must not exceed the buffer size
    unsigned int iSum=0;
```

```
31      for (unsigned int j=1; j<=iNumberOfClasses; j++)
32          iSum += state.n[j];
33      if (iSum>iSystemSize) return false;
34
35      // if there is no customer of class j in the system, the server
36      // cannot serve a customer of class j
37      for (unsigned int j=1; j<=iNumberOfClasses; j++)
38          if ((state.n[j]==0) && (state.k==j)) return false;
39
40      // if the server is idle, the system must be empty
41      if (state.k==0)
42          for (unsigned int j=1; j<iNumberOfClasses; j++)
43              if (state.n[j]!=0) return false;
44
45      return true;
46  }
```

Zwischen den Zuständen gibt es folgende Übergänge (Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*):

- Eine Anforderung der Klasse i kommt an.
- Eine Anforderung der Klasse j wurde bedient.

Betrachtet man die Spezialfälle *Leerlauf wird erreicht* und *Leerlauf wird verlassen* gesondert, ergeben sich insgesamt vier Möglichkeiten:

- Das System ist im Leerlauf und eine Anforderung der Klasse i kommt an. Das ist der Fall, wenn der Anfangszustand der Leerlauf-Zustand (hat immer die Nummer 0) ist und im Endzustand genau eine Anforderung der Klasse i im System ist. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Ankunftsrate der Klasse i.
- Die letzte Anforderung (der Klasse j) wurde bedient, das System erreicht den Leerlauf. Das ist der Fall, wenn der Endzustand der Leerlauf-Zustand ist und im Anfangszustand genau eine Anforderung der Klasse j im System ist. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Bedienrate der Klasse j.
- Eine Anforderung der Klasse i kommt an. Das ist der Fall, wenn der Zustand der Bedieneinheit im Anfangs- und im Endzustand gleich sind und im Endzustand die Anzahl der Anforderungen der Klasse i um 1 größer ist als im Anfangszustand, während die Anzahlen der Anforderungen aller anderen Klassen gleich sind. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Ankunftsrate der Klasse i.
- Eine Anforderung der Klasse j wird bedient und der neue Zustand der Bedieneinheit ist k. Das ist der Fall, wenn die Anzahl der Anforderungen der Klasse j um 1 kleiner ist als im Anfangszustand, während die Anzahlen der Anforderungen aller anderen Klassen gleich sind, außerdem dürfen im Endzustand keine Anforderungen der Klassen $0, 1, \ldots, k-1$ mehr im System sein, da die Bedienung ja prioritätsgesteuert erfolgt. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Bedienrate der Klasse j.

Bei allen diesen Überlegungen müssen nur gültige Zustände berücksichtigt werden. Beispielsweise muss im ersten Fall nicht zusätzlich geprüft werden, ob im Endzustand die Bedieneinheit auch tatsächlich im Zustand i ist.

Listing 3.3: M/M/1/S-Warteschlangensystem mit mehreren Klassen von Anforderungen: Berechnen der Zustandsübergangsraten

```
REAL M_M_1_S_Prio_CommonBuffer_Prio::GetRateSystemState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest)
{
    TSystemState s = vecSystemStates[iStateNumberSrc];
    TSystemState t = vecSystemStates[iStateNumberDest];

    // idle state, arrival of a customer of class i
    if (iStateNumberSrc==0)
    {
        for (unsigned int i=1; i<=iNumberOfClasses; i++)
        {
            bool bMatch=true;
            for (unsigned int j=1; j<i; j++)
                bMatch = bMatch && (t.n[j]==0);
            bMatch = bMatch && (t.n[i]==1);
            for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
                bMatch = bMatch && (t.n[j]==0);
            if (bMatch) return fArrivalRate[i];
        }
        return 0;
    }

    // the last customer (class i) has been served, idle state is reached
    if (iStateNumberDest==0)
    {
        for (unsigned int i=1; i<=iNumberOfClasses; i++)
        {
            bool bMatch=true;
            for (unsigned int j=1; j<i; j++)
                bMatch = bMatch && (s.n[j]==0);
            bMatch = bMatch && (s.n[i]==1);
            for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
                bMatch = bMatch && (s.n[j]==0);
            if (bMatch) return fServiceRate[i];
        }
        return 0;
    }

    // arrival of a customer of class i
    if (t.k==s.k)
    {
        for (unsigned int i=1; i<=iNumberOfClasses; i++)
            {
                bool bMatch=true;
                for (unsigned int j=1; j<i; j++)
                    bMatch = bMatch && (t.n[j]==s.n[j]);
                bMatch = bMatch && (t.n[i]==s.n[i]+1);
                for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
                    bMatch = bMatch && (t.n[j]==s.n[j]);
                if (bMatch)
                    return fArrivalRate[i];
            }
    }

    // a customer has been served
    {
        bool bMatch=true;
        for (unsigned int j=1; j<s.k; j++)
            bMatch = bMatch && (t.n[j]==s.n[j]);
        bMatch = bMatch && (t.n[s.k]==s.n[s.k]-1);
        for (unsigned int j=s.k+1; j<=iNumberOfClasses; j++)
            bMatch = bMatch && (t.n[j]==s.n[j]);
        if (bMatch)
        {
            for (unsigned int j=1; j<t.k; j++) if (t.n[j]!=0) return 0;
            return fServiceRate[s.k];
        }
    }

    return 0; // there are no other transitions
}
```

Ergebnisse

Der Erwartungswert der Anzahl der Anforderungen für die Klasse 1 ist

$$\begin{aligned} \mathrm{E}[X_1] = \quad & 1 \cdot \left(\pi_{1/0/1} + \pi_{1/1/1} + \pi_{1/1/2} + \pi_{1/2/1} + \pi_{1/2/2} + \pi_{1/3/1} + \pi_{1/3/2}\right) + \\ & 2 \cdot \left(\pi_{2/0/1} + \pi_{2/1/1} + \pi_{2/1/2} + \pi_{2/2/1} + \pi_{2/2/2}\right) + \\ & 3 \cdot \left(\pi_{3/0/1} + \pi_{3/1/1} + \pi_{3/1/2} +\right) + \\ & 4 \cdot \pi_{4/0/1} \end{aligned} \tag{3.7}$$

oder allgemein

$$\mathrm{E}[X_1] = \sum_i \pi_i \cdot (\text{Anzahl der Klasse 1-Anforderungen im Zustand } i) \tag{3.8}$$

Für den Erwartungswert der Anzahl der Anforderungen der Klasse 2 ergibt sich

$$\begin{aligned} \mathrm{E}[X_2] = \quad & 1 \cdot \left(\pi_{0/1/2} + \pi_{1/1/1} + \pi_{1/1/2} + \pi_{2/1/1} + \pi_{2/1/2} + \pi_{3/1/1} + \pi_{3/1/2}\right) + \\ & 2 \cdot \left(\pi_{0/2/2} + \pi_{1/2/1} + \pi_{1/2/2} + \pi_{2/2/1} + \pi_{2/2/2}\right) + \\ & 3 \cdot \left(\pi_{0/3/2} + \pi_{1/3/1} + \pi_{1/3/2} +\right) + \\ & 4 \cdot \pi_{0/4/2} \end{aligned} \tag{3.9}$$

oder allgemein

$$\mathrm{E}[X_2] = \sum_i \pi_i \cdot (\text{Anzahl der Klasse 2-Anforderungen im Zustand } i) \tag{3.10}$$

Die Abbildungen 3.8 und 3.9 zeigen die Erwartungswerte der Anzahl der Anforderungen jeder Klasse im System, wenn die Ankunftsrate von Klasse 1-Anforderungen für kurze Zeit außergewöhnlich hoch ist („Überlastimpuls“).

Die Abbildungen 3.11 und 3.12 zeigen die stationäre Anzahl der Anforderungen im System für jede Klasse in Abhängigkeit von der Ankunftsrate λ_1.

Die Blockierwahrscheinlichkeit ist für alle Klassen gleich (da sich alle Klassen den verfügbaren Platz uneingeschränkt teilen):

$$p_{\text{blocking}} = \pi_{0/4/2} + \pi_{1/3/1} + \pi_{1/3/2} + \pi_{2/2/1} + \pi_{2/2/2} + \pi_{3/1/1} + \pi_{3/1/2} + \pi_{4/0/1} \tag{3.11}$$

oder allgemein

$$p_{\text{blocking}} = \sum_i \pi_i \cdot \begin{cases} 1 & \sum_{j=1}^{K} \text{Anz. der Klasse } j\text{-Anf. im Zustand } i = s \\ 0 & \text{sonst} \end{cases} \tag{3.12}$$

Abbildung 3.10 zeigt die Blockierwahrscheinlichkeit während des Überlastimpulses, Abbildung 3.13 zeigt die Blockierwahrscheinlichkeit in Abhängigkeit von der Ankunftsrate λ_1.

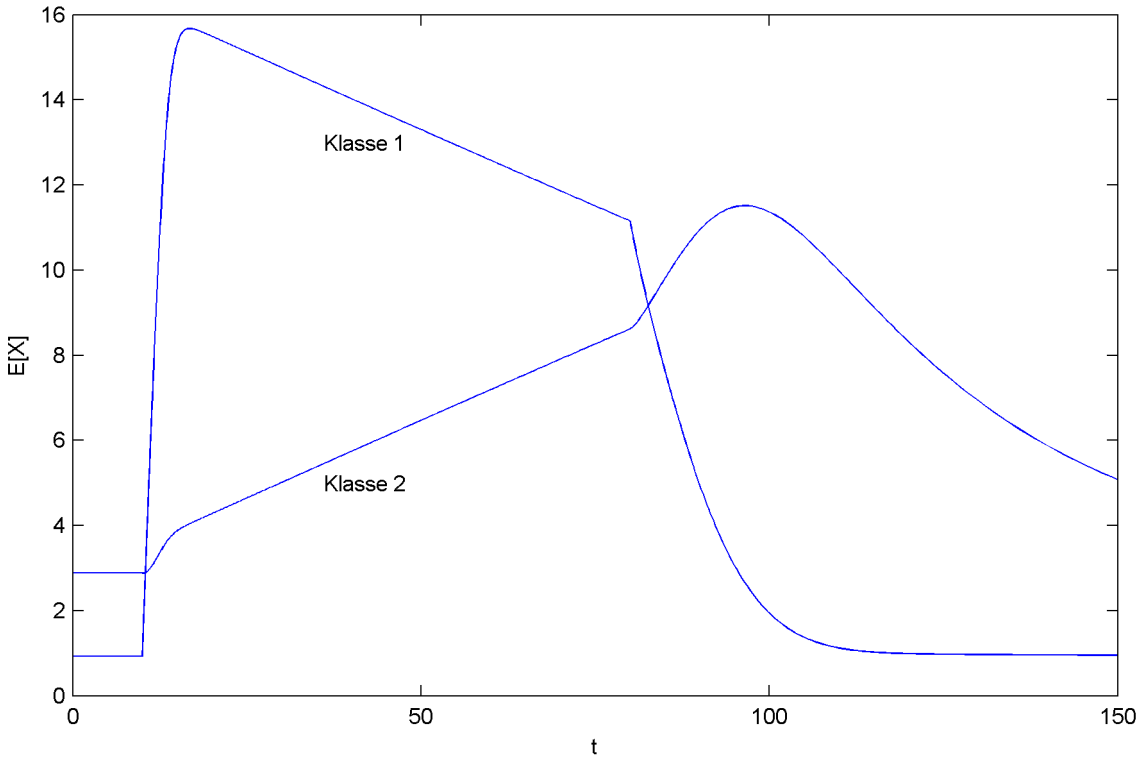


Abbildung 3.8.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Anzahl der Anforderungen im System während eines Überlastimpulses der Klasse 1. $\lambda_1 = 5$ für $10 < t < 80,\ 0.4$ sonst, $\lambda_2 = 0.5,\ \mu = 1$. Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*.

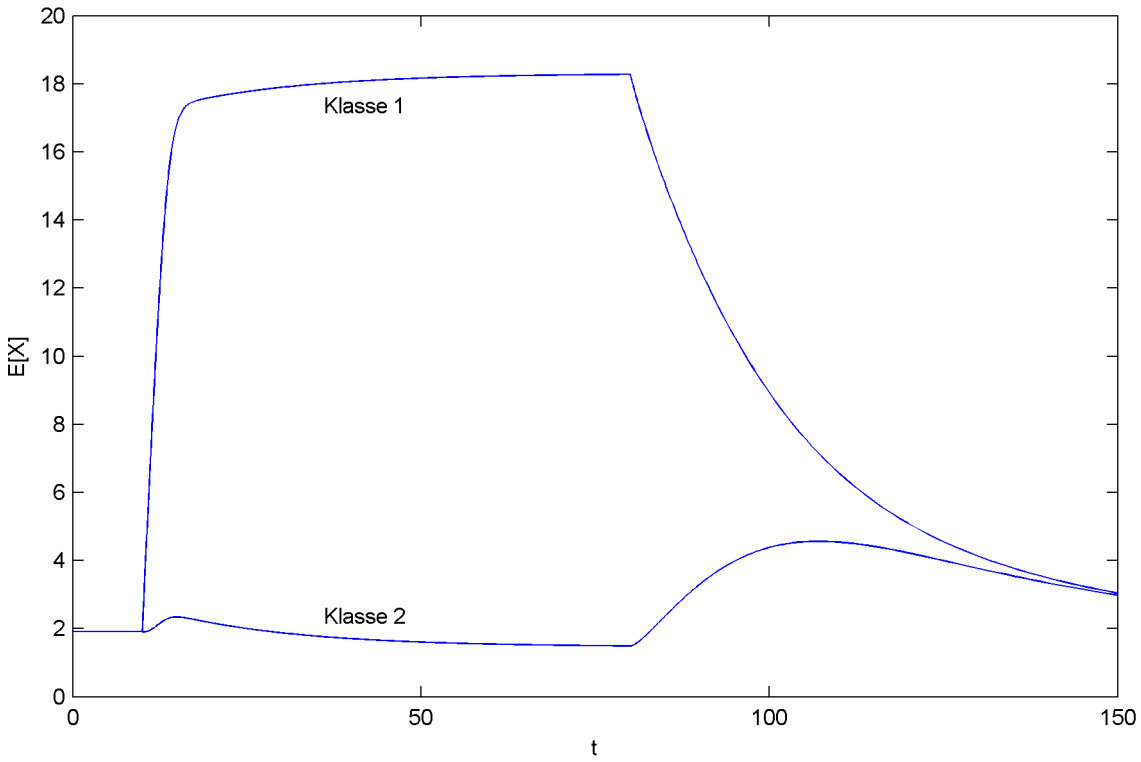


Abbildung 3.9.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Anzahl der Anforderungen im System während eines Überlastimpulses der Klasse 1. $\lambda_1 = 5$ für $10 < t < 80,\ 0.4$ sonst, $\lambda_2 = 0.5,\ \mu = 1$. Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge*.

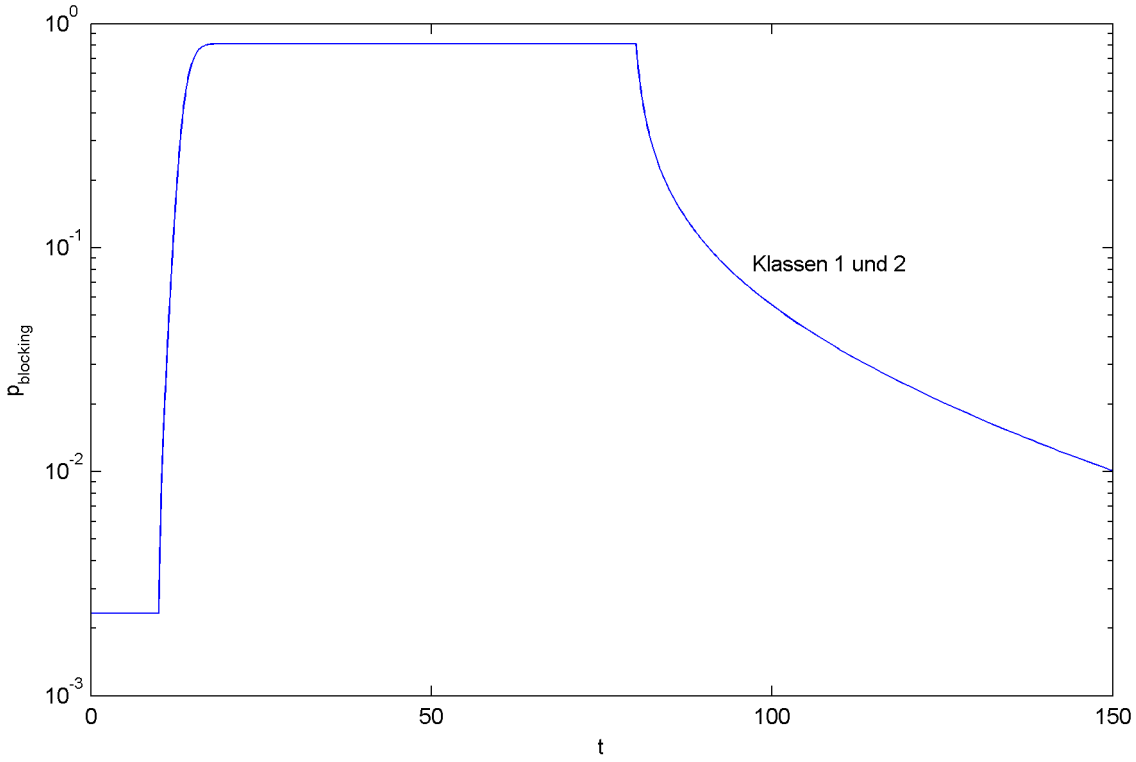


Abbildung 3.10.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Blockierwahrscheinlichkeit während eines Überlastimpulses der Klasse 1. $\lambda_1 = 5$ für $10 < t < 80,\ 0.4$ sonst, $\lambda_2 = 0.5,\ \mu = 1$.

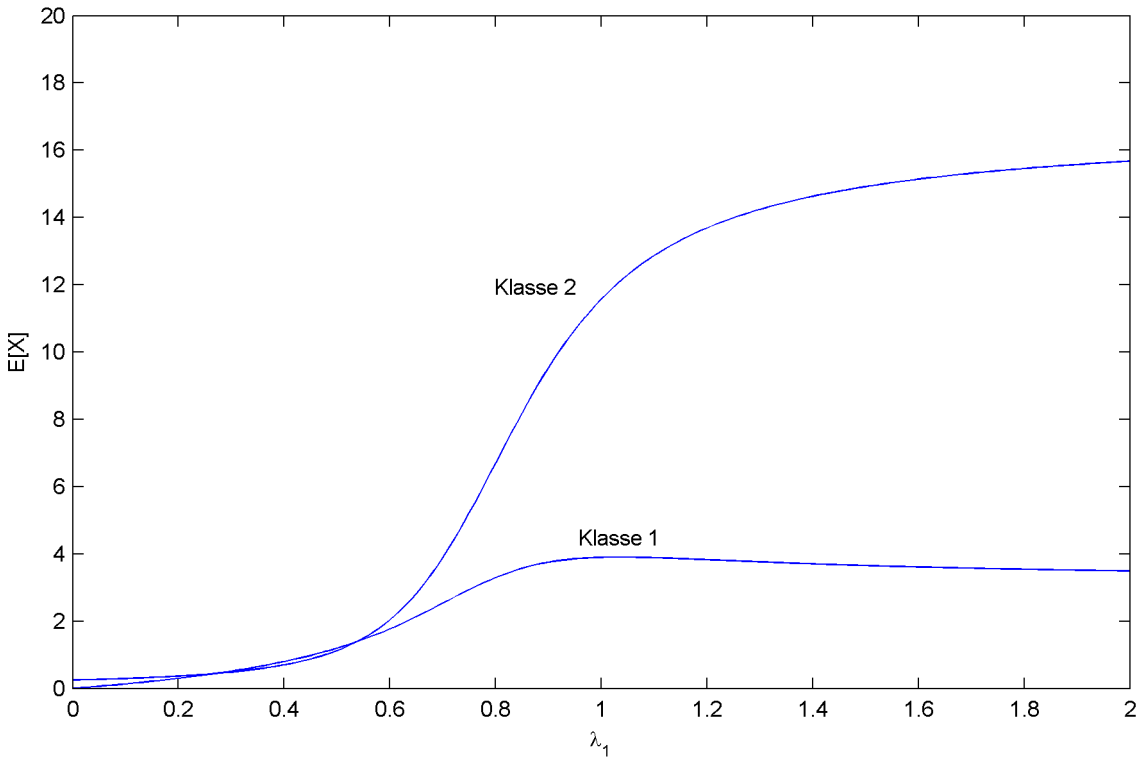


Abbildung 3.11.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Anzahl der Anforderungen im System (stationär) in Abhängigkeit von der Ankunftsrate der Klasse 1. Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*.

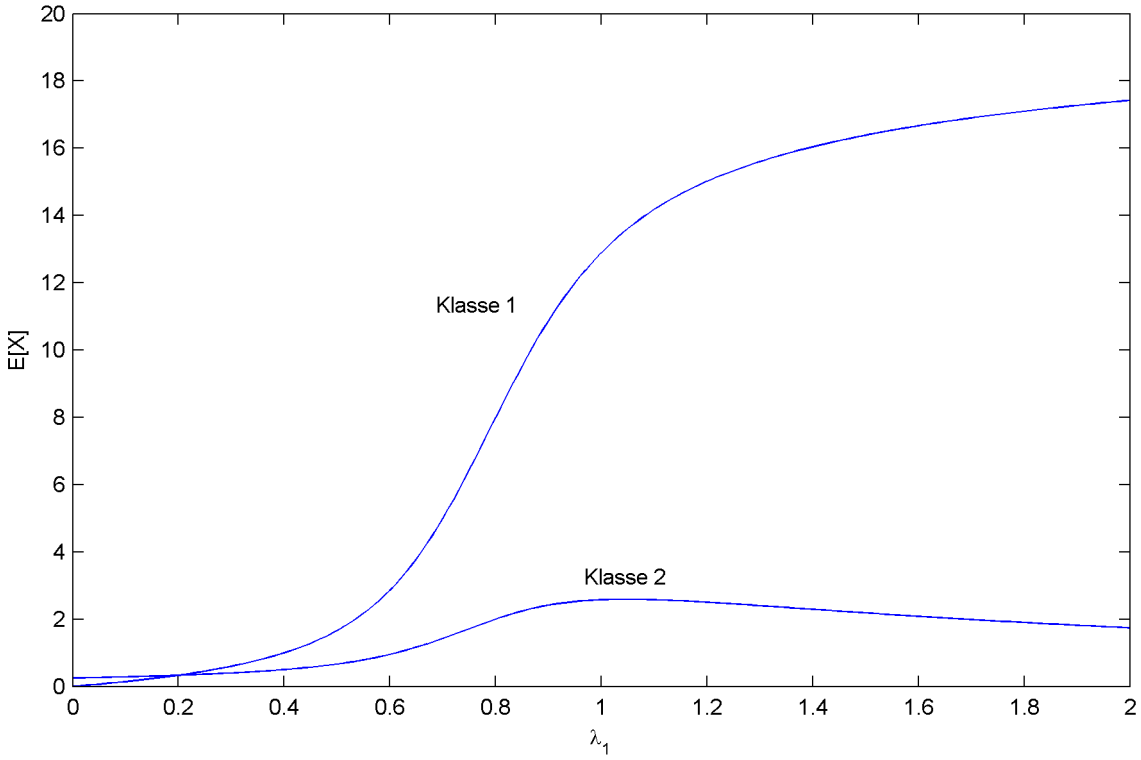


Abbildung 3.12.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Anzahl der Anforderungen im System (stationär) in Abhängigkeit von der Ankunftsrate der Klasse 1. Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge*.

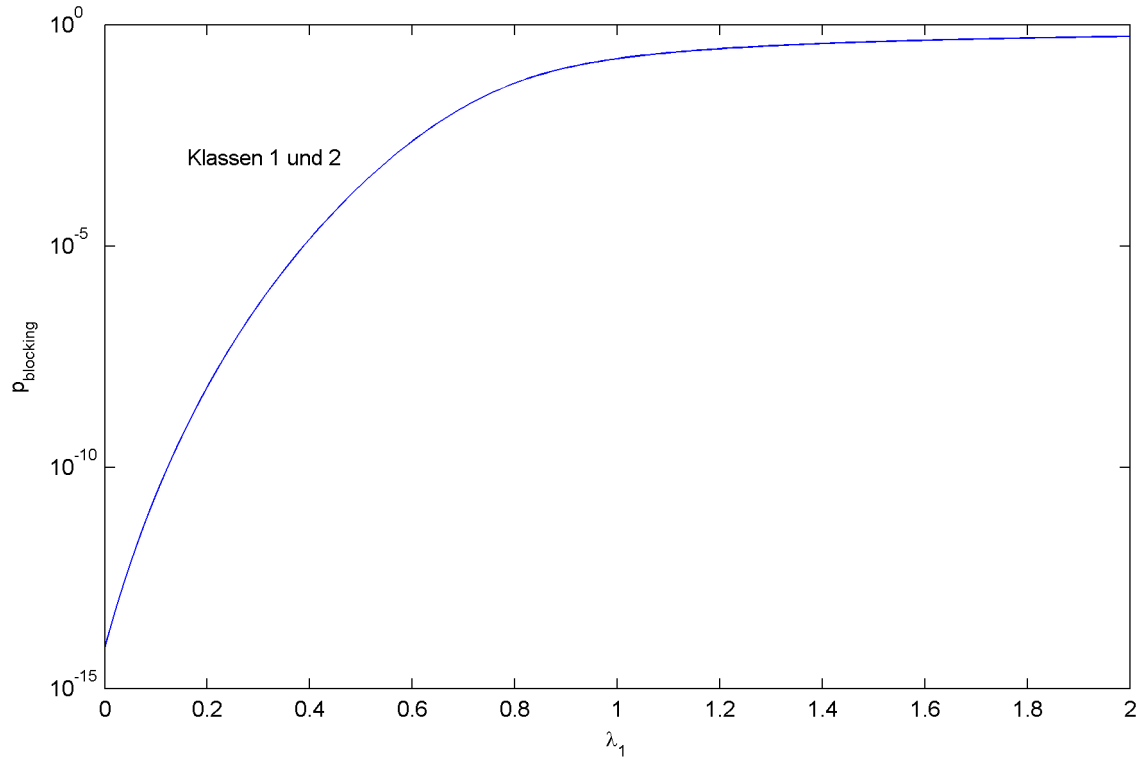


Abbildung 3.13.: M/M/1/S-Warteschlangensystem mit zwei Klassen von Anforderungen: Blockierwahrscheinlichkeit (stationär) in Abhängigkeit von der Ankunftsrate der Klasse 1.

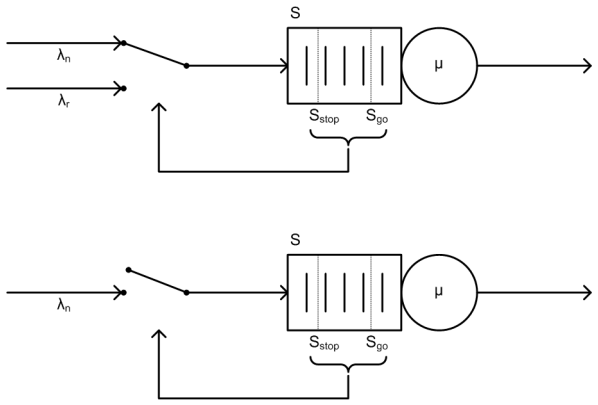


Abbildung 3.14.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate.

3.3.3. M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate

Bei diesem M/M/1/S-Warteschlangensystem (Abbildung 3.14) wird die Ankunftsrate in Abhängigkeit von der Länge der Warteschlange geregelt. Die normale Ankunftsrate ist λ_n. Erreicht die Länge der Warteschlange den Wert S_{stop}, wird auf die reduzierte Ankunftsrate λ_r umgeschaltet, um die Wahrscheinlichkeit für das Vollwerden des Systems zu verringern. Das Zurückschalten auf die normale Ankunftsrate λ_n erfolgt erst, wenn die Warteschlangenlänge den Wert S_{go} erreicht hat. Die Bedienrate ist immer μ.

Modell

Die Markov-Kette für den Systemzustand ist in Abbildung 3.15 gezeigt.

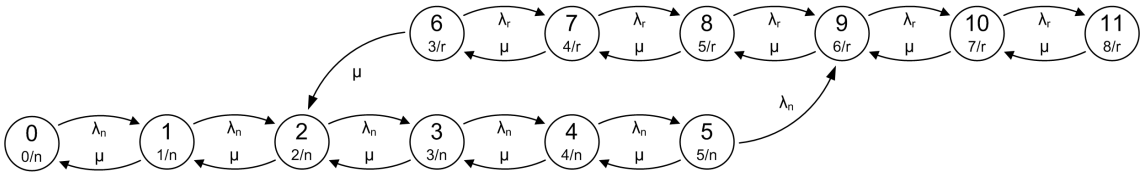


Abbildung 3.15.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate. $S = 8,\ S_{\text{stop}} = 6,\ S_{\text{go}} = 2$. Beschriftung der Zustände: Anzahl der Anforderungen im System / „n“ für normale Ankunftsrate oder „r“ für reduzierte Ankunftsrate.

Modellbildung am Computer

In diesem System ist ein Zustand definiert durch die Anzahl der Anforderungen im System und durch den Zustand der Ankunftsrate (normal/reduziert):

Listing 3.4: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate: Datenstruktur für einen Zustand

```
1 struct TSystemState
2 {
3     unsigned int n;      // number of customers in the system
4     unsigned int s;      // arrival rate: 0 = normal, 1 = reduced
5 };
```

Damit ein Zustand gültig ist, müssen folgende Bedingungen erfüllt sein:

- Die Anzahl der Anforderungen im System darf nicht größer sein als die Größe des Systems.
- Wenn S_{stop} oder mehr Anforderungen im System sind, muss die Ankunftsrate reduziert sein.
- Wenn S_{go} oder weniger Anforderungen im System sind, muss die Ankunftsrate normal sein.

Listing 3.5: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate: Erzeugen der Liste der gültigen Zustände

```
std::vector<struct TSystemState> vecSystemStates;

M_M_1_S_TwoPointA::M_M_1_S_TwoPointA(/*...*/) : CQueueingSystem()
{
    //...

    // create the state space
    struct TSystemState state;
    vecSystemStates.clear();
    for (state.n = 0; state.n <= iSystemSize; state.n++) // number of customers in the system
    {
        state.s = 0;        // arrival rate normal
        if (SystemStateIsValid(state))
            vecSystemStates.push_back(state);
        state.s = 1;        // arrival rate reduced
        if (SystemStateIsValid(state))
            vecSystemStates.push_back(state);
    }

    //...
}


bool M_M_1_S_TwoPointA::SystemStateIsValid(const TSystemState & state)
{
    // ThresholdStop is reached, but the arrival rate is still normal
    if (state.n>=iThresholdStop && state.s==0) return false;
    // ThresholdGo is reached, but the arrival rate is still reduced
    if (state.n<=iThresholdGo && state.s==1) return false;
    // all other states are valid
    return true;
}
```

Zwischen den Zuständen der Markov-Kette gibt es folgende Übergänge:

- Eine Anforderung kommt an und es findet keine Umschaltung statt. Das ist der Fall, wenn im Endzustand die Anzahl der Anforderungen im System um 1 höher ist als im Ausgangszustand und die Ankunftsrate (normal/reduziert) in beiden gleich ist. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die normale/reduzierte Ankunftsrate.
- Eine Anforderung kommt an und es wird auf die reduzierte Ankunftsrate umgeschaltet. Das ist dann der Fall, wenn im Endzustand die Anzahl der Anforderungen im System um 1 höher ist als im Ausgangszustand, im Ausgangszustand die Ankunftsrate normal ist und im Endzustand reduziert. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die normale Ankunftsrate.
- Eine Anforderung wird bedient und es findet keine Umschaltung statt. Das ist der Fall, wenn im Endzustand die Anzahl der Anforderungen im System um 1 kleiner ist als im Ausgangszustand und die Ankunftsrate (normal/reduziert) in

beiden gleich ist. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Bedienrate.

- Eine Anforderung wird bedient und es wird auf die normale Ankunftsrate umgeschaltet. Das ist dann der Fall, wenn im Endzustand die Anzahl der Anforderungen im System um 1 kleiner ist als im Ausgangszustand, im Ausgangszustand die Ankunftsrate reduziert ist und im Endzustand normal. Die Übergangsrate zwischen Anfangs- und Endzustand ist dann die Bedienrate.

Listing 3.6: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate: Berechnen der Zustandsübergangsraten

```
REAL M_M_1_S_TwoPointA::GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
    iStateNumberDest)
{
    TSystemState A = vecSystemStates[iStateNumberSrc];
    TSystemState B = vecSystemStates[iStateNumberDest];

    // arrival of a customer
    if (B.n == A.n+1)
    {
        // normal rate, threshold not reached
        if (A.s==0 && B.s==0 && B.n < iThresholdStop)
            return fArrivalRateNormal;
        // normal rate, threshold reached
        if (A.s==0 && B.s==1 && B.n == iThresholdStop)
            return fArrivalRateNormal;
        // reduced rate
        if (A.s==1 && B.s==1)
            return fArrivalRateReduced;
    }

    // service of a customer
    if (B.n == A.n-1)
    {
        // reduced rate, threshold not reached
        if (A.s==1 && B.s==1 && B.n > iThresholdGo)
          return fServiceRate;
        // reduced rate, threshold reached
        if (A.s==1 && B.s==0 && B.n == iThresholdGo)
          return fServiceRate;
        // normal rate
        if (A.s==0 && B.s==0)
          return fServiceRate;
    }

    // there are no other transitions
    return 0;
}
```

Ergebnisse

Der Erwartungswert der Anzahl der Anforderungen im System ist

$$\begin{aligned} \mathrm{E}[X] = \; & 1 \cdot \pi_{1/n} + 2 \cdot \pi_{2/n} + 3 \cdot \left(\pi_{3/n} + \pi_{3/r}\right) + \\ & 4 \cdot \left(\pi_{4/n} + \pi_{4/r}\right) + 5 \cdot \left(\pi_{5/n} + \pi_{5/r}\right) + \\ & 6 \cdot \pi_{6/r} + 7 \cdot \pi_{7/r} + 8 \cdot \pi_{8/r} \end{aligned} \tag{3.13}$$

oder allgemein

$$\mathrm{E}[X] = \sum_{i=0}^{S_{\text{stop}}-1} \pi_{i/n} + \sum_{i=S_{\text{go}}+1}^{s} \pi_{i/r} \tag{3.14}$$

Die Wahrscheinlichkeit, dass die Ankunftsrate reduziert ist, ist

$$p_{\text{reduced}} = \pi_{3/r} + \pi_{4/r} + \pi_{5/r} + \pi_{6/r} + \pi_{7/r} + \pi_{8/r} \tag{3.15}$$

oder allgemein

$$p_{\text{reduced}} = \sum_{i=S_{\text{go}}+1}^{s} \pi_{i/r} \tag{3.16}$$

Die Auslastung der Bedieneinheit ist wieder

$$\rho = 1 - \pi_0 \tag{3.17}$$

3.3.4. M/M/1/S Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten

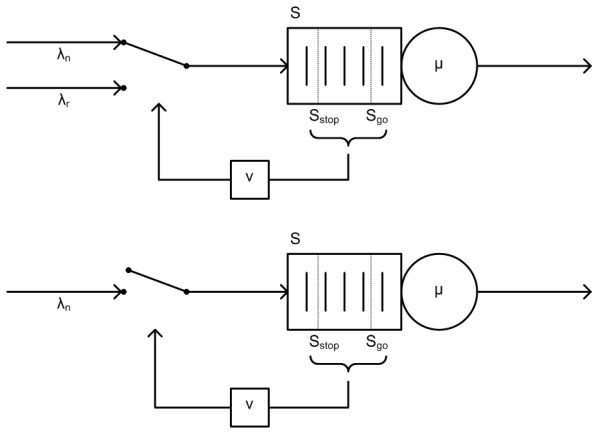


Abbildung 3.16.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten.

Dieses Warteschlangensystem (Abbildung 3.16) ist eine Erweiterung des zuletzt gezeigten Systems. Das Warteschlangensystem sendet nun bei Erreichen der beiden Grenzwerte Steuernachrichten zur Quelle, um das Ein- beziehungsweise Ausschalten des Datenflusses anzuregen. Die Zeit, die diese Nachrichten benötigen, um vom Warteschlangensystem zur Quelle zu gelangen, ist exponentialverteilt. Die Übertragungsrate dieser Nachrichten wird mit ν bezeichnet.

Modell

Die Markov-Kette für den Systemzustand dieses Warteschlangensystems ist in Abbildung 3.17 zu sehen.

Bei diesem Warteschlangensystem kommt es zu Problemen, wenn die maximale Anzahl von Steuernachrichten, die gleichzeitig übertragen werden kann, ungerade ist: Abbildung 3.18 zeigt die Markov-Kette, wenn die maximale Anzahl 3 ist. Ist nun die reduzierte Ankunftsrate gleich 0, so entsteht daraus das in Abbildung 3.19 gezeigte Modell mit dem absorbierenden Zustand 1/0/1! Setzt man hingegen bei der in Abbildung 3.17 gezeigten Markov-Kette (die maximale Anzahl ist hier 4) die reduzierte Ankunftsrate auf 0, dann entsteht kein absorbierender Zustand (Abbildung 3.20). Das heißt, wenn im Warteschlangensystem die maximale Anzahl von Steuernachrichten, die gleichzeitig übertragen werden kann, unendlich ist, dann muss im Modell für diese Anzahl ein gerader Wert angenommen werden.

Ergebnisse

Das Berechnen des Erwartungswerts der Anzahl der Anforderungen im System und der Blockierwahrscheinlichkeit funktionieren analog zu den bisher gezeigten Systemen, daher werden die Formeln hier nicht angegeben.

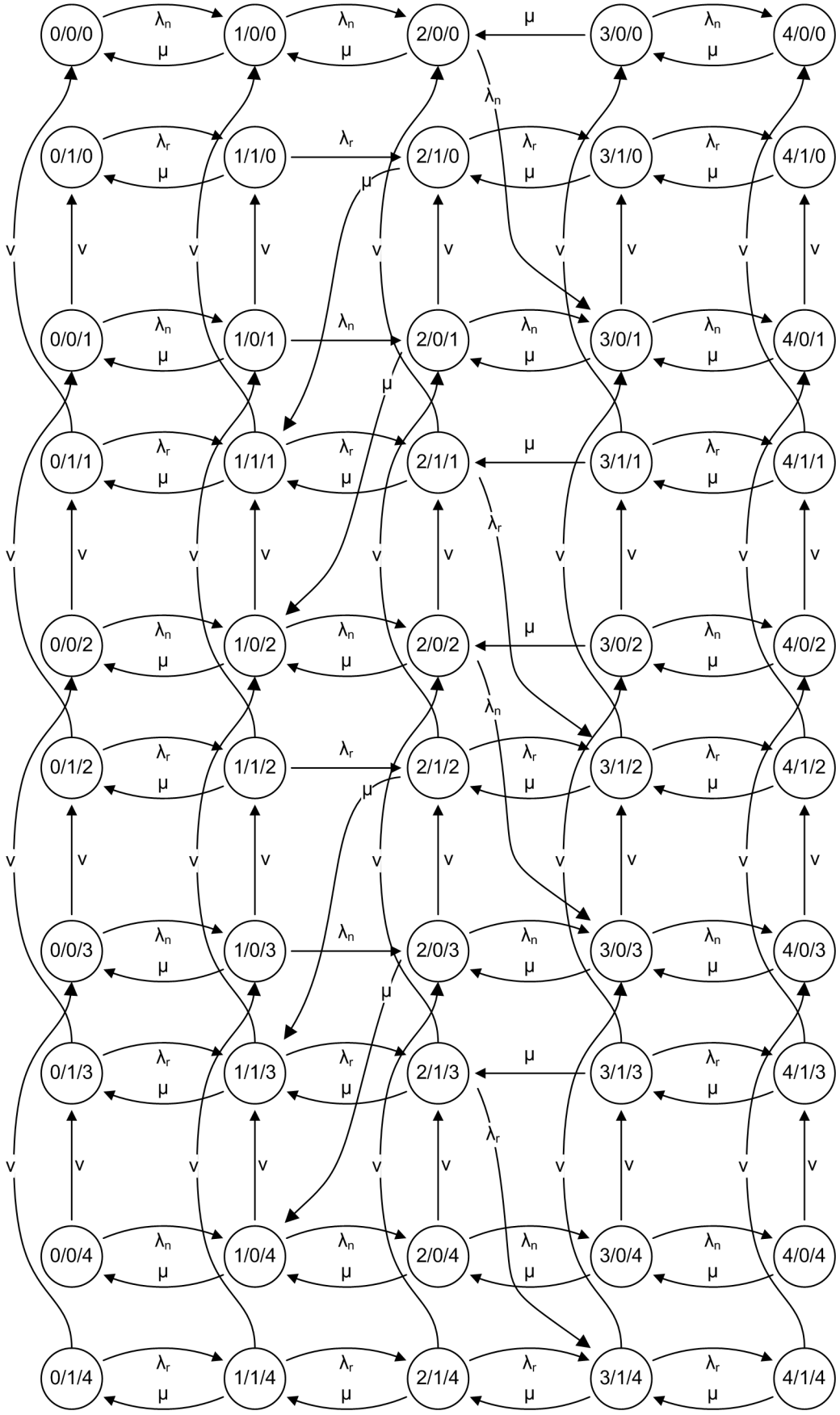


Abbildung 3.17.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten.

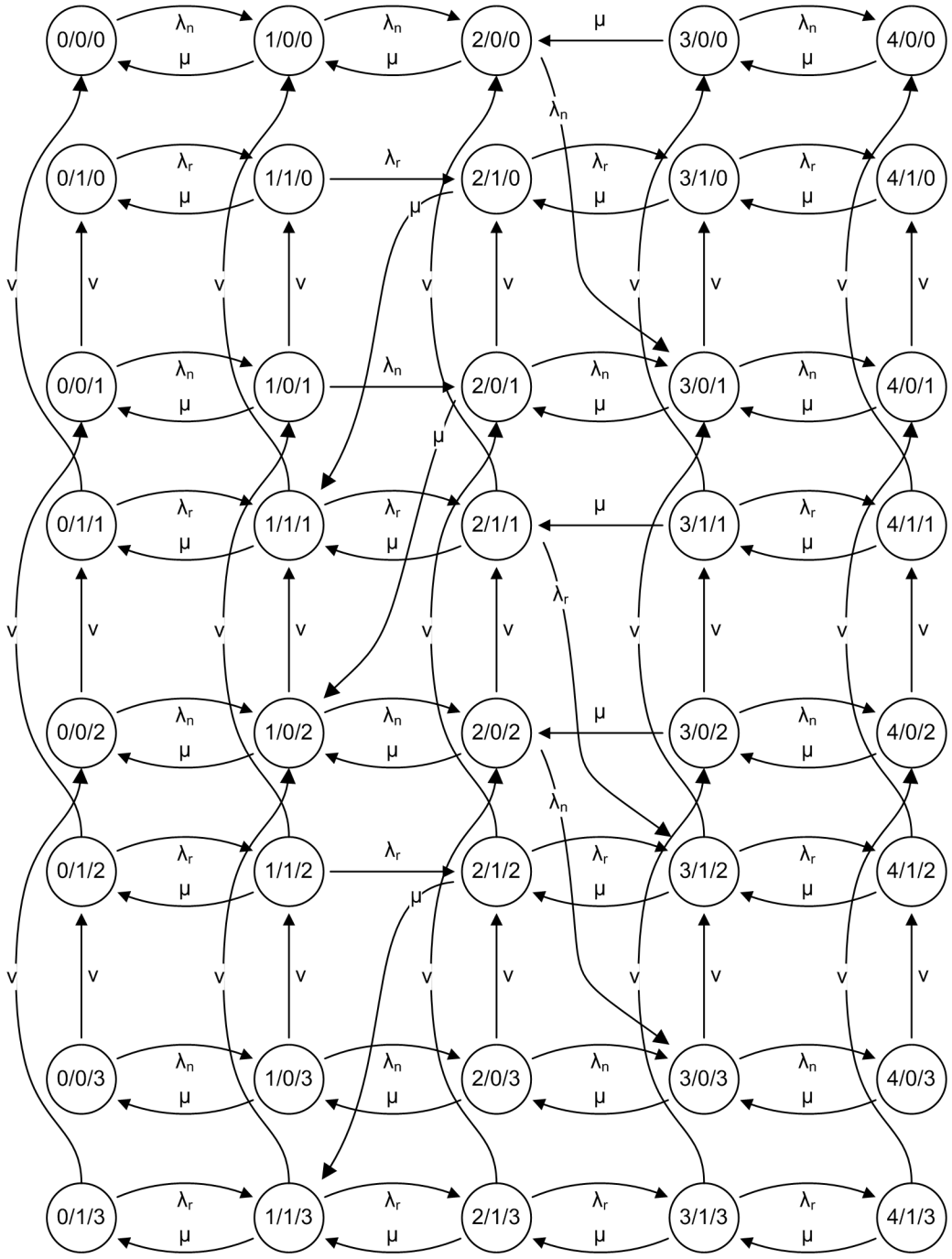


Abbildung 3.18.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten.

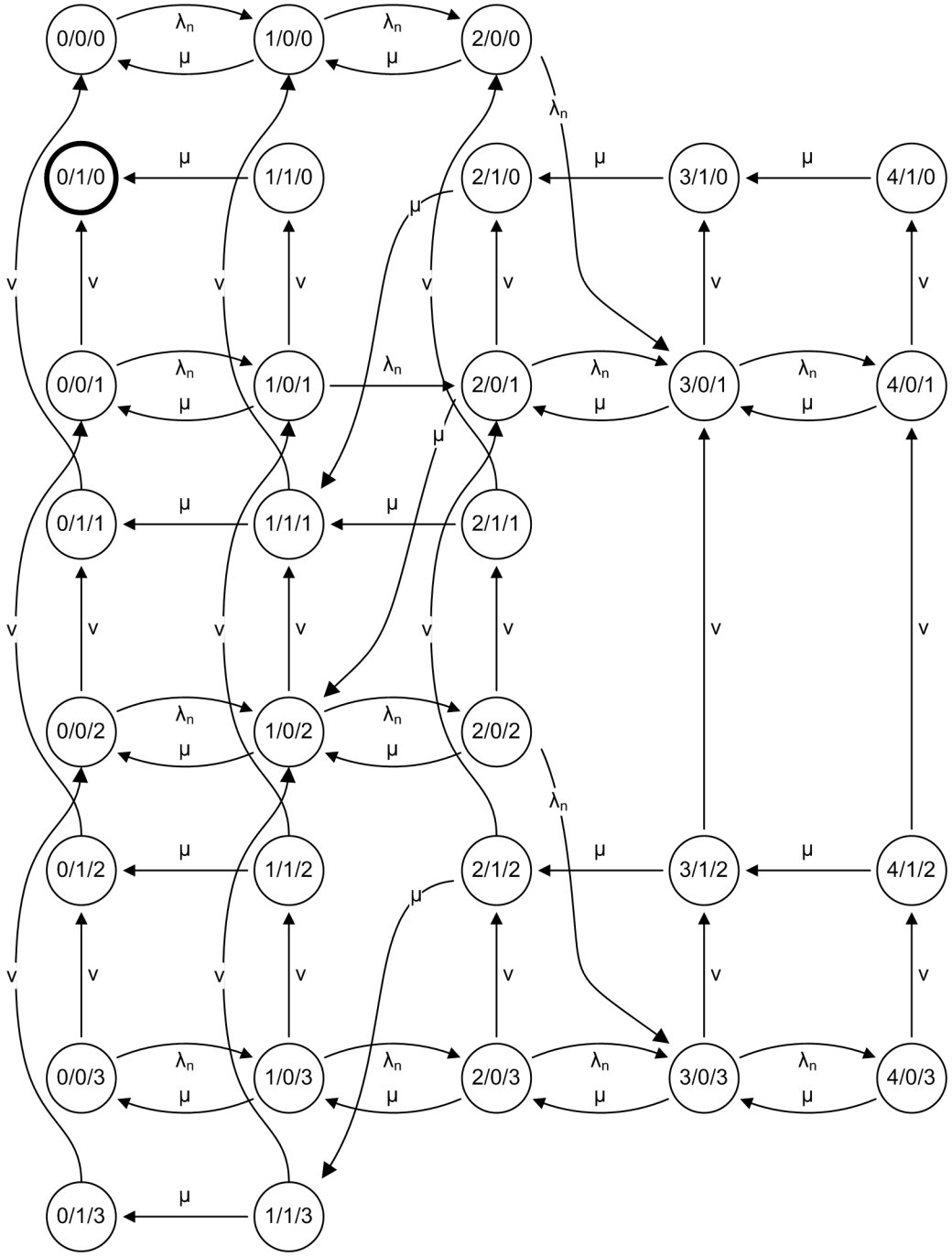


Abbildung 3.19.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate ($\lambda_r = 0$) und verzögerter Übertragung der Steuernachrichten.

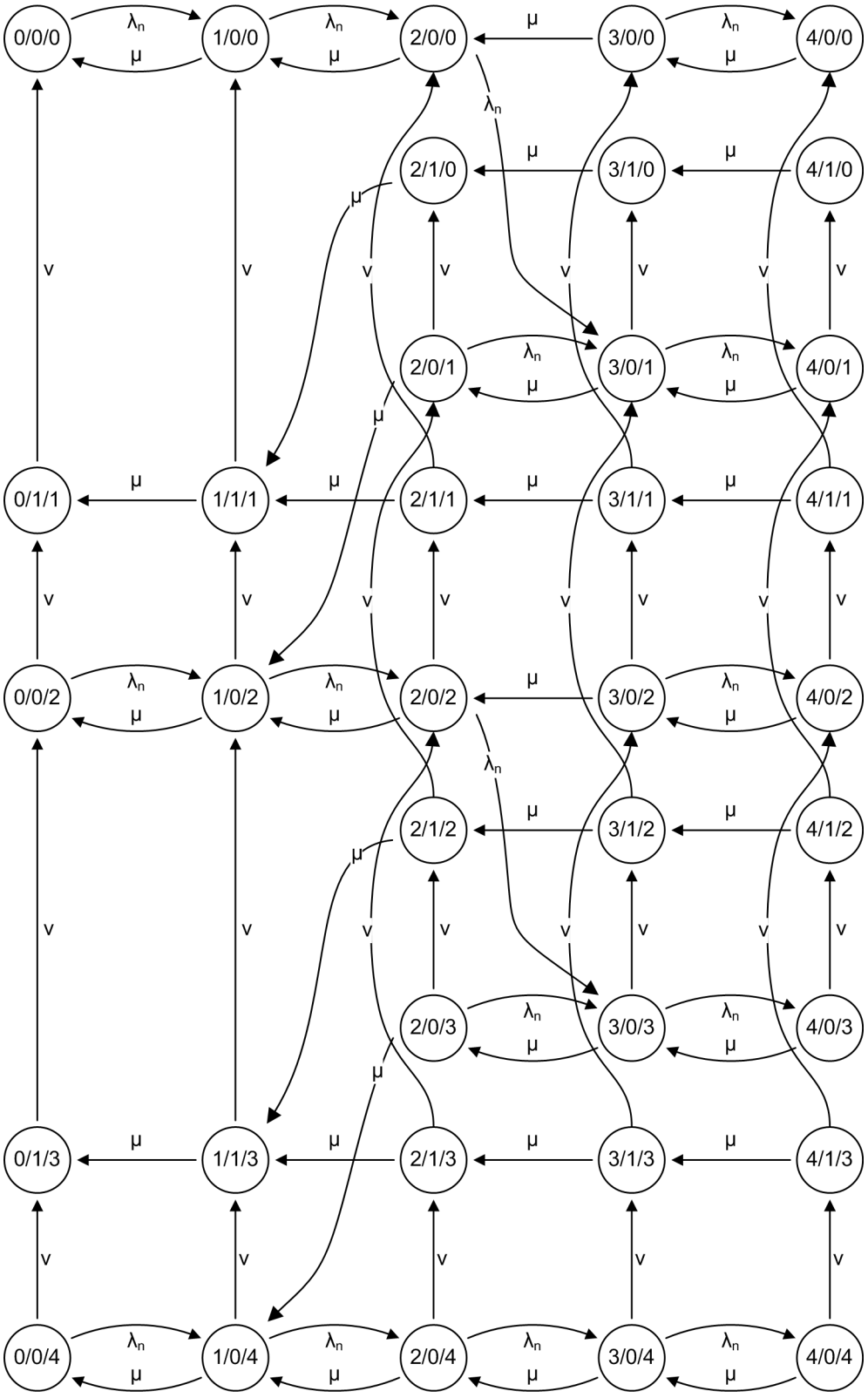


Abbildung 3.20.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate ($\lambda_r = 0$) und verzögerter Übertragung der Steuernachrichten.

Die Abbildungen 3.21 und 3.22 zeigen den Erwartungswert der Anzahl der Anforderungen und die Blockierwahrscheinlichkeit während eines Überlastimpulses für verschiedene Werte von S_{go}.

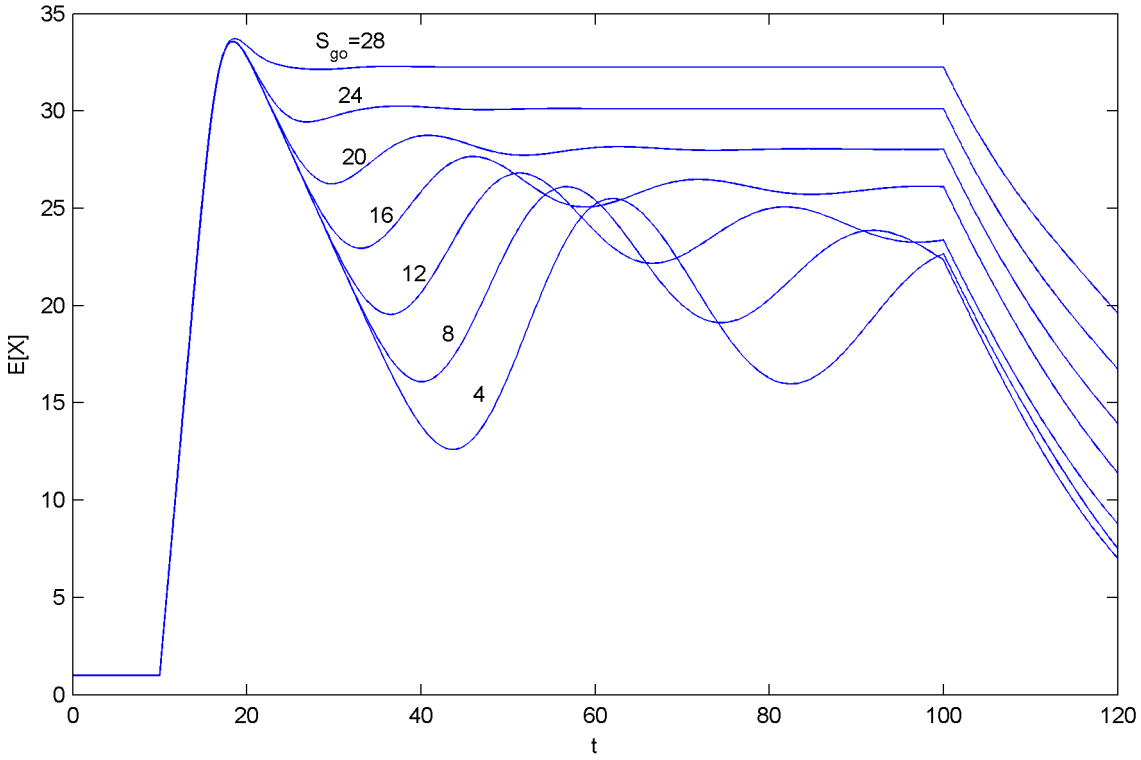


Abbildung 3.21.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten: Anzahl der Anforderungen im System während eines Überlastimpulses. $S = 40$, $S_{\text{stop}} = 32$, $\lambda_n = 6$ für $10 < t < 100$, 0.5 sonst, $\lambda_r = 0$, $\mu = \nu = 1$.

Die Abbildungen 3.23 und 3.24 zeigen den Erwartungswert der Anzahl der Anforderungen und die Blockierwahrscheinlichkeit während eines Überlastimpulses für verschiedene Übertragungsraten der Steuernachrichten (ν). Wenn $\nu = \infty$ gesetzt wird (gestrichelte Kurve), werden die Steuernachrichten in unendlich kurzer Zeit übertragen. Das System entspricht somit dem im letzten Unterabschnitt vorgestellten System.

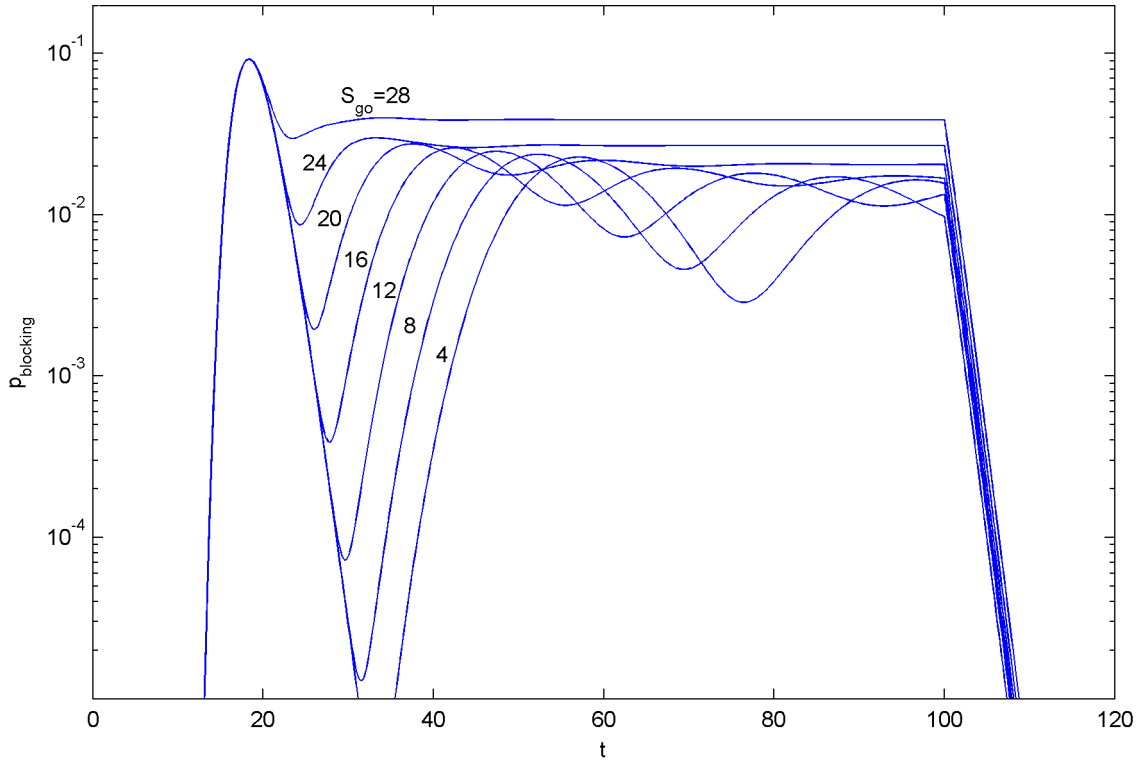


Abbildung 3.22.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten: Blockierwahrscheinlichkeit während eines Überlastimpulses. $S = 40,\ S_{\text{stop}} = 32,\ \lambda_n = 6$ für $10 < t < 100,\ 0.5$ sonst, $\lambda_r = 0,\ \ \mu = \nu = 1.$

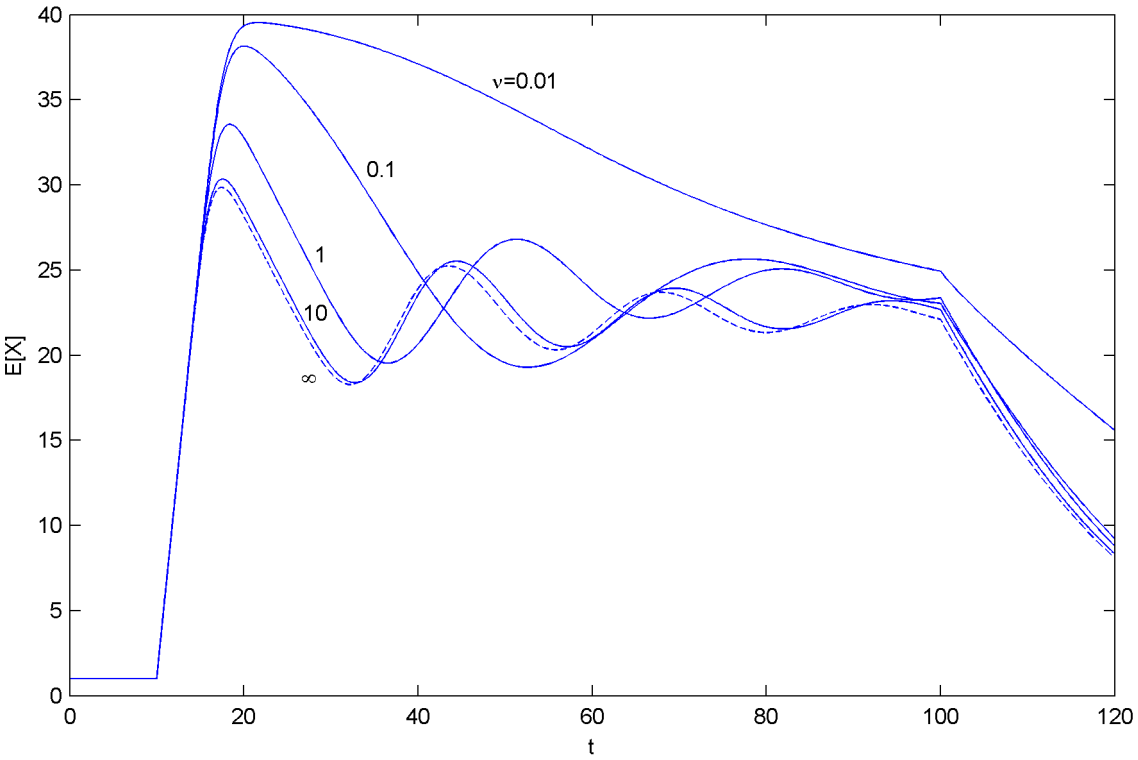


Abbildung 3.23.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten: Anzahl der Anforderungen im System während eines Überlastimpulses. $S = 40,\ S_{\text{stop}} = 32,\ S_{\text{go}} = 12,\ \lambda_n = 6$ für $10 < t < 100,\ 0.5$ sonst, $\lambda_r = 0,$ $\mu = 1.$

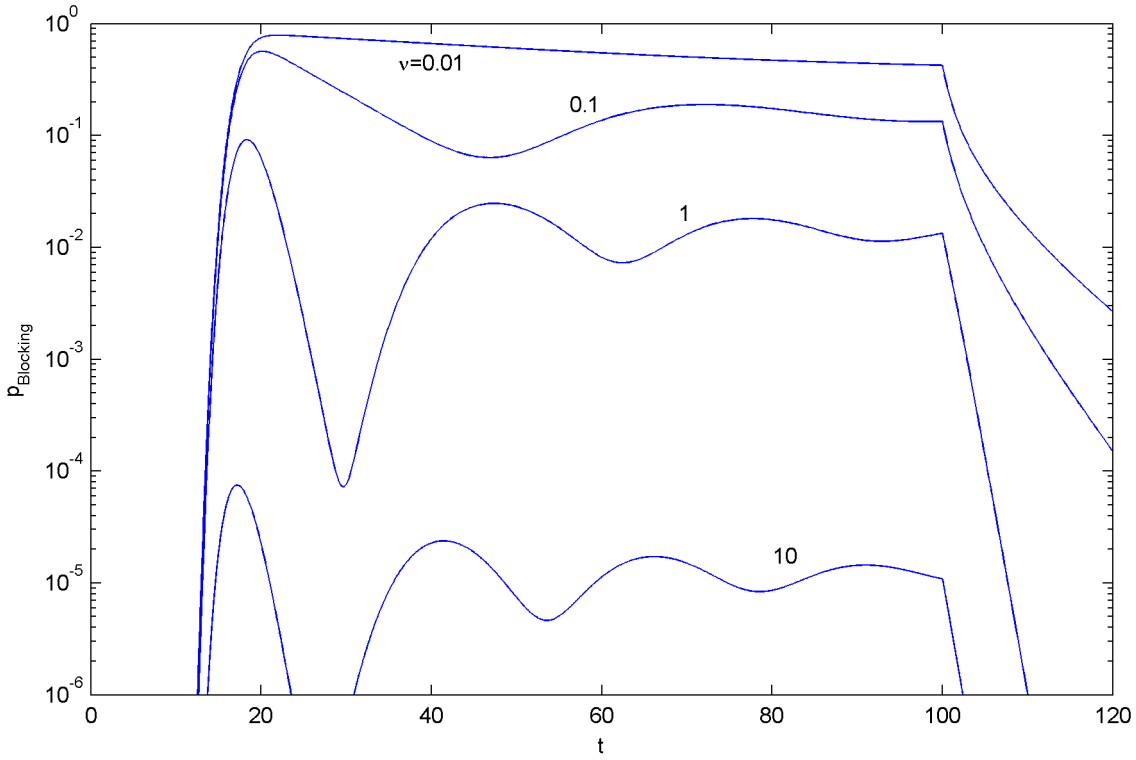


Abbildung 3.24.: M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Übertragung der Steuernachrichten: Blockierwahrscheinlichkeit während eines Überlastimpulses. $S = 40,\ S_{\text{stop}} = 32,\ S_{\text{go}} = 12,\ \lambda_n = 6$ für $10 < t < 100,\ 0.5$ sonst, $\lambda_r = 0,\ \ \mu = 1.$

3.3.5. Netz aus M/M/1/S-Warteschlangensystemen

Zuletzt werden einfache Netze aus M/M/1/S-Warteschlangensystemen mit mehreren Klassen von Anforderungen betrachtet. Die Systeme sind dabei so vernetzt, dass eine Anforderung, die in einem System i bedient wurde, mit der Wahrscheinlichkeit q_{i0} das Netz verläßt oder mit der Wahrscheinlichkeit q_{ij} zu einem anderen System j weitergeleitet wird. Ist das System j voll, kann die Bedienung der Anforderung im System i nicht abgeschlossen werden, nachfolgende Anforderungen werden dann blockiert. Neue Anforderungen können bei jedem System ins Netz eintreten (mit der Ankunftsrate λ_i für das System i).

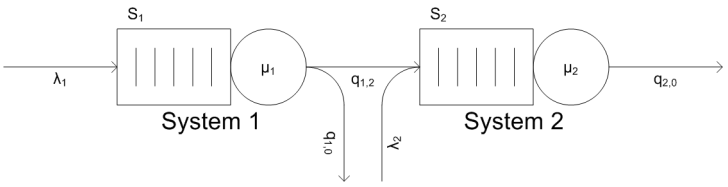


Abbildung 3.25.: Einfaches Netz aus M/M/1/S-Warteschlangensystemen („Tandemsystem“).

Abbildung 3.25 zeigt ein aus zwei Warteschlangensystemen bestehendes Netz. Die Wahrscheinlichkeit, dass eine im System 1 bediente Anforderung das Netz verläßt, ist $q_{1,0}$. Die Wahrscheinlichkeit, dass eine bediente Anforderung ins System 2 weitergeleitet wird, ist $q_{1,2}$. Im System 2 bediente Anforderungen verlassen immer das System. Neue Anforderungen kommen an beiden Systemen an, und zwar mit der Ankunftsrate λ_1 am System 1 und mit der Ankunftsrate λ_2 am System 2.

Modell

Die Markov-Kette für den Systemzustand eines Netzes besteht aus allen möglichen Kombinationen von Zuständen der Markov-Ketten für den Systemzustand der einzelnen Systeme. Die Übergangsraten werden aus den einzelnen Markov-Ketten übernommen und dann so abgeändert, dass das Weiterleiten zwischen den einzelnen Systemen berücksichtigt wird.

Abbildung 3.26 zeigt beispielsweise die Markov-Kette für den Systemzustand des Netzes aus Abbildung 3.25. Hier enthält jede Zeile die Zustände der Markov-Kette für den Systemzustand des ersten Systems und jede Spalte enthält die Zustände der Markov-Kette für den Systemzustand des zweiten Systems. Die Bedienung einer Anforderung im System 1 führt hier jedoch mit der Wahrscheinlichkeit $q_{1,2}$ zu einer Ankunft im System 2.

Modellbildung am Computer

Zunächst wird für jedes System die Liste der gültigen Zustände erstellt. Dann werden alle Kombinationen von Zuständen aus den einzelnen Listen gebildet, das sind die Zustände des Netzes.

Ein Zustandswechsel zwischen zwei Zuständen ist genau dann möglich, wenn eine der folgenden Bedingungen zutrifft:

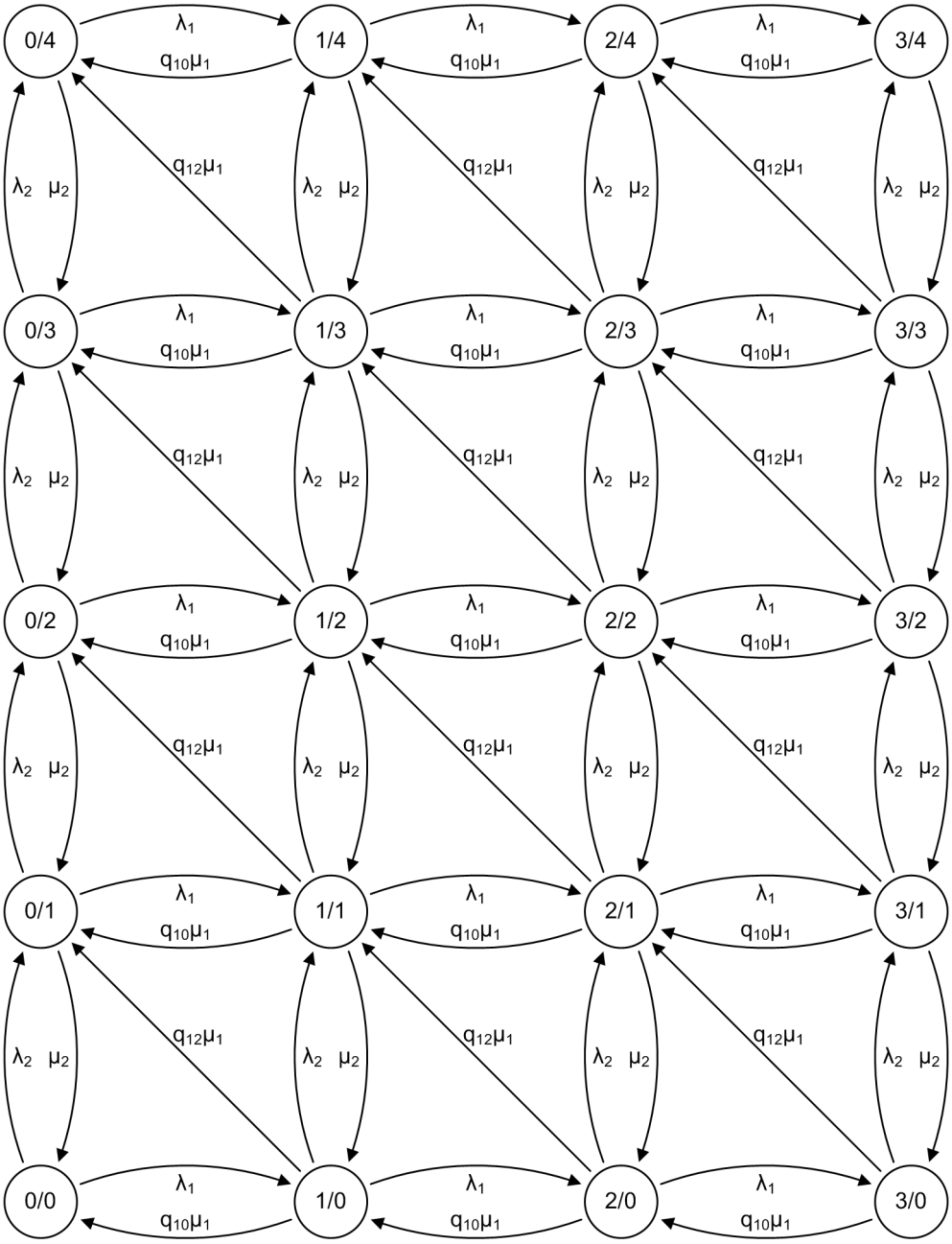


Abbildung 3.26.: Markov-Kette für den Systemzustand eines einfachen Netzes aus M/M/1/S-Warteschlangensystemen (Tandemsystem). $S_1 = 3,\ S_2 = 4$.

- Bei einem System i wird die Bedienung einer Anforderung abgeschlossen, bei einem anderen System j trifft eine Anforderung der bearbeiteten Klasse ein, die Routingwahrscheinlichkeit q_{ij} zwischen System i und System j ist größer als 0 und alle anderen Systeme bleiben unverändert.

- Bei einem System i wird die Bearbeitung einer Anforderung abgeschlossen, die Wahrscheinlichkeit, dass das Netz verlassen wird, q_{i0} ist größer als 0 und alle anderen Systeme bleiben unverändert.

- Bei einem System trifft eine Anforderung ein, die Ankunftsrate von neuen Anforderungen λ_i ist größer als 0 und alle anderen Systeme bleiben unverändert.

In den ersten beiden Fällen setzt sich die Übergangsrate zusammen aus der Bedienrate μ_i des bedienenden Systems, der Routingwahrscheinlichkeit q_{ij} bzw. q_{i0} und der von der Abfertigungsdisziplin abhängigen Wahrscheinlichkeit, dass das System i nach der Bedienung den festgestellten neuen Zustand einnimmt. Im dritten Fall entspricht die Übergangsrate der Ankunftsrate von neuen Anforderungen.

Näherungsverfahren

Die Anzahl der Kombinationen von Zuständen der einzelnen Systeme und damit die Anzahl der Zustände im Netz wächst mit steigender Netzgröße sehr rasch an. Beispielsweise hat ein aus zwei M/M/1/S-Warteschlangensystemen mit $S = 10$ bestehendes Netz schon $11 \cdot 11 = 121$ Zustände, ein aus drei solchen Systemen bestehendes Netz hat $11 \cdot 11 \cdot 11 = 1331$ Zustände. Nimmt man auch noch an, dass es zwei Klassen von Anforderungen gibt, haben die Netze $111 \cdot 111 = 12321$ beziehungsweise $111 \cdot 111 \cdot 111 = 1367631$ Zustände. Aus diesem Grund können nur sehr kleine Netze mit der hier beschriebenen Methode untersucht werden.

Um auch größere Netze untersuchen zu können, wurde folgendes Näherungsverfahren entwickelt:

Die einzelnen Warteschlangensysteme werden unabhängig voneinander betrachtet. Die Wechselwirkungen im Netz werden nachgebildet, indem die Bedienraten und Ankunftsraten der einzelnen Systeme verändert werden: Die Auslastung eines Systems hat Einfluss auf die Rate, mit der Anforderungen an nachfolgende Systeme weitergeleitet werden. Je höher die Auslastung ist, desto höher ist die Ankunftsrate an den nachfolgenden Systemen. Die Blockierwahrscheinlichkeit eines Systems hat Einfluss auf die Bedienraten aller Systeme, die Anforderungen an dieses System weiterleiten. Je höher die Blockierwahrscheinlichkeit ist, desto höher ist die Wahrscheinlichkeit, dass die Bedieneinheit eines vorangehenden Systems blockiert wird, wodurch dessen effektive Bedienrate geringer wird. Da die Änderungen an einem System über Umwege auf das System selbst rückwirken können, muss dieser Vorgang mehrmals wiederholt werden, bis sich die Werte der einzelnen Systeme nicht mehr ändern.

Die Wechselwirkungen wurden nun folgendermaßen nachgebildet:

Ausgangsprozess, eine Anforderungsklasse: Wenn ein System i im Leerlauf ist, ist die Ausgangsrate δ_i gleich 0, andernfalls ist die Ausgangsrate gleich der Bedienrate. Die Ausgangsrate wurde daher durch folgende Formel angenähert:

$$\delta_i = \mu_i(1 - p_{i,\text{idle}}) \tag{3.18}$$

Nachfolgende Systeme j erhalten einen entsprechend der Routingwahrscheinlichkeit gewichteten Anteil:

$$\lambda_{j,\text{gesamt}} = \lambda_j + \sum_i \delta_i \; q_{ij} \tag{3.19}$$

Ausgangsprozess, mehrere Anforderungsklassen: Wenn ein System im Leerlauf ist oder eine andere Klasse als die Klasse k bearbeitet, ist die Ausgangsrate $\delta_{i,k}$ gleich 0, andernfalls ist die Ausgangsrate gleich der Bedienrate. Die Ausgangsrate wurde daher durch folgende Formel angenähert:

$$\delta_{i,k} = \mu_{i,k} \; p_{i,\text{Klasse } k \text{ wird bedient}} \tag{3.20}$$

Nachfolgende Systeme j erhalten wieder einen entsprechend der Routingwahrscheinlichkeit gewichteten Anteil:

$$\lambda_{j,k,\text{gesamt}} = \lambda_{j,k} + \sum_i \delta_{i,k} \; q_{ij} \tag{3.21}$$

Blockierendes Nachfolgesystem, eine Anforderungsklasse: Wenn ein System i eine fertig bearbeitete Anforderung nicht weiterleiten kann, weil das Nachfolgesystem j blockiert ist, dann bleibt diese Anforderung im System und verhindert die Bearbeitung aller weiteren Anforderungen. Diese Wechselwirkung wird modelliert, indem die Bedienrate eines Systems je nach Blockierwahrscheinlichkeit ihrer Nachfolgesysteme abgesenkt wird:

$$\mu_i^* = \mu_i \sum_j p_{j,\text{blocking}} \; q_{ij} \tag{3.22}$$

Diese verminderte Bedienrate darf jedoch nicht für die Berechnung der Ausgangsrate verwendet werden, dort muss mit der tatsächlichen Bedienrate gerechnet werden.

Blockierendes Nachfolgesystem, mehrere Anforderungsklassen:

$$\mu_{i,k}^* = \mu_{i,k} \sum_j p_{j,\text{blocking für Klasse } k} \; q_{ij} \tag{3.23}$$

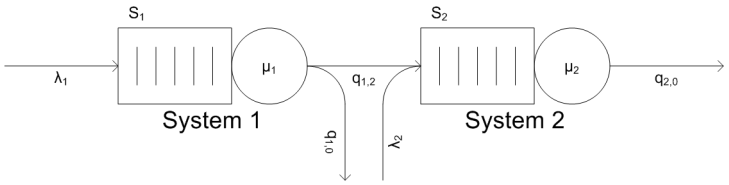
Ergebnisse

Es wurden die in Abbildung 3.27 gezeigten Netze untersucht.

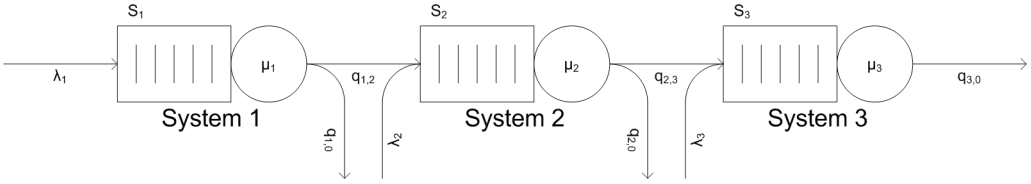
Die Abbildungen 3.28 bis 3.41 zeigen die Auswirkungen eines Überlastimpulses an einem System auf das restliche Netz.

Die Abbildungen 3.42 bis 3.51 zeigen den Zustand des Netzes in Abhängigkeit von der Ankunftsrate von einzelnen Systemen.

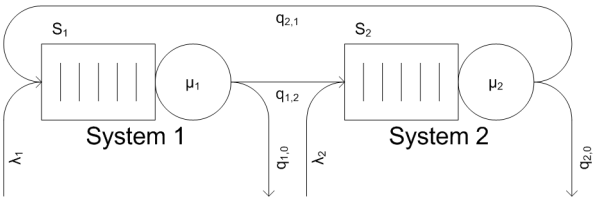
Die durchgezogenen Linien sind die Ergebnisse der exakten Rechnung, die gestrichelten Linien sind die Ergebnisse der Näherung.



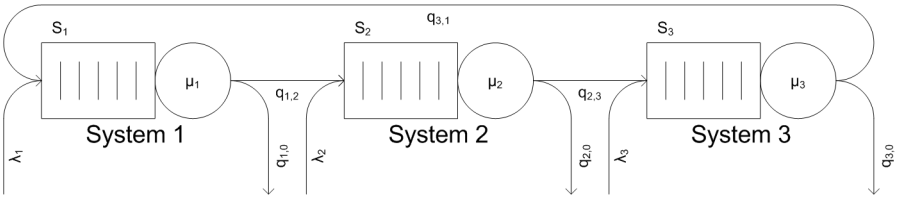
(a) Netz 1. $q_{1,0} = \frac{2}{7},\ q_{1,2} = \frac{5}{7},\ q_{2,0} = 1.$



(b) Netz 2. $q_{1,0} = q_{2,0} = \frac{2}{7},\ q_{1,2} = q_{2,3} = \frac{5}{7},\ q_{3,0} = 1.$



(c) Netz 3. $q_{1,0} = \frac{2}{7},\ q_{1,2} = \frac{5}{7},\ q_{2,0} = \frac{5}{7},\ q_{2,1} = \frac{2}{7}.$



(d) Netz 4. $q_{1,0} = q_{2,0} = \frac{2}{7},\ q_{1,2} = q_{2,3} = \frac{5}{7},\ q_{3,0} = \frac{5}{7},\ q_{3,1} = \frac{2}{7}.$

Abbildung 3.27.: Die untersuchten Netze aus Warteschlangensystemen.

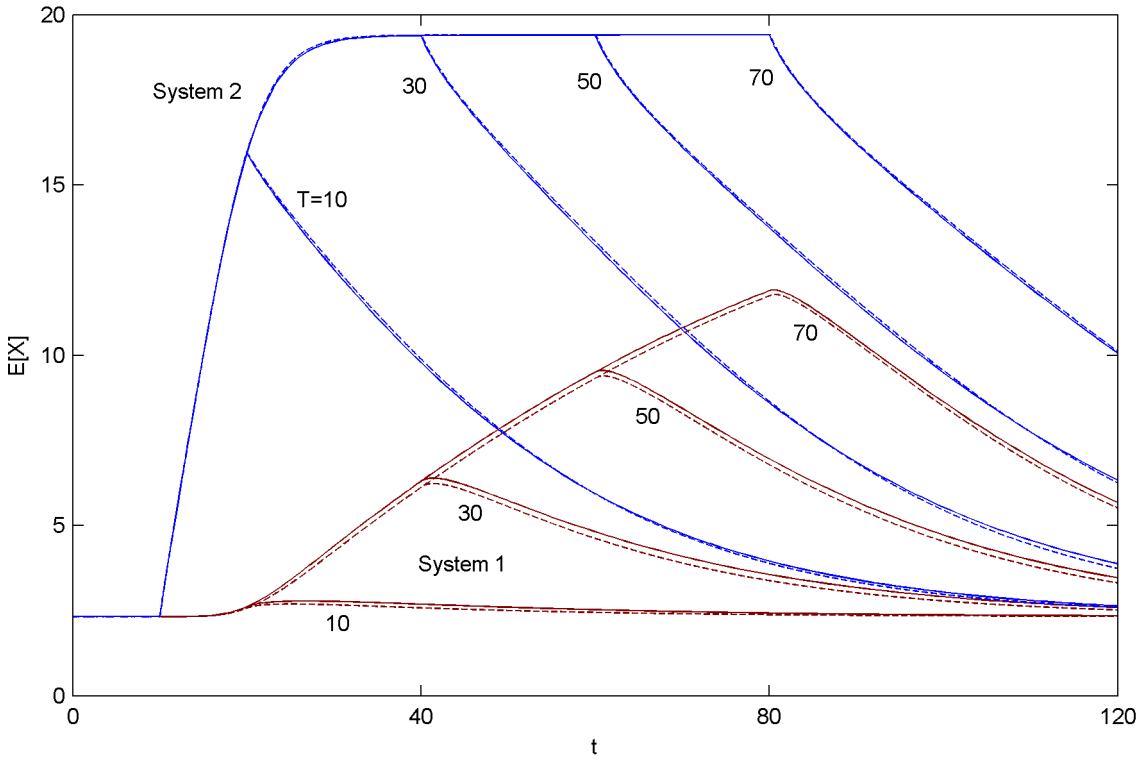


Abbildung 3.28.: Netz 1 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 2. $\mu_1 = \mu_2 = 1,\ \lambda_1 = 0.7,\ \lambda_2 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = 20$.

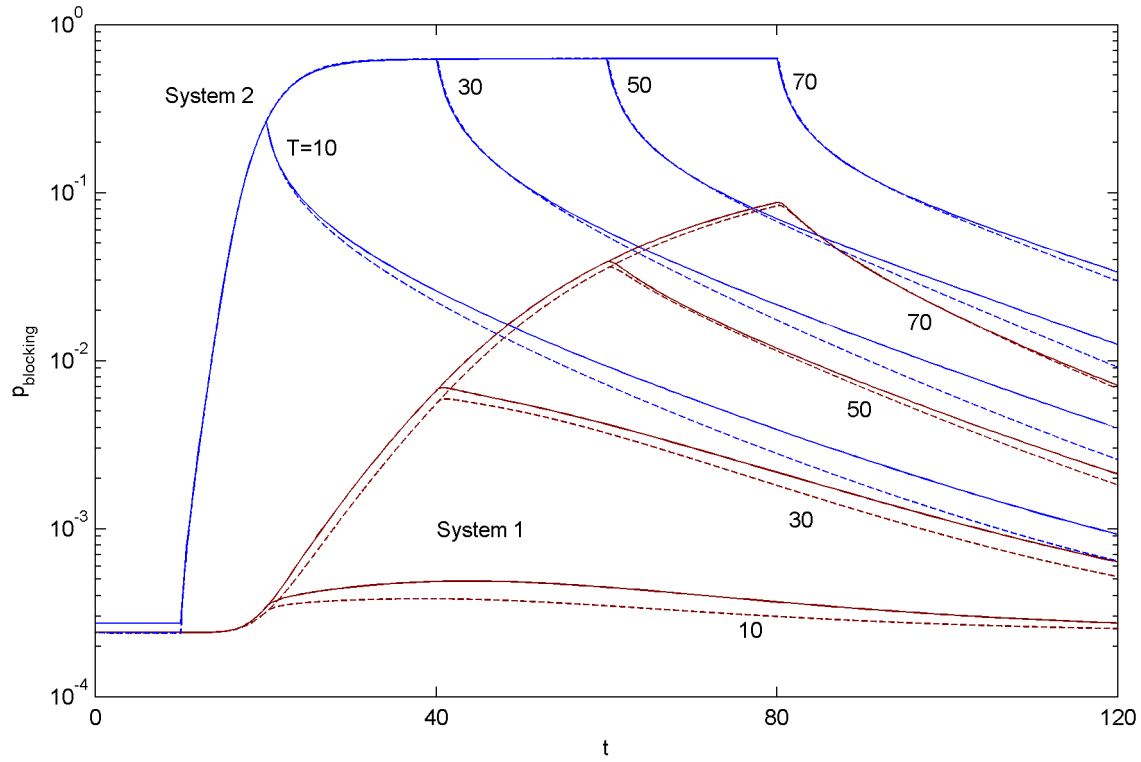


Abbildung 3.29.: Netz 1 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 2. $\mu_1 = \mu_2 = 1,\ \lambda_1 = 0.7,\ \lambda_2 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = 20$.

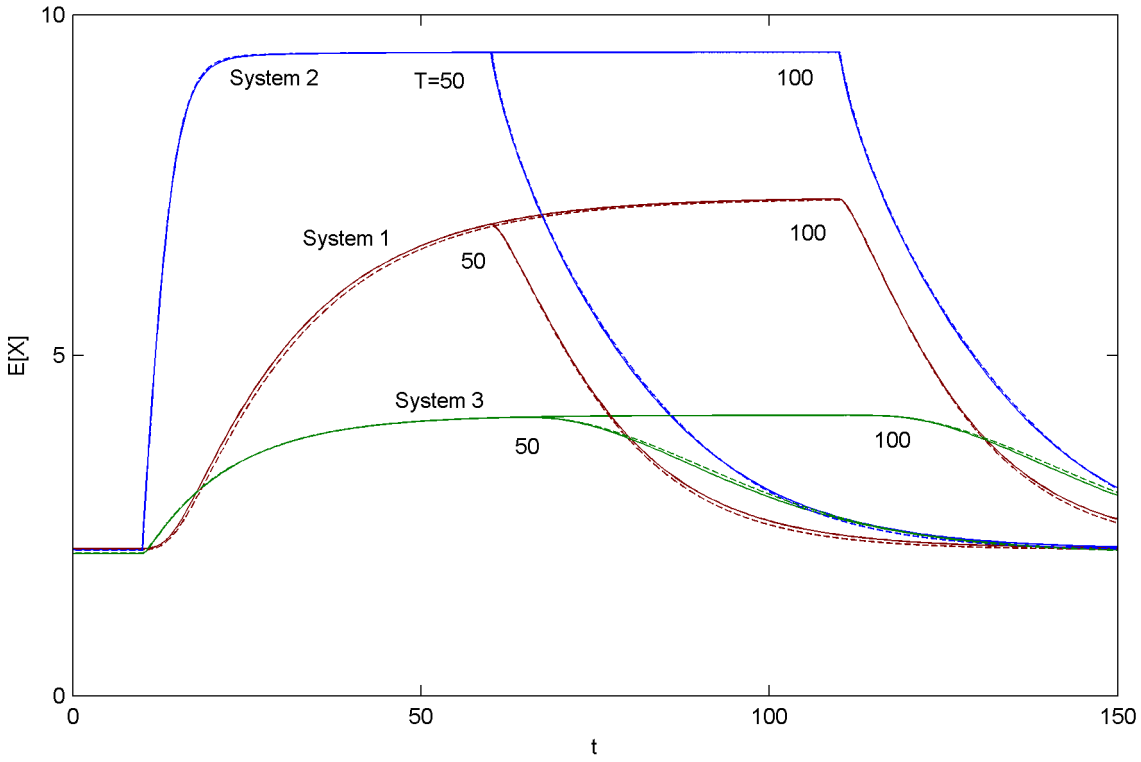


Abbildung 3.30.: Netz 2 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 2. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7,\ \lambda_2 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $\lambda_3 = 0.2,\ S_1 = S_2 = S_3 = 10$.

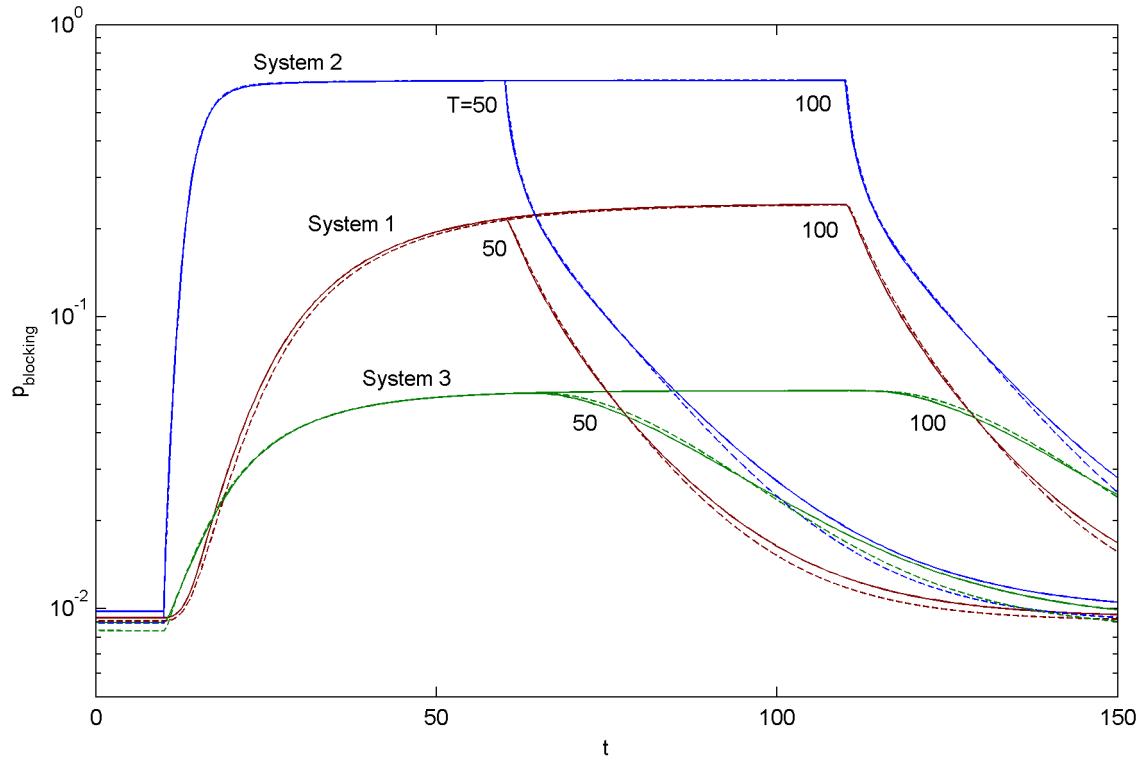


Abbildung 3.31.: Netz 2 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 2. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7,\ \lambda_2 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $\lambda_3 = 0.2,\ S_1 = S_2 = S_3 = 10$.

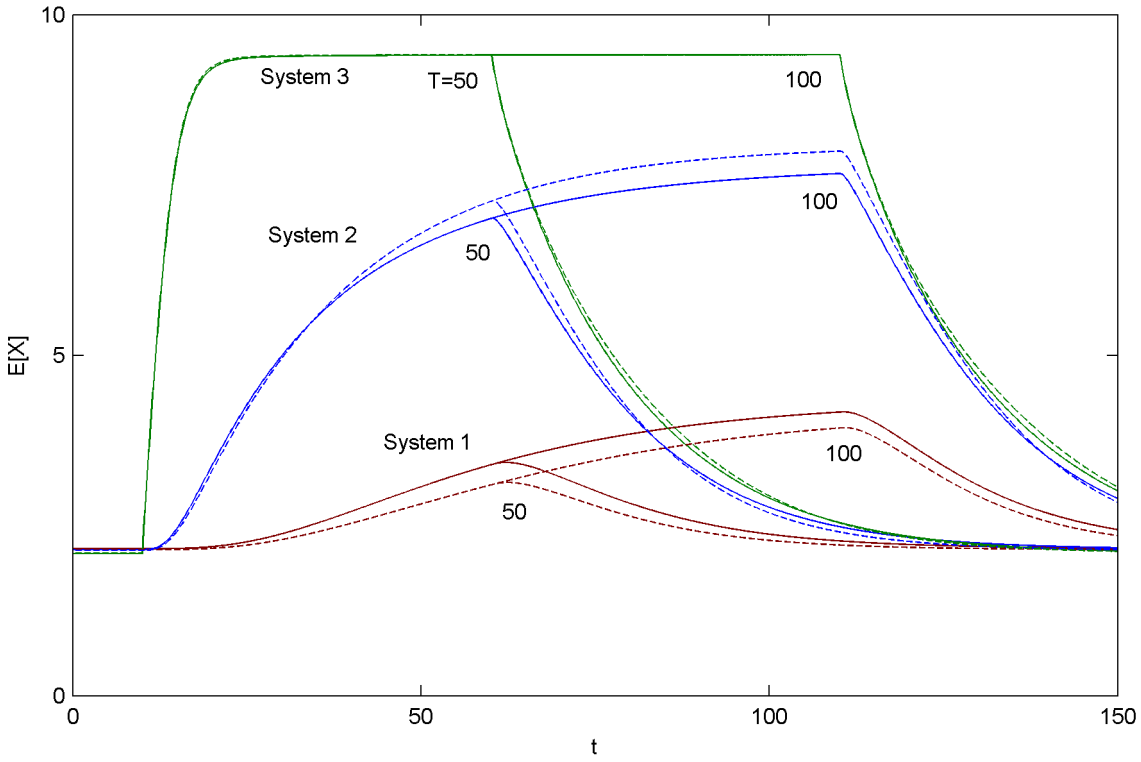


Abbildung 3.32.: Netz 2 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 3. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7, \lambda_2 = 0.2,\ \lambda_3 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = S_3 = 10$.

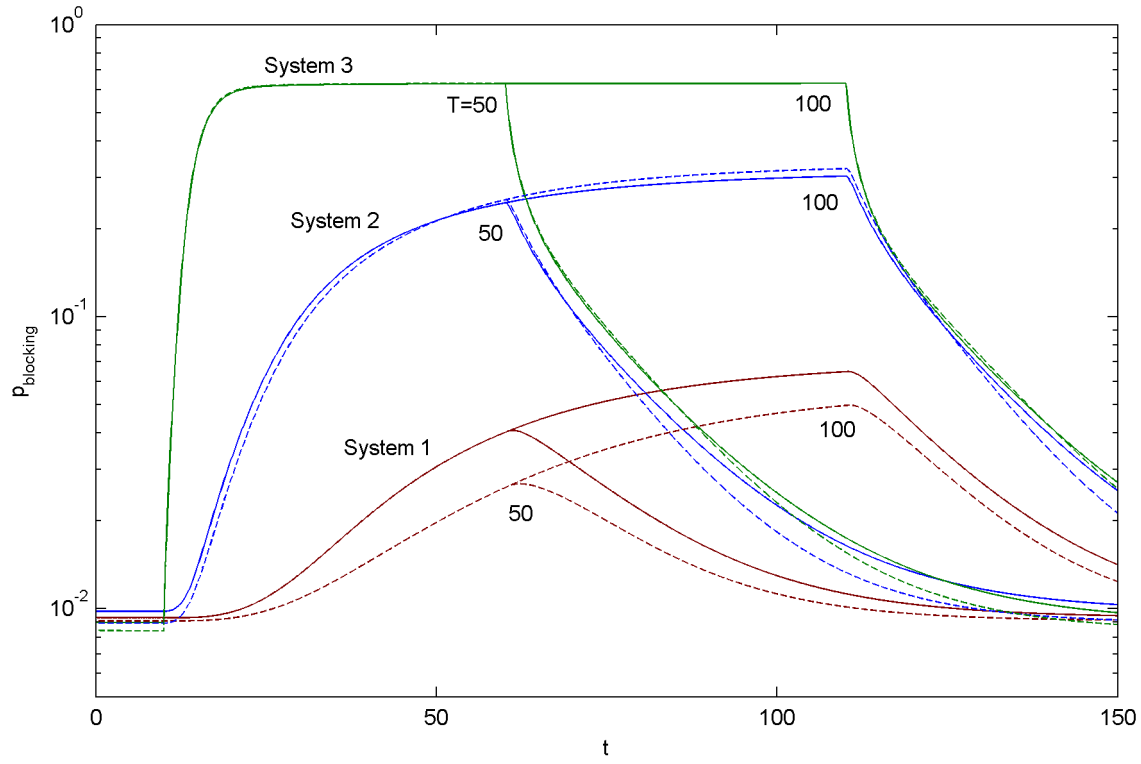


Abbildung 3.33.: Netz 2 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 3. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7, \lambda_2 = 0.2,\ \lambda_3 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = S_3 = 10$.

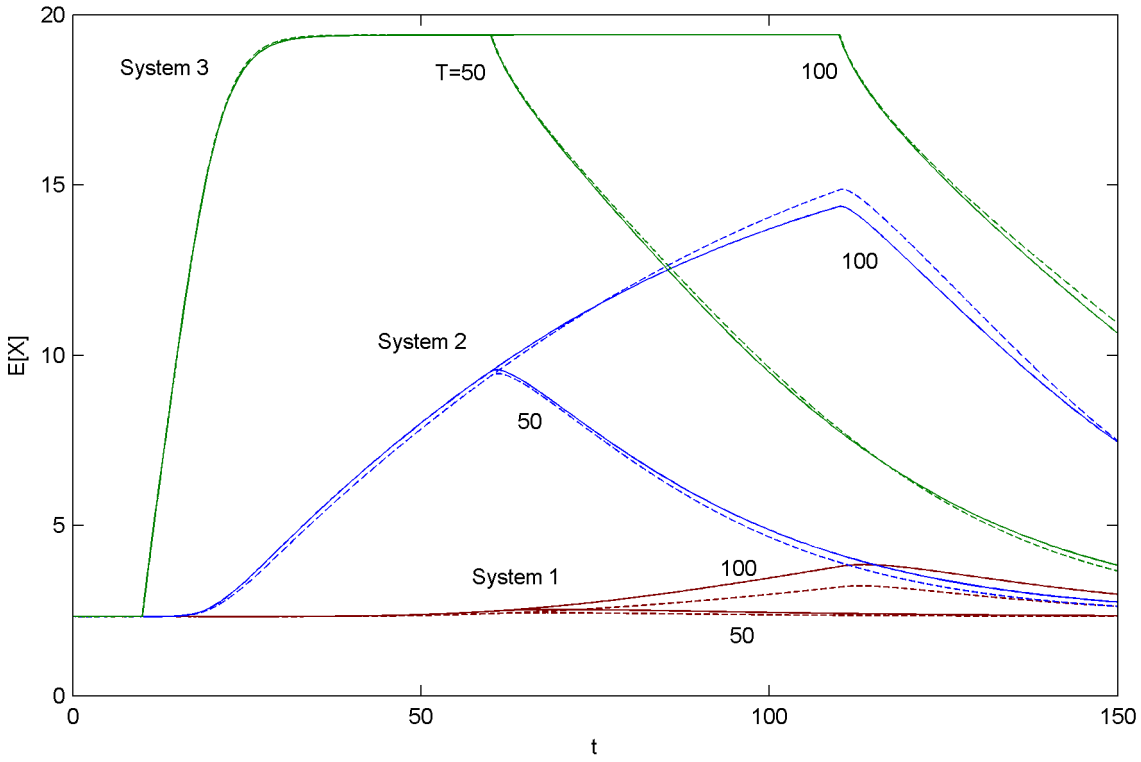


Abbildung 3.34.: Netz 2 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 3. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7, \lambda_2 = 0.2,\ \lambda_3 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = S_3 = 20$.

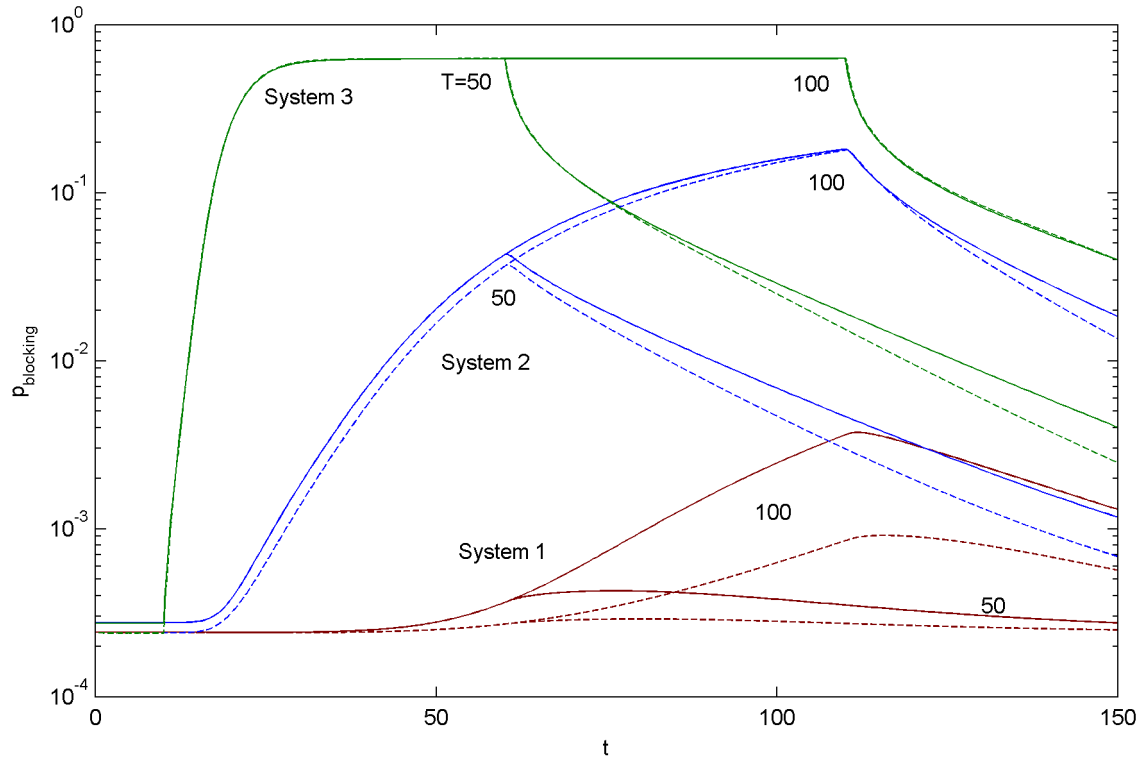


Abbildung 3.35.: Netz 2 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 3. $\mu_1 = \mu_2 = \mu_3 = 1,\ \lambda_1 = 0.7, \lambda_2 = 0.2,\ \lambda_3 = 2$ für $10 < t \leq 10 + T,\ 0.2$ sonst, $S_1 = S_2 = S_3 = 20$.

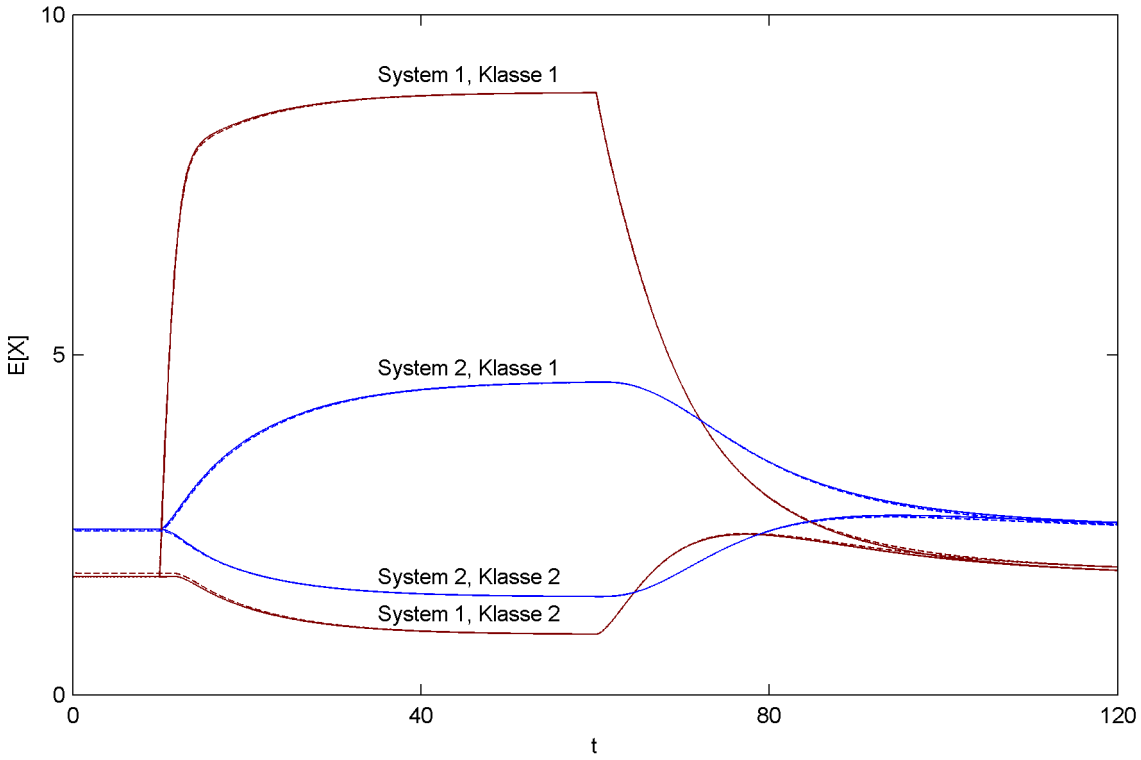


Abbildung 3.36.: Netz 2 mit zwei Anforderungsklassen: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 1. $\mu_{1,1} = \mu_{1,2} = \mu_{2,1} = \mu_{2,2} = 1,\ \lambda_{1,1} = 4$ für $10 < t < 10 + T,\ 0.4$ sonst, $\lambda_{1,2} = 0.4,\ \lambda_{2,1} = \lambda_{2,2} = 0.2,$ Common Buffer, $S_1 = S_2 = 10$. Abfertigungsdisziplin: *Abfertigung in zufälliger Reihenfolge*.

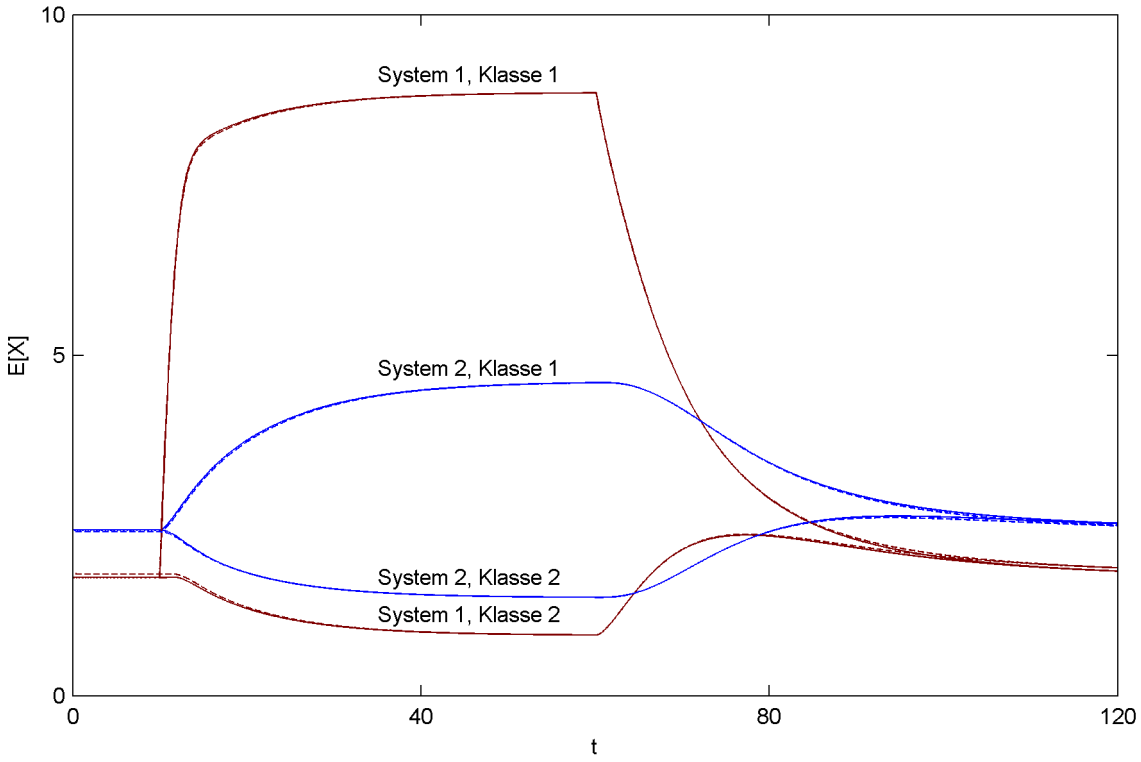


Abbildung 3.37.: Netz 2 mit zwei Anforderungsklassen: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 1. $\mu_{1,1} = \mu_{1,2} = \mu_{2,1} = \mu_{2,2} = 1,\ \lambda_{1,1} = 4$ für $10 < t < 10 + T,\ 0.4$ sonst, $\lambda_{1,2} = 0.4,\ \lambda_{2,1} = \lambda_{2,2} = 0.2,$ Common Buffer, $S_1 = S_2 = 10$. Abfertigungsdisziplin: *Abfertigung in zufälliger Reihenfolge*.

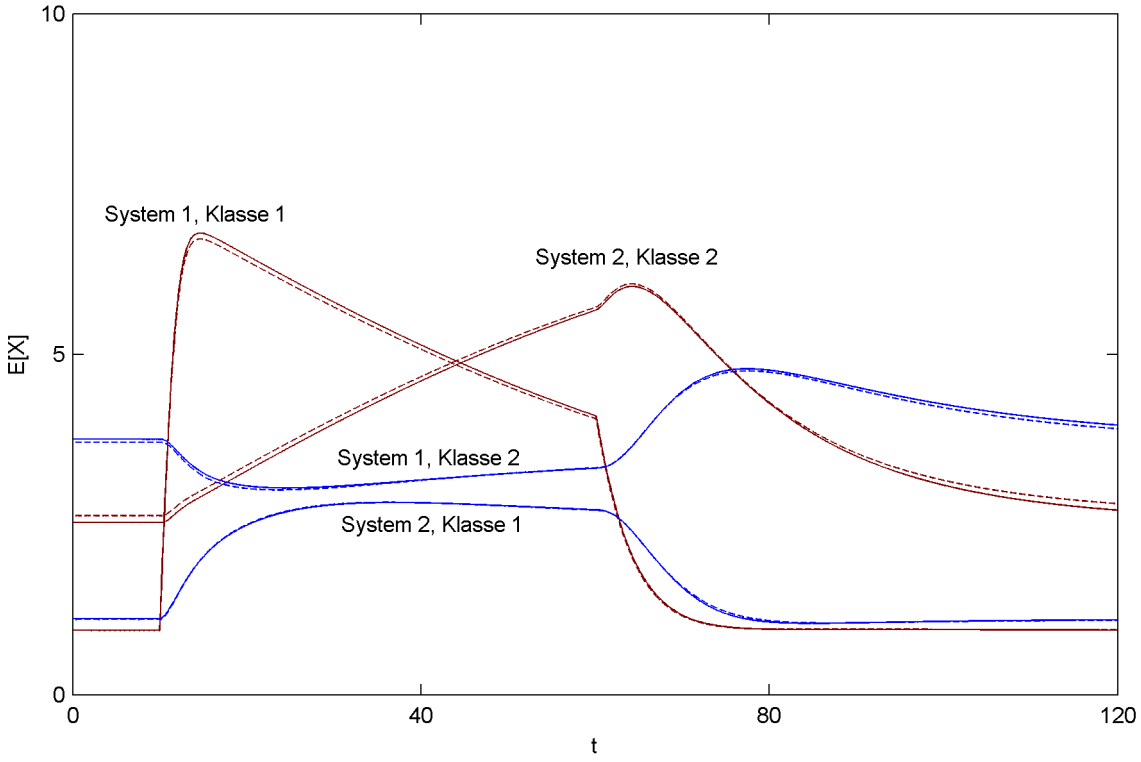


Abbildung 3.38.: Netz 2 mit zwei Anforderungsklassen: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 1. $\mu_{1,1} = \mu_{1,2} = \mu_{2,1} = \mu_{2,2} = 1,\ \lambda_{1,1} = 4$ für $10 < t < 10 + T,\ 0.4$ sonst, $\lambda_{1,2} = 0.4,\ \lambda_{2,1} = \lambda_{2,2} = 0.2,$ Common Buffer, $S_1 = S_2 = 10$. Abfertigungsdisziplin: *Nichtunterbrechende Prioritäten*.

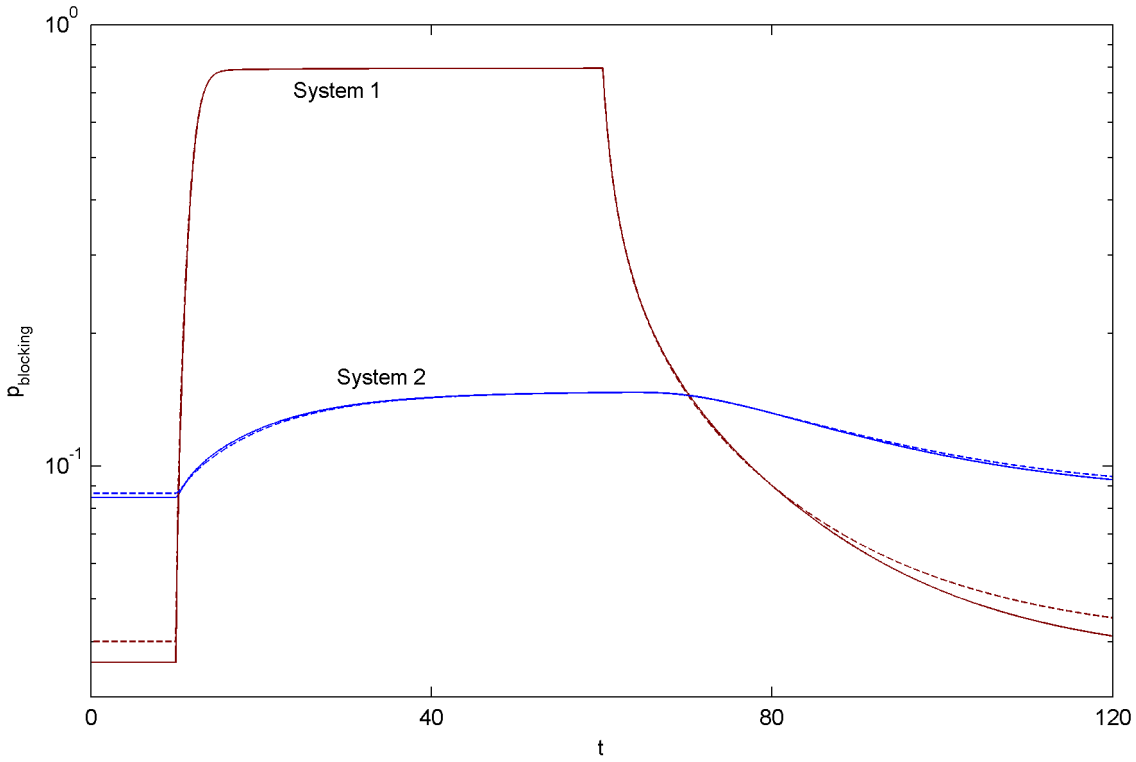


Abbildung 3.39.: Netz 2 mit zwei Anforderungsklassen: Blockierwahrscheinlichkeit der einzelnen Systeme bei einem Überlastimpuls an System 1. $\mu_{1,1} = \mu_{1,2} = \mu_{2,1} = \mu_{2,2} = 1,\ \lambda_{1,1} = 4$ für $10 < t < 10 + T,\ 0.4$ sonst, $\lambda_{1,2} = 0.4,\ \lambda_{2,1} = \lambda_{2,2} = 0.2,$ Common Buffer, $S_1 = S_2 = 10$. Abfertigungsdisziplin: *Nichtunterbrechende Prioritäten*.

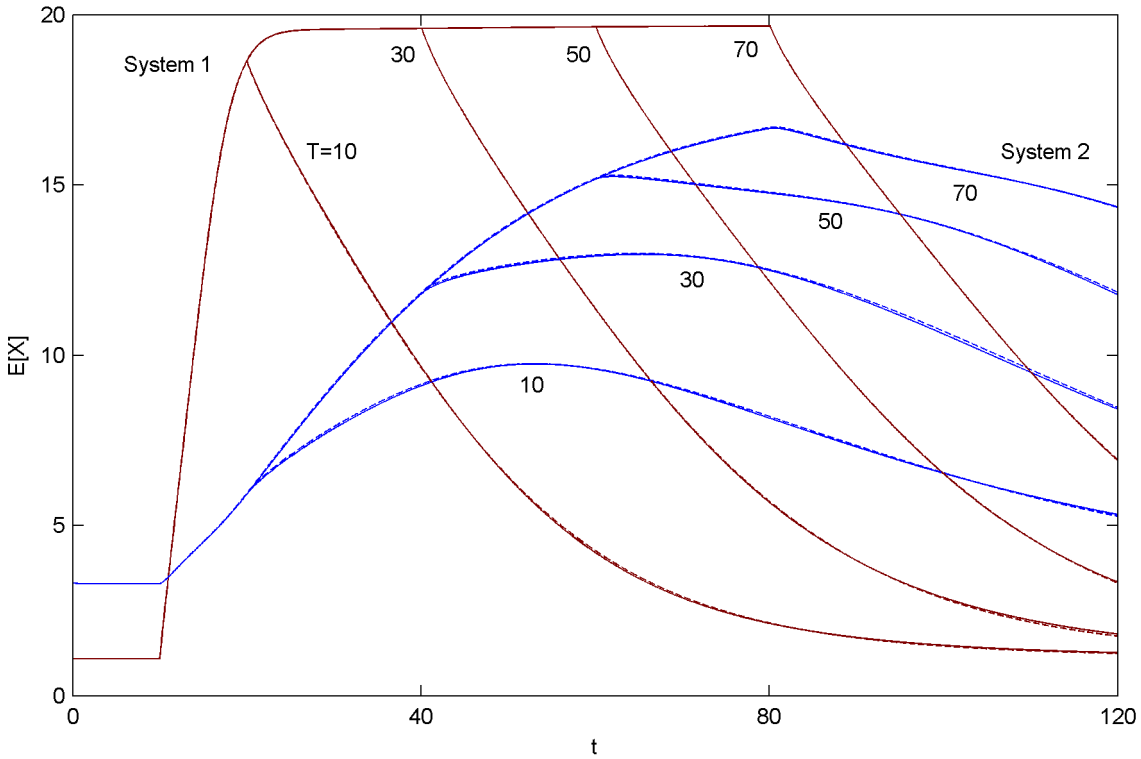


Abbildung 3.40.: Netz 3 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 1. $\mu_1 = \mu_2 = 1,\ \lambda_1 = 3$ für $10 < t < 10 + T,\ 0.3$ sonst, $\lambda_2 = 0.4,\ S_1 = S_2 = 20$.

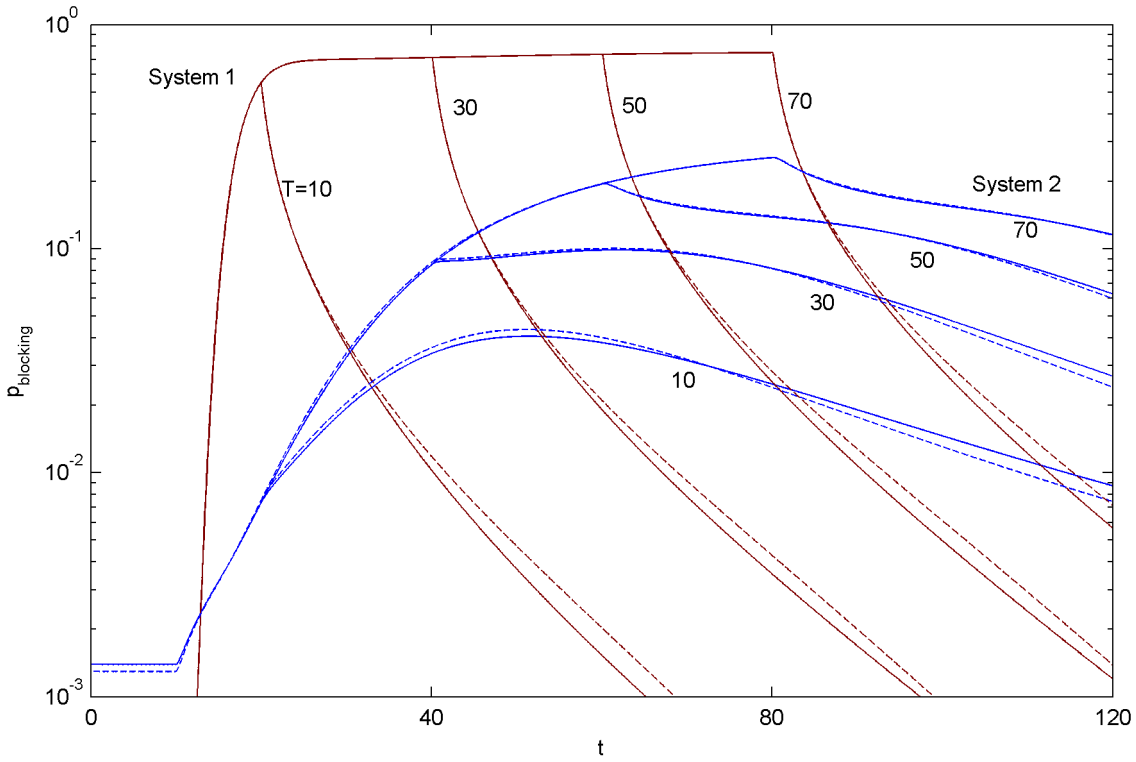


Abbildung 3.41.: Netz 3 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen bei einem Überlastimpuls an System 1. $\mu_1 = \mu_2 = 1,\ \lambda_1 = 3$ für $10 < t < 10 + T,\ 0.3$ sonst, $\lambda_2 = 0.4,\ S_1 = S_2 = 20$.

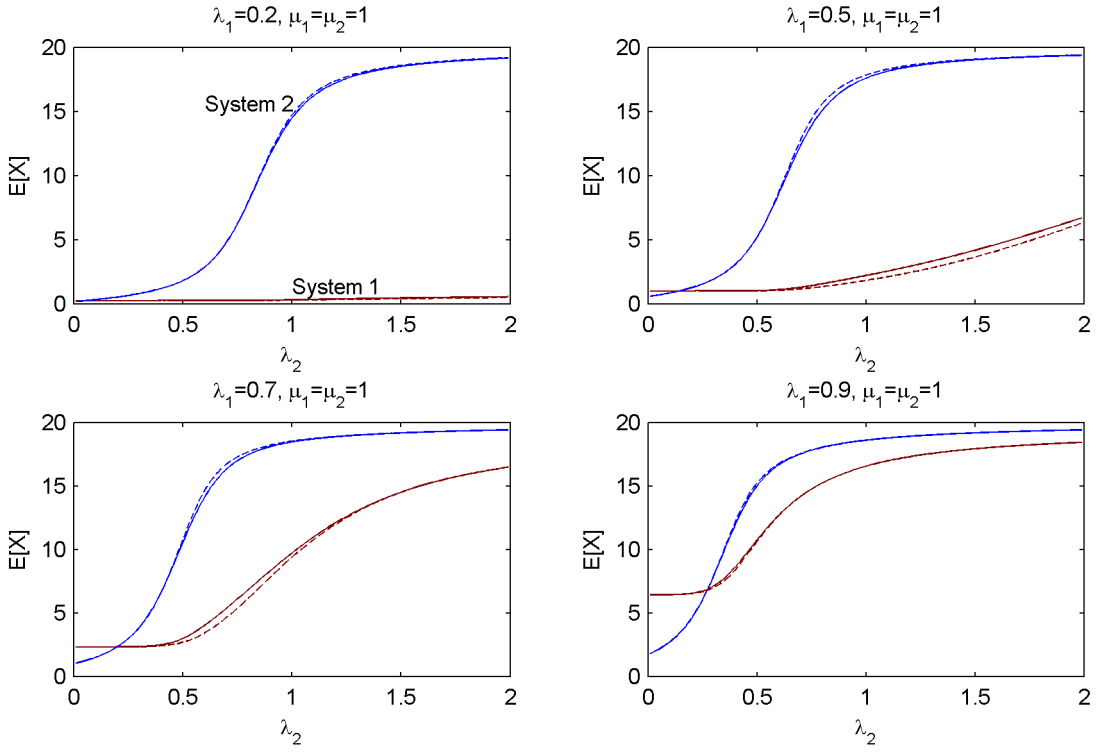


Abbildung 3.42.: Netz 1 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen in Abhängigkeit von der Ankunftsrate λ_2. $S_1 = S_2 = 20$.

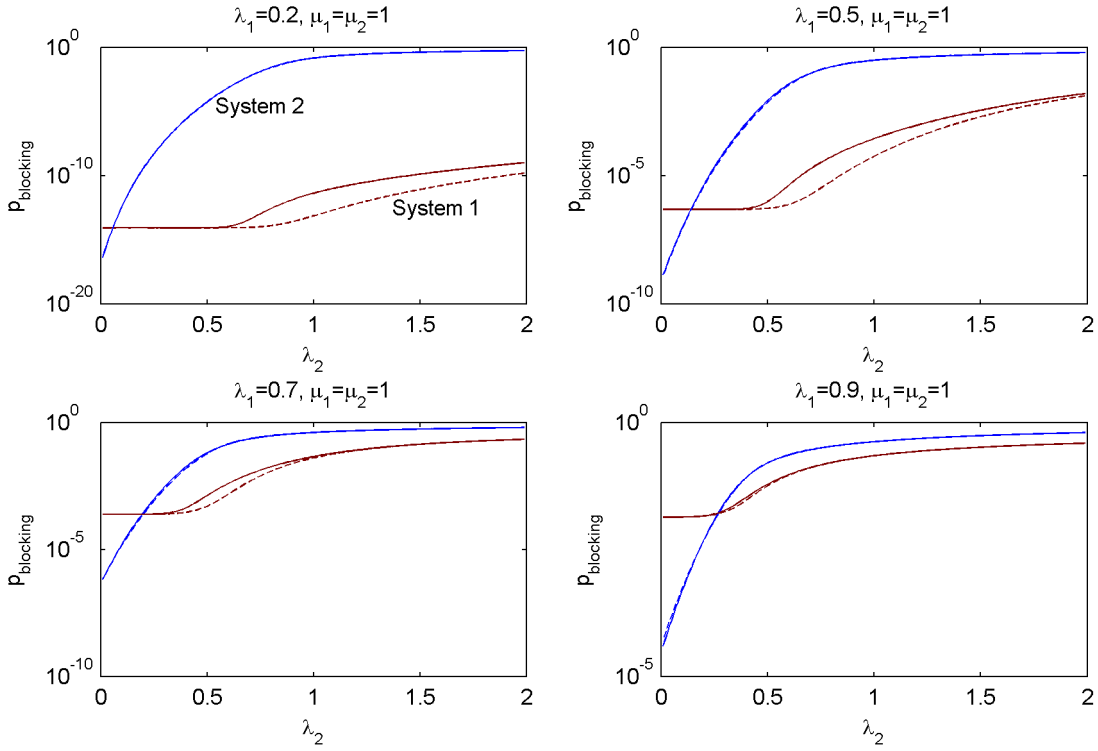


Abbildung 3.43.: Netz 1 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit in den einzelnen Systemen in Abhängigkeit von der Ankunftsrate λ_2. $S_1 = S_2 = 20$.

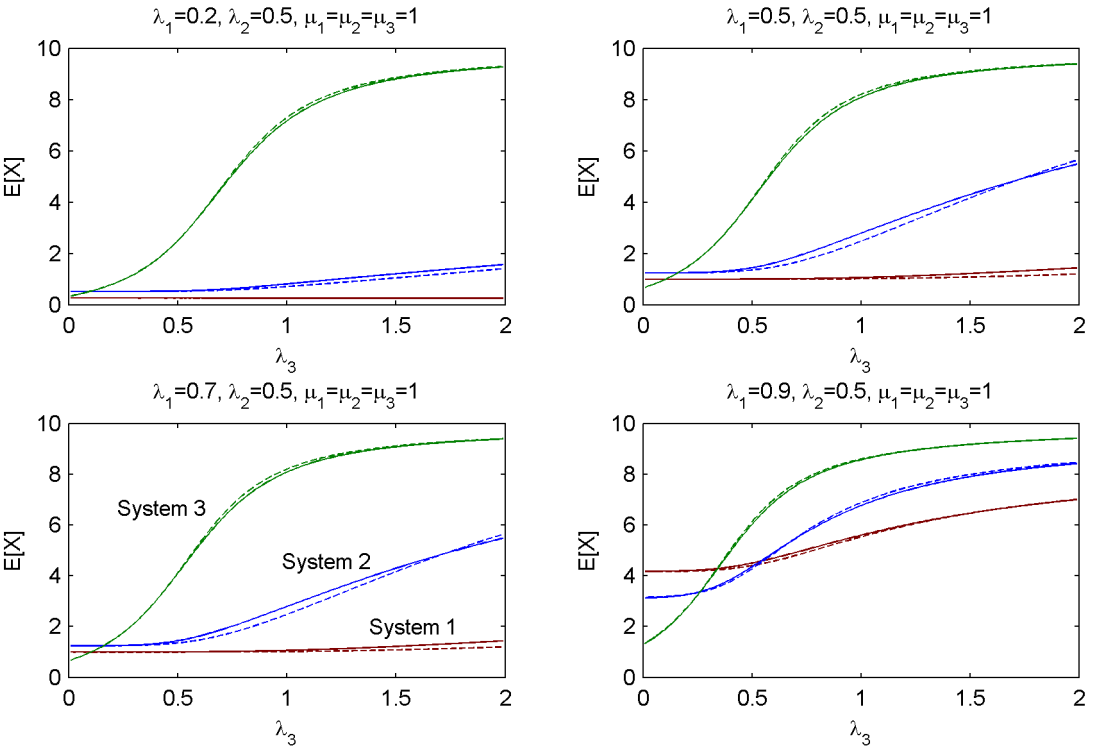


Abbildung 3.44.: Netz 2 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 10$.

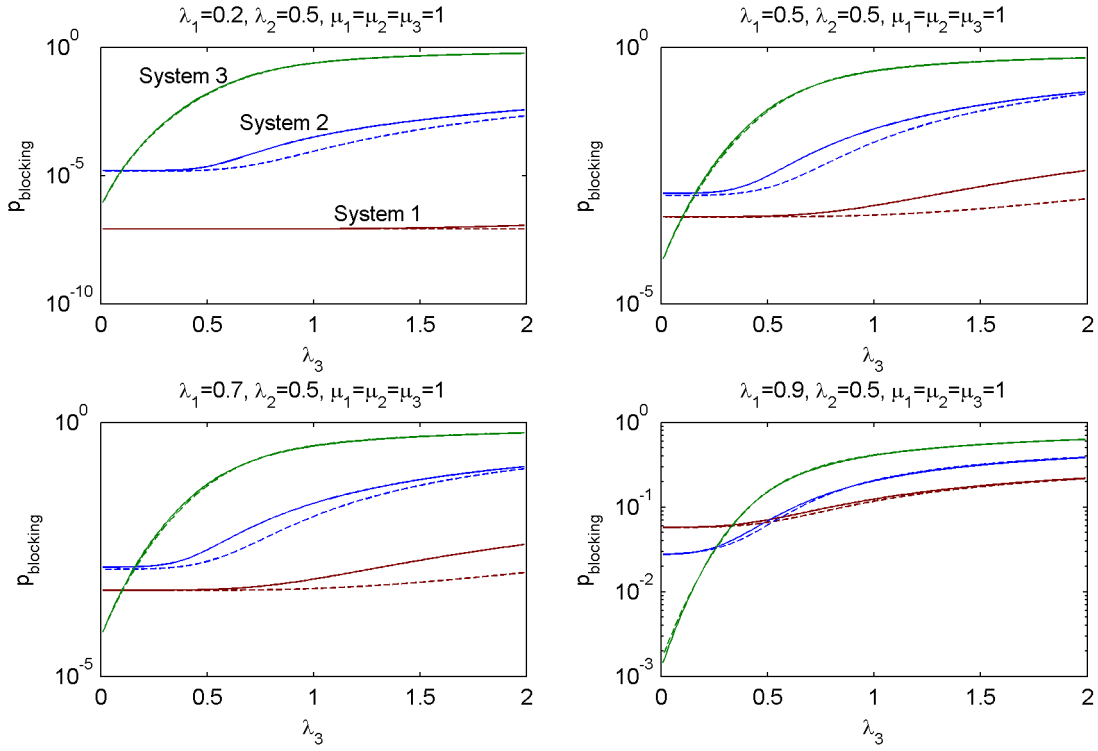


Abbildung 3.45.: Netz 2 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 10$.

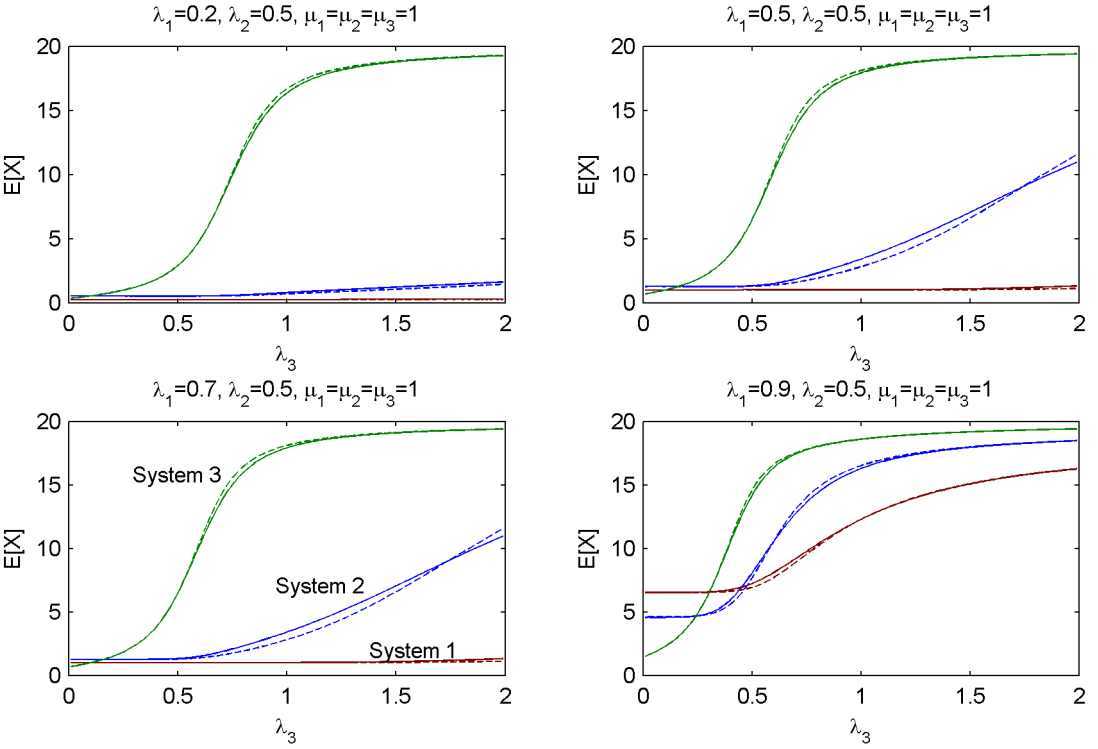


Abbildung 3.46.: Netz 2 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 20$.

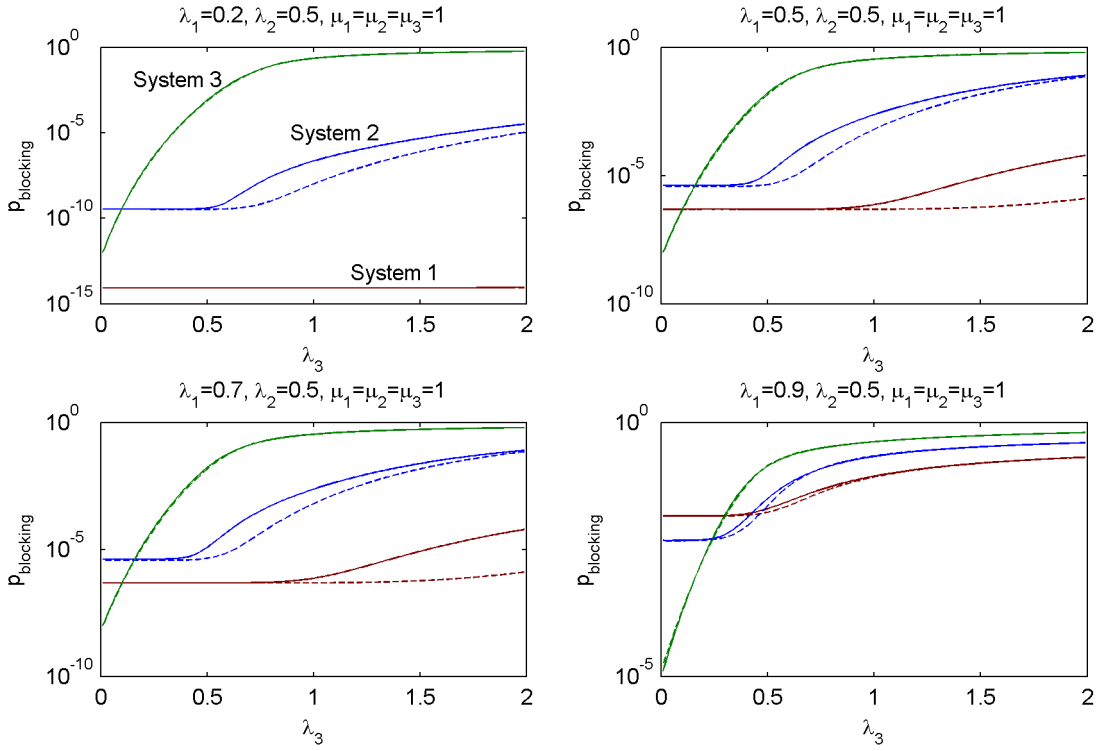


Abbildung 3.47.: Netz 2 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 20$.

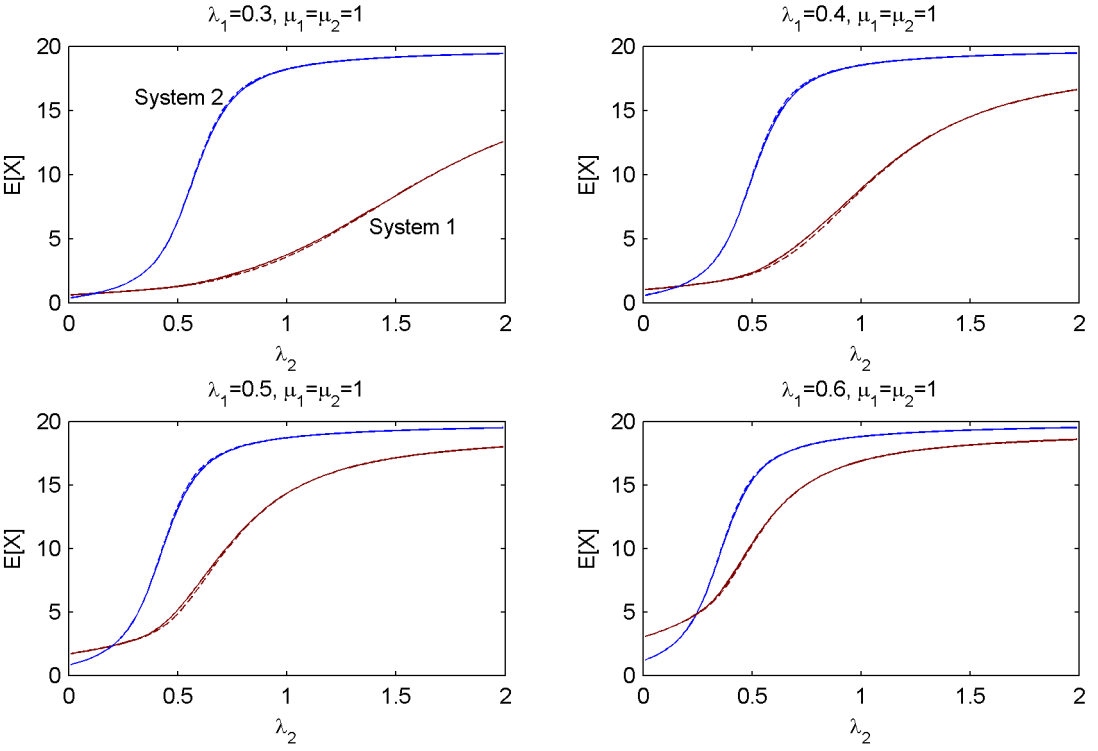


Abbildung 3.48.: Netz 3 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen (stationär) in Abhängigkeit von der Ankunftsrate λ_2. $S_1 = S_2 = 20$.

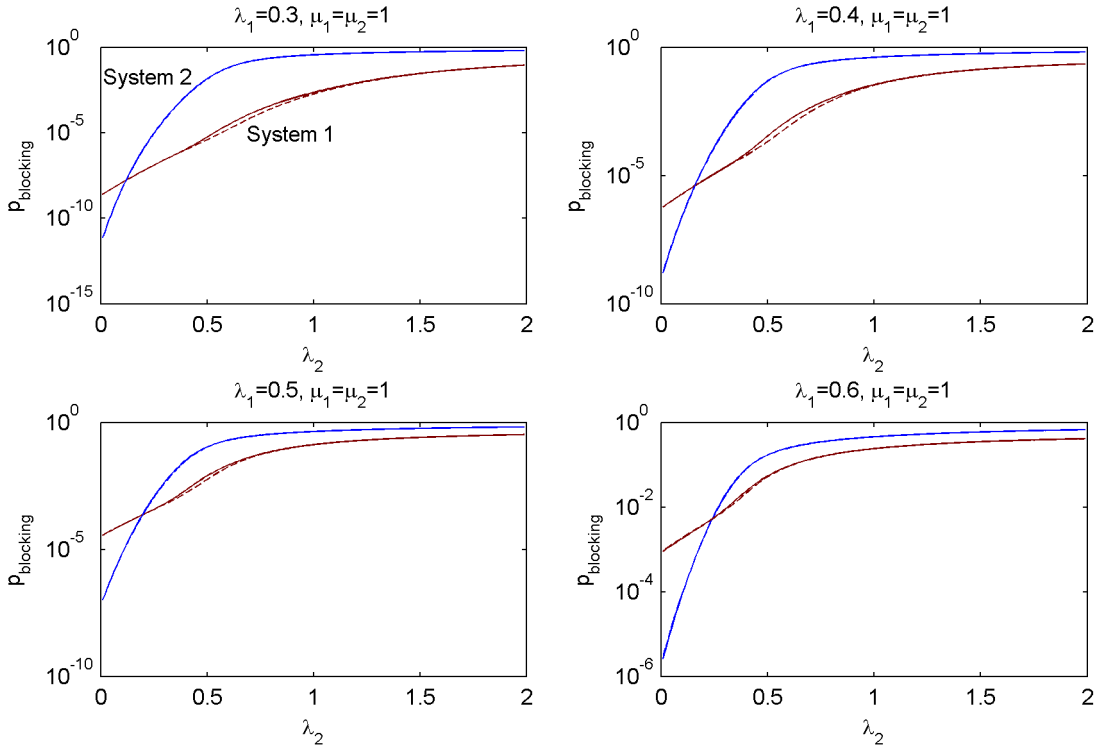


Abbildung 3.49.: Netz 3 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme (stationär) in Abhängigkeit von der Ankunftsrate λ_2. $S_1 = S_2 = 20$.

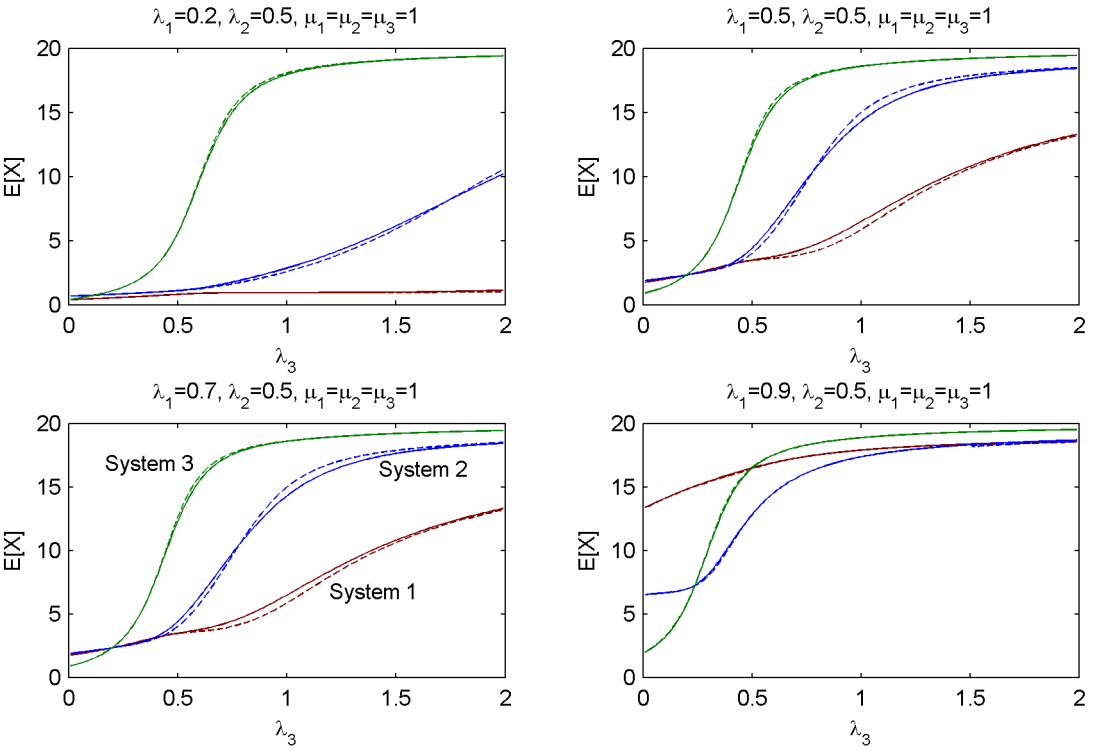


Abbildung 3.50.: Netz 4 mit einer Anforderungsklasse: Anzahl der Anforderungen in den einzelnen Systemen (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 20$.

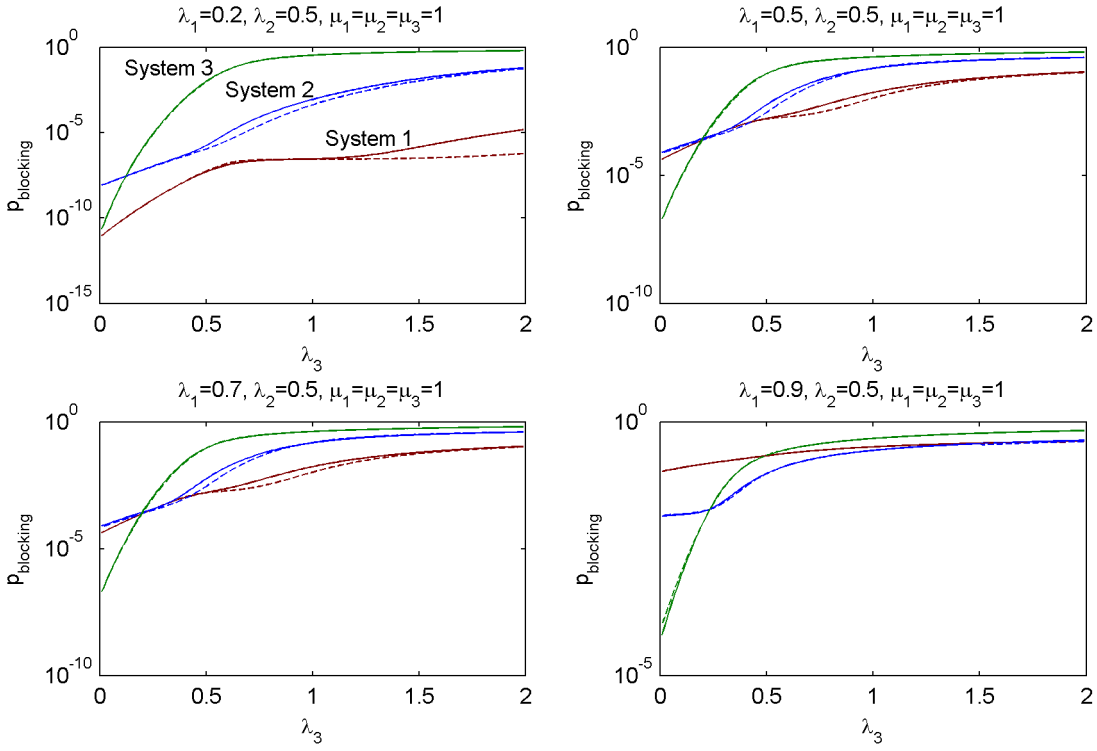


Abbildung 3.51.: Netz 4 mit einer Anforderungsklasse: Blockierwahrscheinlichkeit der einzelnen Systeme (stationär) in Abhängigkeit von der Ankunftsrate λ_3. $S_1 = S_2 = S_3 = 20$.

3.4. Modellierung des Durchflussprozesses

Die Durchflusszeit durch ein Warteschlangensystem kann nicht mehr nur anhand des Systemzustands berechnet werden. Stattdessen wird das Schicksal einer sogenannten Testanforderung betrachtet. Die Testanforderung tritt zu einem beliebigen Zeitpunkt in das Warteschlangensystem ein und beginnt ihren Durchflussprozess. Der Durchflussprozess endet, wenn die Testanforderung das System wieder verlassen hat. Die Länge des Intervalls zwischen Eintritt und Austritt der Testanforderung ist die Durchflusszeit. Es werden nur solche Testanforderungen betrachtet, die nach dem Eintreffen vom Warteschlangensystem akzeptiert und nicht (beispielsweise weil das System voll ist) abgewiesen werden.

In den meisten Fällen hängt das Schicksal der Testanforderung sowohl vom Systemzustand als auch von ihrer Entwicklung innerhalb der aktuellen Abfertigungsdisziplin des Systems (im folgenden Abfertigungsprozess genannt) ab.

In der erzeugten Markov-Kette können drei Arten von Zuständen unterschieden werden:

- Zustände, bei denen der Durchflussprozess beginnen kann. Diese Zustände können unmittelbar nach dem Eintritt der Testanforderung eingenommen werden. Die Menge dieser Zustände wird mit $\mathcal{M}_1$ bezeichnet.
- Zustände, bei denen der Durchflussprozess nicht beginnen kann. Diese Zustände können nur indirekt erreicht werden. Die Menge dieser Zustände wird mit $\mathcal{M}_2$ bezeichnet.
- Zustände, an denen die Testanforderung das System wieder verlassen hat. Die Menge dieser Zustände wird mit $\mathcal{H}$ bezeichnet. Diese Zustände sind absorbierend, denn mit dem Erreichen eines dieser Zustände ist der Durchflussprozess beendet.

Die Durchflusszeit entspricht der Zeit, die die Markov-Kette, beginnend bei einem Beginnzustand aus der Menge $\mathcal{M}_1$, benötigt, um einen absorbierenden Zustand zu erreichen.

Die Wahrscheinlichkeitsverteilung der Durchflusszeit T_F wird mit Hilfe der komplementären Verteilungsfunktion

$$F(\tau) = W\{T_F > \tau\} \tag{3.24}$$

angegeben.

Je nachdem, welchen Systemzustand die Testanforderung bei ihrer Ankunft vorfindet (das heißt bei welchem Beginnzustand der Durchflussprozess startet), wird die Durchflusszeit länger oder kürzer sein. Die gesamte Durchflusszeit ergibt sich aus der gewichteten Summe der Durchflusszeiten $f_i(\tau)$ für die einzelnen Beginnzustände i, wobei zur Gewichtung die (zum Zeitpunkt des Eintreffens der Testanforderung geltenden) Zustandswahrscheinlichkeiten für die Systemzustände, die zu den entsprechenden Beginnzuständen führen, verwendet werden.

$$f_i(\tau) = W\{T_F > \tau \mid \text{Beginnzustand } i\} \tag{3.25}$$

$$F(\tau) = \sum_{i \in \mathcal{M}_1} f_i \cdot W\{\text{Beginnzustand } i\} \tag{3.26}$$

Die Wahrscheinlichkeiten $f_i(\tau)$ können mit Hilfe der Gleichung (2.51) berechnet werden.

Die mittlere Durchflusszeit ergibt sich (durch partielle Integration der Formel für den Erwartungswert) zu

$$\mathrm{E}[T_F] = \int_{\tau=0}^{\infty} F(\tau)\,\mathrm{d}\tau \tag{3.27}$$

3.4.1. M/M/1/S-Warteschlangensystem

Zunächst wird ein einfaches M/M/1/S-Warteschlangensystem mit Ankunftsrate λ, Bedienrate μ und Abfertigung in der Reihenfolge des Eintreffens *(FIFO, First In – First Out)* betrachtet.

Bei diesem System hat der Systemzustand (außer für die Bestimmung der Wahrscheinlichkeiten der einzelnen Beginnzustände) keinen Einfluss auf den Durchflussprozess. Die Entwicklung innerhalb der aktuellen Abfertigungsdisziplin, und somit auch der Durchflussprozess, ist festgelegt durch die Anzahl der Anforderungen, die beim Eintreffen der Testanforderung im System sind und daher noch vor der Testanforderung bedient werden.

Modell

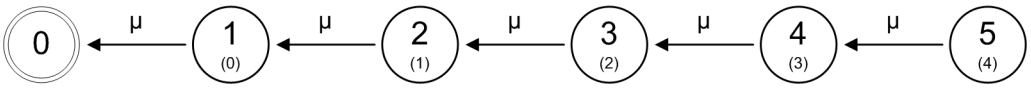


Abbildung 3.52.: M/M/1/S-Warteschlangensystem: Durchflussprozess (=Abfertigungsprozess) für $S = 4$. In Klammern ist angegeben, welcher beim Eintreffen der Testanforderung vorgefundene Systemzustand (siehe Abbildung 3.1) zum entsprechenden Zustand im Durchflussprozess führt.

Die Markov-Kette für den Durchflussprozess ist in Abbildung 3.52 gezeigt. Der absorbierende Zustand 0 bedeutet, dass die Testanforderung das System verlassen hat. Die anderen Zustände sind mögliche Beginnzustände. Im Zustand 1 wird beispielsweise begonnen, wenn die Testanforderung beim Eintreffen ein leeres System vorfindet. Sie wird dann sofort bedient, und mit der Bedienrate μ erreicht der Durchflussprozess den Zustand 0. Beim Zustand 2 wird begonnen, wenn die Testanforderung beim Eintreffen ein System vorfindet, in dem sich eine Anforderung befindet. Diese Anforderung verläßt mit der Bedienrate μ das System (Zustand 1), woraufhin die Testanforderung selbst bedient wird. Ihre Bedienung ist mit der Bedienrate μ abgeschlossen, anschließend verläßt sie

das System (Zustand 0). Die Wahrscheinlichkeiten für die verschiedenen Beginnzustände hängen vom Systemzustand $\pi(t)$ zum Zeitpunkt t des Eintreffens der Testanforderung ab:

$$W\{\text{Beginnzustand } 1\} = \pi_0(t)$$
$$W\{\text{Beginnzustand } 2\} = \pi_1(t)$$
$$\ldots$$

Für die gesamte Durchflusszeit gilt:

$$F(\tau) = f_1(\tau)\ \pi_0(t) + f_2(\tau)\ \pi_1(t) + f_3(\tau)\ \pi_2(t) + f_4(\tau)\ \pi_3(t) + f_5(\tau)\ \pi_4(t) \tag{3.28}$$

3.4.2. M/M/1/S-Warteschlangensystem mit geregelter Bedienrate

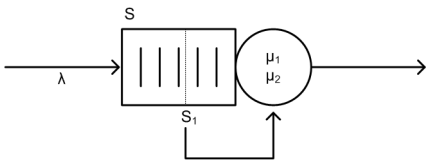


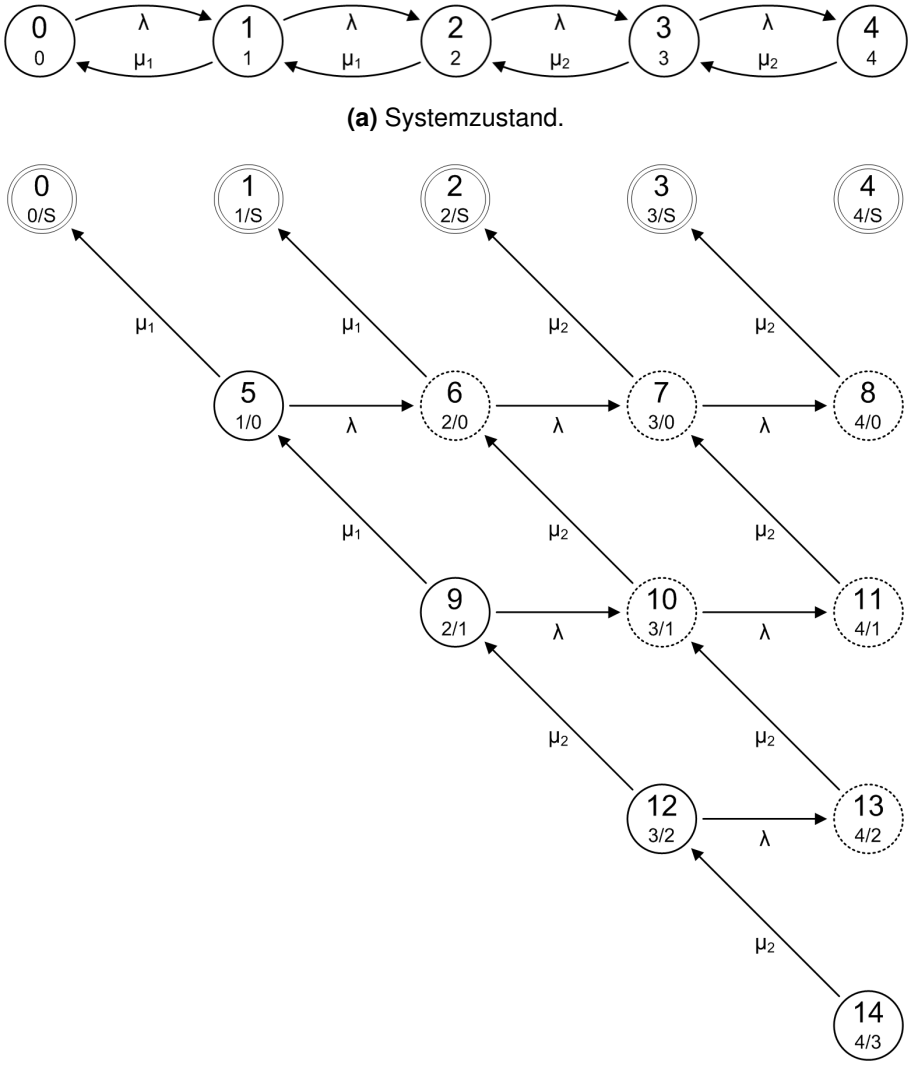
Abbildung 3.53.: M/M/1/S-Warteschlangensystem mit geregelter Bedienrate.

Bei diesem M/M/1/S-Warteschlangensystem (Abbildung 3.53) wird die Bedienrate in Abhängigkeit von der Anzahl der Anforderungen im System geregelt: Die normale Bedienrate ist μ_1. Wird die Länge der Warteschlange größer als 2, wird auf die erhöhte Bedienrate μ_2 umgeschaltet, um die Wahrscheinlichkeit für das Vollwerden des Systems zu verringern. Das Zurückschalten erfolgt, sobald die Länge der Warteschlange wieder kleiner oder gleich 2 ist. Die Ankunftsrate ist konstant λ und die Abfertigung erfolgt wieder in der Reihenfolge des Eintreffens.

Hier wird das Schicksal der Testanforderung auch von nachfolgenden Anforderungen beeinflußt. Kommen beispielsweise nach der Testanforderung so viele Anforderungen nach, dass ständig mehr als zwei Anforderungen im System sind, dann werden die Testanforderung und alle vor ihr bedienten Anforderungen mit der erhöhten Bedienrate bedient. Kommen andererseits nach der Testanforderung keine Anforderungen mehr an, wird die Testanforderung und die vor ihr bediente Anforderung nur mit der normalen Bedienrate bedient; die Durchflusszeit ist somit höher.

Modell

Die Markov-Kette für den Durchflussprozess ist in Abbildung 3.54c zu sehen. Sie kann aus den Markov-Ketten für den Systemzustand (Abbildung 3.54a) und den Abfertigungsprozess (Abbildung 3.54b) konstruiert werden.



(a) Systemzustand.

(b) Abfertigungsprozess.

(c) Durchflussprozess. Beschriftung der Zustände: Anzahl von Anforderungen im System / Anzahl der Anforderungen, die noch vor der Testanforderung bedient werden oder „S“, wenn die Testanforderung bedient wurde. Nur indirekt erreichbare Zustände (Menge $\mathcal{M}_2$) sind strichliert gezeichnet, absorbierende Zustände (Menge $\mathcal{H}$) mit doppelten Linien.

Abbildung 3.54.: M/M/1/S-Warteschlangensystem mit geregelter Bedienrate: Markov-Ketten für Systemzustand, Abfertigungsprozess und Durchflussprozess ($S = 4$).

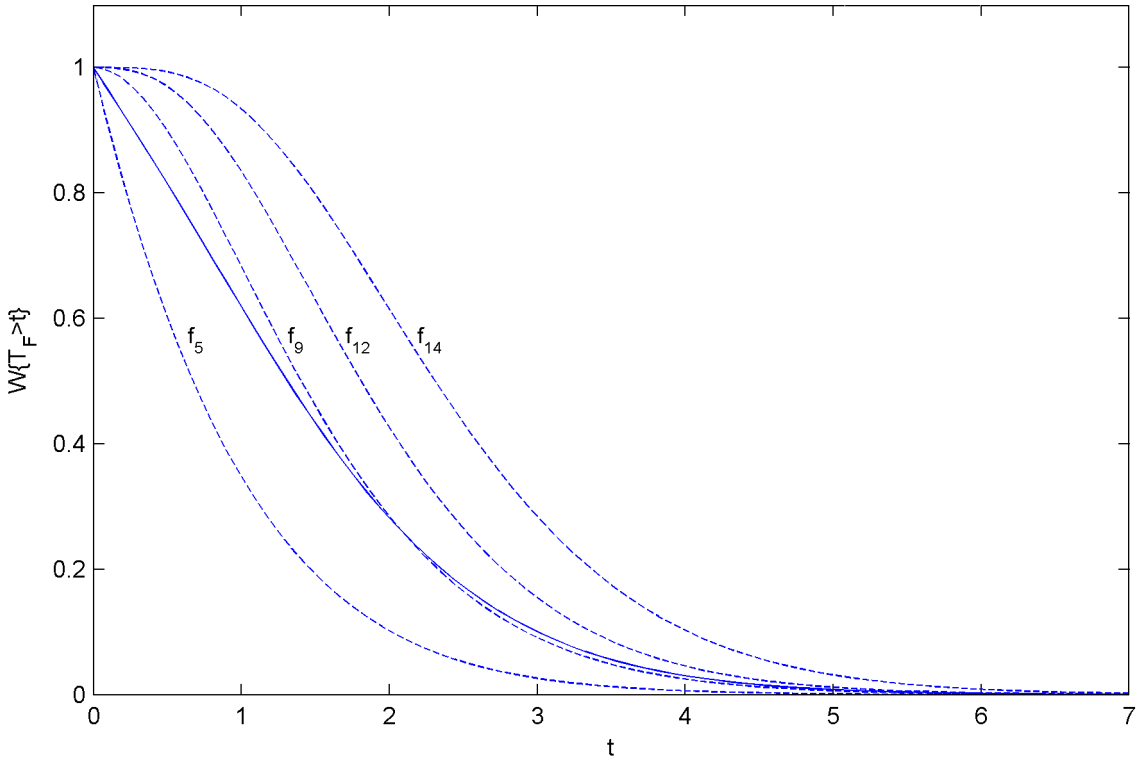


Abbildung 3.55.: M/M/1/S-Warteschlangensystem mit geregelter Bedienrate: Durchflusszeiten für die einzelnen Beginnzustände (gestrichelte Linien), gesamte Durchflusszeit (durchgezogene Linie). $\lambda = 0.8,\ \mu_1 = 1,\ \mu_2 = 2$.

Ergebnisse

Die Durchflusszeiten für die verschiedenen Beginnzustände und die gesamte Durchflusszeit sind in Abbildung 3.55 zu sehen.

Die Auswirkung von nachfolgenden Anforderungen auf die Durchflusszeit der Testanforderung ist in Abbildung 3.56 zu sehen.

Das für die Untersuchung dieses Warteschlangensystems verwendete MATLAB-Programm *M_M_1_S_OnePointS.m* ist im Anhang B zu finden (Listing B.3).

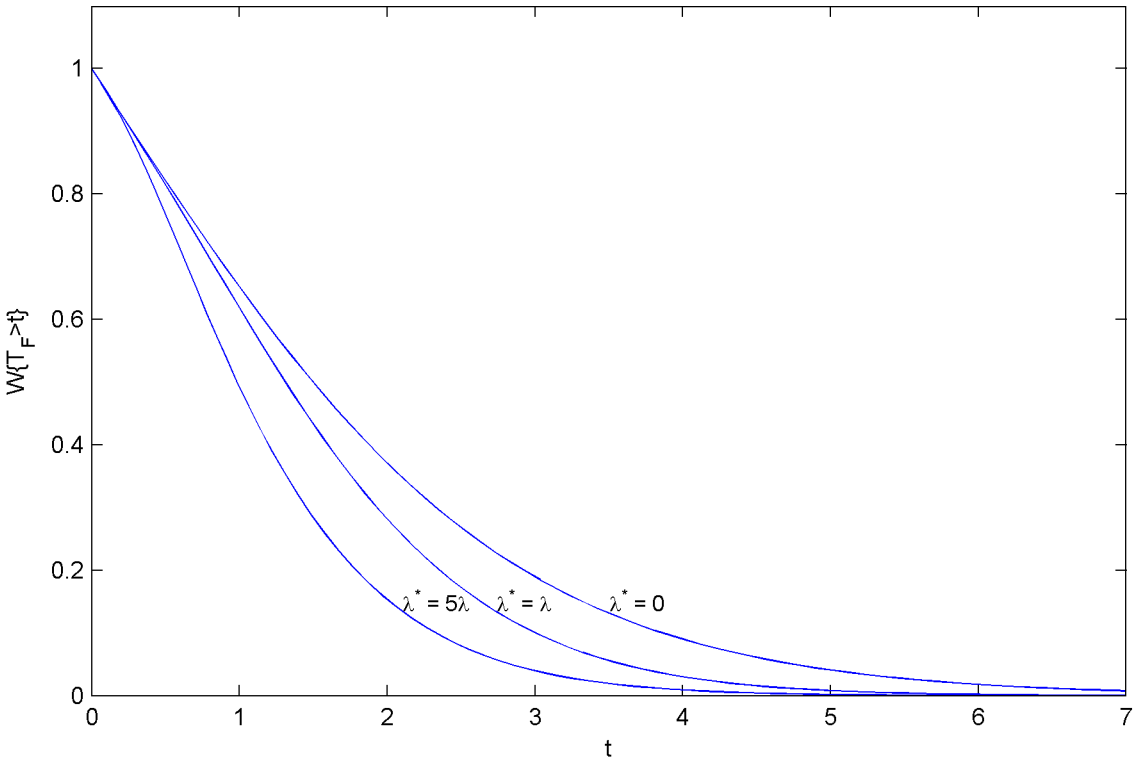


Abbildung 3.56.: M/M/1/S-Warteschlangensystem mit geregelter Bedienrate: Auswirkung von nachfolgenden Anforderungen (neue Ankunftsrate λ^*) auf die Durchflusszeit der Testanforderung.

3.5. Berechnen der Zeit bis zum Erreichen eines bestimmten Systemzustands

Manchmal ist es erforderlich zu wissen, wie lange ein Warteschlangensystem benötigt, um von einem Zustand A in einen anderen Zustand B zu wechseln. Das kann berechnet werden, indem in der Markov-Kette für den (bei Bedarf zuvor erweiterten) Systemzustand alle Übergangsraten vom Zustand B zu anderen Zuständen auf 0 gesetzt werden, und anschließend mit dem in Abschnitt 3.4 erklärten Verfahren berechnet wird, wie lange die Markov-Kette außerhalb dieses absorbierenden Zustands B bleibt, wenn der Anfangszustand A ist.

3.5.1. M/M/1/S-Warteschlangensystem

Es soll ermittelt werden, wie lange in einem zunächst leeren M/M/1/S-Warteschlangensystem mit Ankunftsrate λ und Bedienrate μ ankommende Anforderungen bedient werden, bevor zum ersten Mal eine Anforderung abgewiesen wird, weil das System voll ist.

Modell

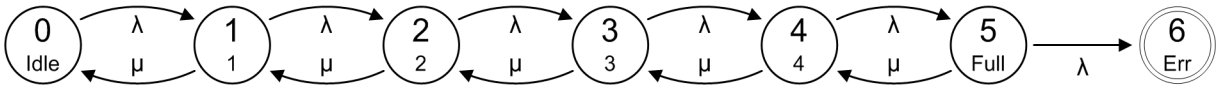


Abbildung 3.57.: M/M/1/S-Warteschlangensystem: Markov-Kette für den Systemzustand, erweitert um den Zustand *Anforderung wird abgewiesen*.

Die entsprechende Markov-Kette ist in Abbildung 3.57 gezeigt. Es wird die Zeit berechnet, die die Markov-Kette benötigt, um vom Zustand 0 *(Leerlauf)* in den Zustand 6 *(Verlust einer Anforderung)* zu gelangen:

$$f_i(0) = 1 \text{ für } i = 0 \ldots 5 \qquad f_6(0) = 0 \tag{3.29}$$

$$f(\tau)' = \mathcal{Q}\, f(\tau) \tag{3.30}$$

$$W\,\{\text{bis zum Zeitpunkt } t \text{ gehen keine Anforderungen verloren}\} = f_0(t) \tag{3.31}$$

Ergebnisse

Die Ergebnisse sind in Abbildung 3.58 zu sehen.

Das für die Untersuchung dieses Warteschlangensystems verwendete MATLAB-Programm *M_M_1_S_Ausfall.m* ist im Anhang B zu finden (Listing B.4).

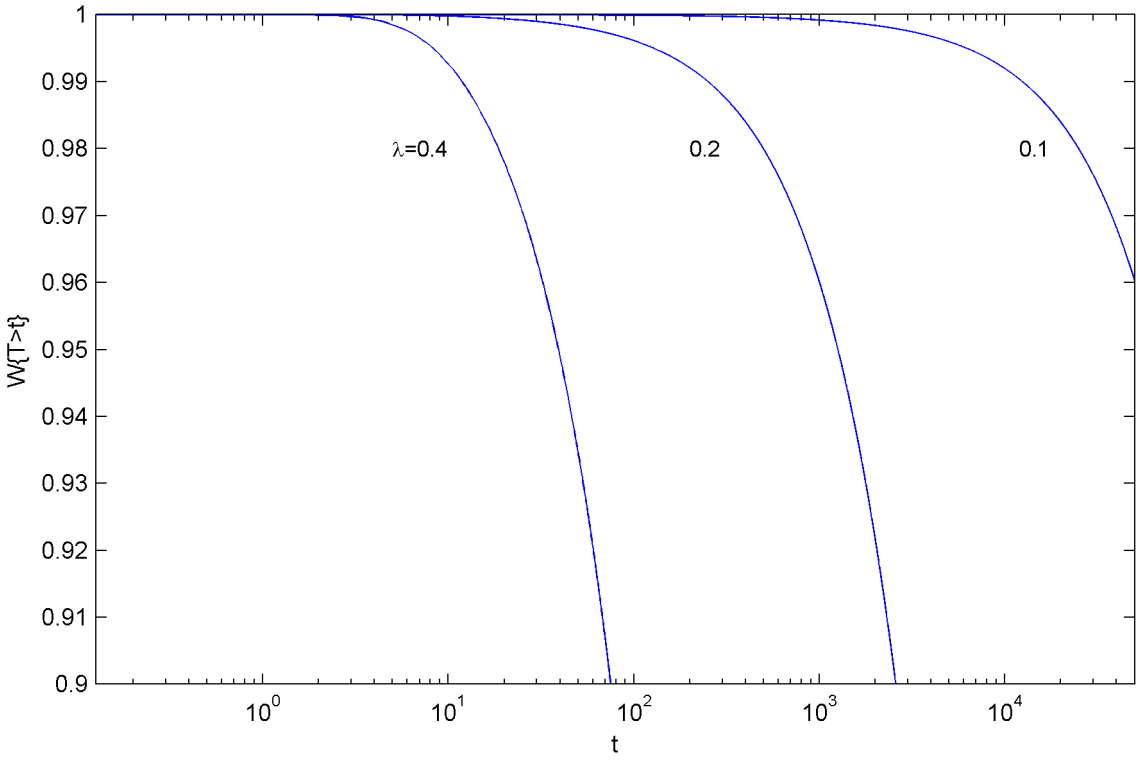


Abbildung 3.58.: M/M/1/S-Warteschlangensystem: Wahrscheinlichkeit, dass länger als eine bestimmte Zeit alle eintreffenden Anforderungen bedient (das heißt nicht abgewiesen) werden.

3.6. Nachbildung allgemeiner Prozesse

Bei Markov-Ketten ist die Zeit zwischen zwei aufeinanderfolgenden Übergängen (das heißt die Verweilzeit in einem Zustand) exponentialverteilt. Mit Markov-Ketten können daher nur solche Prozesse nachgebildet werden, bei denen die Zwischenereigniszeiten ebenfalls exponentialverteilt sind (sogenannte Poisson-Prozesse).

Die Wahrscheinlichkeitsverteilungen von nicht-exponentialverteilten Zwischenereigniszeiten können jedoch oft durch eine der im folgenden Abschnitt gezeigten Wahrscheinlichkeitsverteilungen, die sich durch eine Kombination von mehreren Exponentialverteilungen erzeugen lassen, dargestellt oder angenähert werden.

Die Annäherung kann dabei beliebig genau erfolgen. Jedoch erfordert eine bessere Annäherung im allgemeinen auch eine größere Anzahl von Exponentialverteilungen, wodurch auch die Größe der Markov-Kette steigt, was wiederum zu einem größeren Rechenaufwand führt.

3.6.1. Hypoexponentialverteilung

Wenn eine Anforderung mehrere Stufen mit exponentialverteilten Verweilzeiten (Abbildung 3.59) durchläuft, dann ist die gesamte Durchflusszeit hypoexponentialverteilt.

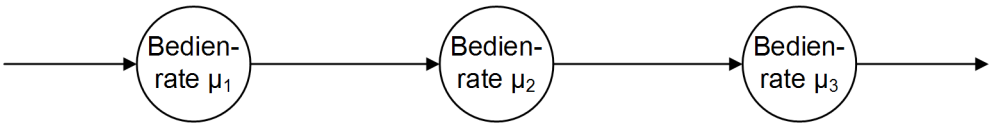


Abbildung 3.59.: Entstehung einer Hypoexponentialverteilung (3 Stufen).

Eine Hypoexponentialverteilung mit k Stufen und den Verweilzeiten $\frac{1}{\mu_i}, 1 \leq i \leq k$ hat die Wahrscheinlichkeitsdichtefunktion

$$f_X(t) = \sum_{i=1}^{k} \left(\prod_{j=1, j\neq i}^{k} \frac{\mu_j}{\mu_j - \mu_i} \right) \mu_i \, e^{-\mu_i t}, \quad t \geq 0 \tag{3.32}$$

und die Verteilungsfunktion

$$F_X(t) = \sum_{i=1}^{k} \left(\prod_{j=1, j\neq i}^{k} \frac{\mu_j}{\mu_j - \mu_i} \right) \left(1 - e^{-\mu_i t}\right), \quad t \geq 0 \tag{3.33}$$

Erwartungswert und Varianz ergeben sich zu

$$\mathrm{E}[X] = \sum_{i=1}^{k} \frac{1}{\mu_i} \tag{3.34}$$

$$\mathrm{Var}[X] = \sum_{i=1}^{k} \frac{1}{\mu_i^2} \tag{3.35}$$

Der Variationskoeffizient ist bei Hypoexponentialverteilungen ≤ 1:

$$c_X = \sqrt{\frac{\sum_{i=1}^{k} \frac{1}{\mu_i^2}}{\sum_{i=1}^{k} \frac{1}{\mu_i^2} + 2 \sum_{i \neq j}^{k} \frac{1}{\mu_i \mu_j}}} \leq 1 \tag{3.36}$$

In Abbildung 3.60 ist gezeigt, wie der Systemzustand eines GI/M/1/S-Warteschlangensystems mit hypoexponentialverteilten Zwischenankunftszeiten modelliert werden kann.

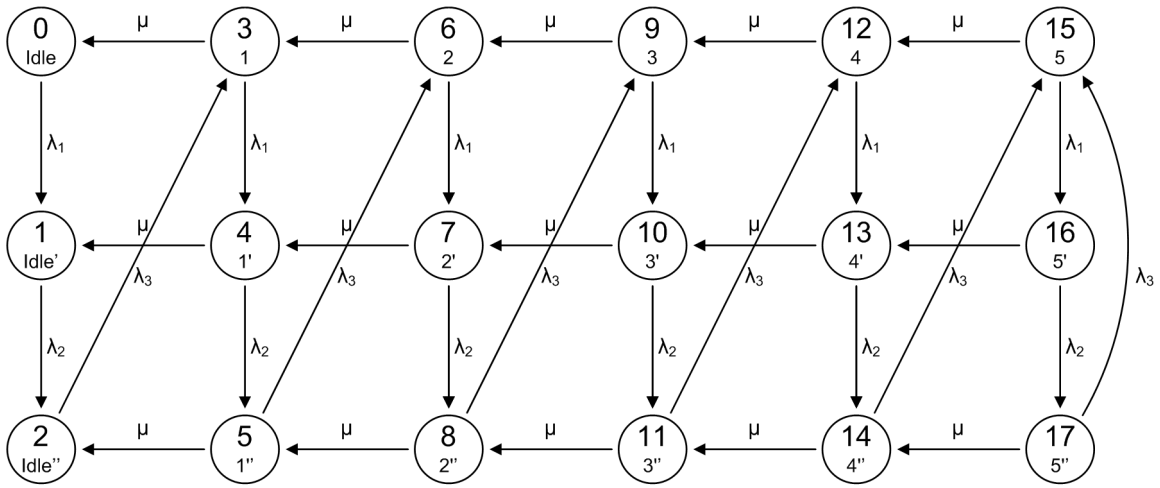


Abbildung 3.60.: Markov-Kette für den Systemzustand eines GI/M/1/S-Warteschlangensystems mit hypoexponentialverteilten Zwischenankunftszeiten.

Zusätzlich zum Zustand des Warteschlangensystems wird nun auch der Zustand des Ankunftsprozesses berücksichtigt. Während Bedienungen weiterhin in einem Schritt stattfinden können, müssen ankommende Anforderungen nun mehrere Stufen durchlaufen. Es ist zu beachten, dass hier ein Zustand des Warteschlangensystems auf mehrere Zustände der Markov-Kette abgebildet wird, zum Beispiel der Zustand *Leerlauf* des Warteschlangensystems wird auf die Zustände *Idle*, *Idle'* und *Idle"* der Markov-Kette abgebildet. Um die Wahrscheinlichkeit eines Zustands des Warteschlangensystems zu berechnen, müssen die Zustandswahrscheinlichkeiten aller zugehörigen Zustände der Markov-Kette addiert werden.

Erlang-Verteilung

Ein wichtiger Spezialfall der Hypoexponentialverteilung ist die Erlang-Verteilung. Sie entsteht, wenn $\mu_1 = \mu_2 = \cdots = \mu_k$ gilt. Sie hat die Wahrscheinlichkeitsdichtefunktion

$$f_X(t) = \frac{\mu^k}{(k-1)!}\, t^{k-1}\, e^{-\mu t}, \quad t \geq 0 \tag{3.37}$$

und die Verteilungsfunktion

$$F_X(t) = 1 - e^{-\mu t} \sum_{j=0}^{k-1} \frac{(\mu t)^j}{j!}, \quad t \geq 0 \tag{3.38}$$

Erwartungswert und Varianz sind

$$\mathrm{E}[X] = \frac{k}{\mu} \tag{3.39}$$

$$\mathrm{Var}[X] = \frac{k}{\mu^2} \tag{3.40}$$

Der Variationskoeffizient ist

$$c_X = \frac{1}{\sqrt{k}} \leq 1 \tag{3.41}$$

3.6.2. Hyperexponentialverteilung

Wenn eine Anforderung zufällig eine von mehreren parallel geschalteten Stufen mit exponentialverteilten Verweilzeiten (Abbildung 3.61) durchläuft, dann ist die gesamte Durchflusszeit hyperexponentialverteilt.

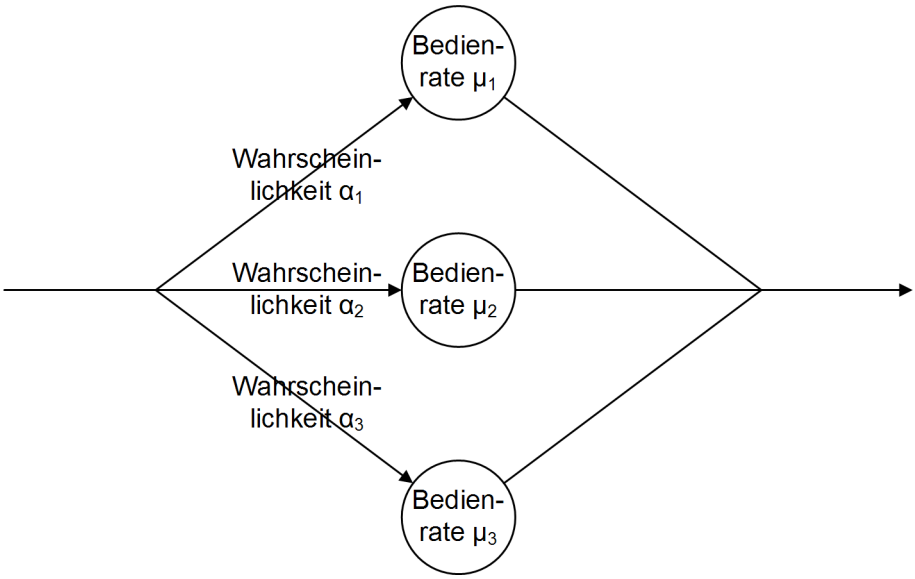


Abbildung 3.61.: Entstehung einer Hyperexponentialverteilung (3 Stufen).

Hyperexponentialverteilungen haben die Wahrscheinlichkeitsdichtefunktion

$$f_X(t) = \sum_{j=1}^{k} \alpha_j \, \mu_j \, e^{-\mu_j t}, \quad t \geq 0 \tag{3.42}$$

und die Verteilungsfunktion

$$F_X(t) = \sum_{j=1}^{k} \alpha_j \left(1 - e^{-\mu_j t}\right), \quad t \geq 0 \tag{3.43}$$

Erwartungswert und Varianz ergeben sich zu

$$\mathrm{E}[X] = \sum_{j=1}^{k} \frac{\alpha_j}{\mu_j} \tag{3.44}$$

$$\mathrm{Var}[X] = 2\sum_{j=1}^{k} \frac{\alpha_j}{\mu_j^2} - \mathrm{E}[X]^2 \tag{3.45}$$

Der Variationskoeffizient ist bei Hyperexponentialverteilungen ≥ 1:

$$c_X = \sqrt{2\frac{1}{\mathrm{E}[X]^2}\sum_{j=1}^{k} \frac{\alpha_j}{\mu_j^2} - 1} \geq 1 \tag{3.46}$$

In Abbildung 3.62 ist gezeigt, wie der Systemzustand eines GI/M/1/S-Warteschlangensystems mit hyperexponentialverteilten Zwischenankunftszeiten modelliert werden kann.

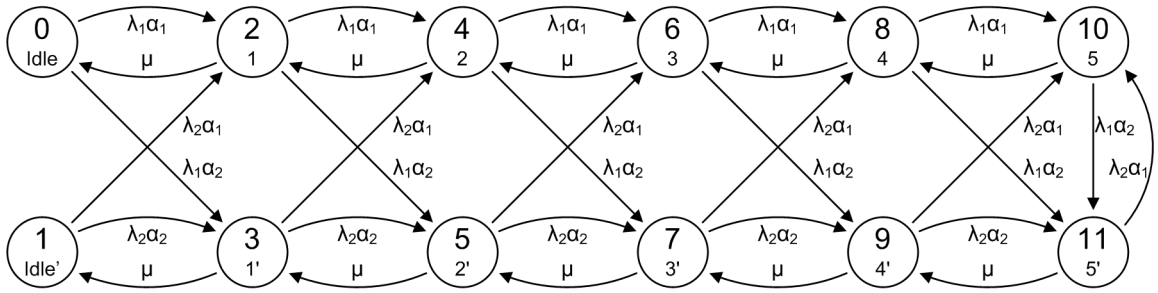


Abbildung 3.62.: Markov-Kette für den Systemzustand eines GI/M/1/S-Warteschlangensystems mit hyperexponentialverteilten Zwischenankunftszeiten.

Auch hier wird jeder Zustand des Warteschlangensystems auf mehrere Zustände in der Markov-Kette abgebildet. Zum Beispiel wird der Zustand *Leerlauf* durch die beiden Zustände *Idle* (mit der Wahrscheinlichkeit α_1) und *Idle'* (mit der Wahrscheinlichkeit α_2) dargestellt.

3.6.3. Cox-Verteilung

Wenn eine Anforderung mehrere Stufen mit exponentialverteilten Verweilzeiten (Abbildung 3.63) durchläuft und dabei nach jeder Stufe mit einer bestimmten Wahrscheinlichkeit das System verläßt, dann ist die gesamte Durchflusszeit Cox-verteilt.

Mit Cox-Verteilungen können sehr viele Wahrscheinlichkeitsverteilungen nachgebildet werden.

In Abbildung 3.64 ist gezeigt, wie der Systemzustand eines GI/M/1/S-Warteschlangensystems mit Cox-verteilten Zwischenankunftszeiten modelliert werden kann.

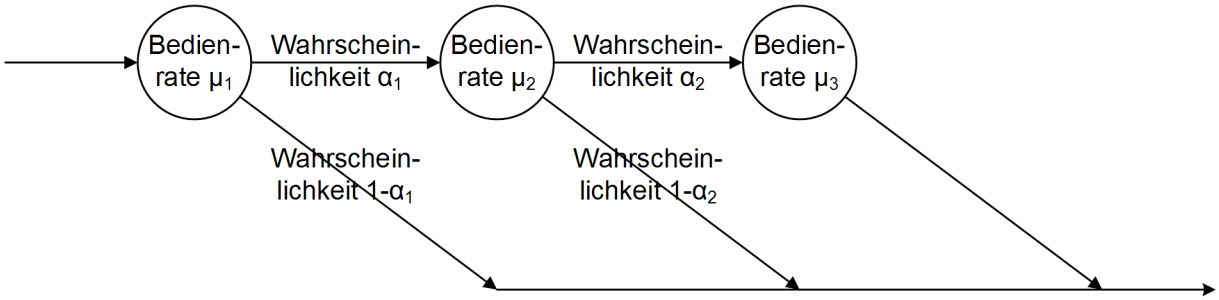


Abbildung 3.63.: Entstehung einer Coxverteilung (3 Stufen).

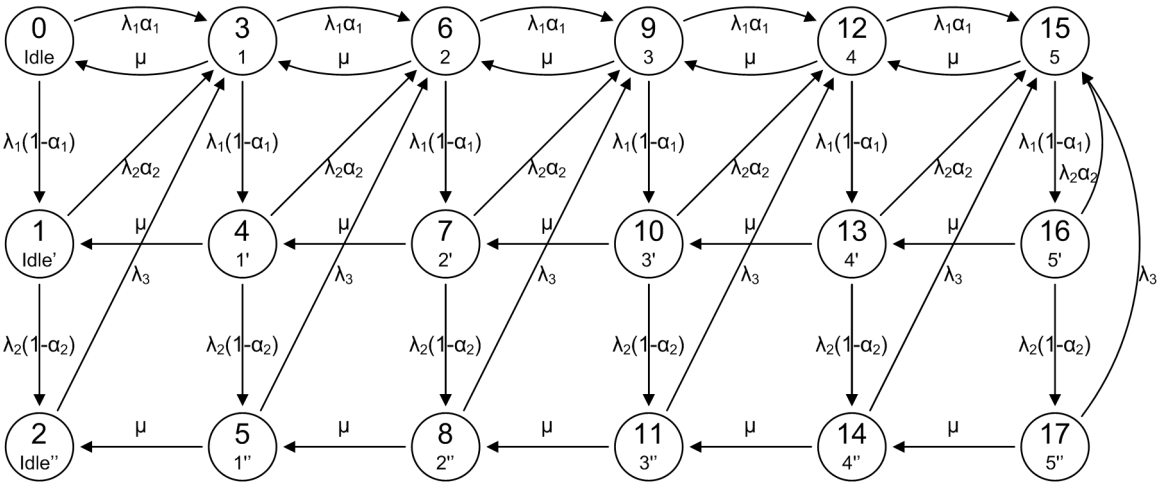


Abbildung 3.64.: Markov-Kette für den Systemzustand eines GI/M/1/S-Warteschlangensystems mit Cox-verteilten Zwischenankunftszeiten.

3.6.4. Finden der passenden Verteilung

Will man eine gegebene, allgemeine Wahrscheinlichkeitsverteilung annähern, muss zuerst eine geeignete Verteilung gewählt werden. Ist das erfolgt, müssen die Parameter der Verteilung (Verweilzeiten, Wahrscheinlichkeiten) bestimmt werden. Für einige Verteilungen gibt es dazu Formeln. Wenn das nicht der Fall ist (beispielsweise bei Cox-Verteilungen mit vielen Stufen), muss mit Näherungsverfahren gearbeitet werden.

Es wird nun gezeigt, wie die Parameter einer zur Annäherung einer gegebenen Verteilung gesuchten Cox-Verteilung mit Hilfe eines evolutionären Algorithmus und dem in Abschnitt 3.4 gezeigten Verfahren zur Bestimmung der Durchflusszeit durch ein Warteschlangensystem bestimmt werden können.

Evolutionärer Algorithmus

Evolutionäre Algorithmen arbeiten mit Mengen *(Populationen)* von möglichen Lösungen *(Individuen)*. Beim hier besprochenen Algorithmus sind die Individuen Cox-Verteilungen, die durch die Verweilzeiten und die Abzweig-Wahrscheinlichkeiten definiert sind. Beginnend bei einer zufälligen Ausgangspopulation werden die Schritte Rekombination, Mutation und Selektion so lange ausgeführt, bis eine oder mehrere Lösungen die gewünschte Qualität erreicht haben (oder ein anderes Abbruchkriterium erfüllt ist).

Bei der Rekombination wird aus zwei (oder mehr) Elternlösungen eine neue Lösung erzeugt. Dabei werden die Eigenschaften der neuen Lösung zufällig aus den Eigenschaften der Eltern zusammengesetzt. Die Rekombination soll ermöglichen, dass sich auf mehrere Lösungen (den Eltern) verteilte gute Eigenschaften in den erzeugten Lösungen vereinen können.

Bei der Mutation werden einzelne Eigenschaften von Individuen zufällig geändert, damit neue, bessere Eigenschaften in die Population einfließen können. Ohne Mutation würden alle Individuen immer nur aus den in der Ausgangspopulation vorhandenen Eigenschaften bestehen.

Bei der Selektion werden aus der Population die besten Individuen ausgewählt. Die Bewertung, wie gut oder schlecht ein Individuum ist, erfolgt mit Hilfe der sogenannten Fitness-Funktion. Die Fitness einer Cox-Verteilung wird durch die Abweichung zur gegebenen Wahrscheinlichkeitsverteilung bestimmt. Je kleiner die Abweichung ist, desto höher ist die Fitness der Cox-Verteilung. Zur Berechnung der Wahrscheinlichkeitsverteilung der Lösung wird zunächst die Markov-Kette für den Durchflussprozess einer Anforderung in dem in Abbildung 3.63 gezeigten System erstellt (Abbildung 3.65). Der Durchflussprozess beginnt im Zustand 1, er endet, wenn der Zustand 4 erreicht ist. Nun kann mit Gleichung (2.51) die komplementäre Verteilungsfunktion für die Durchflusszeit (also die Wahrscheinlichkeit, dass die Anforderung länger als eine bestimmte Zeit im System ist) berechnet und mit der komplementären Verteilungsfunktion der gegebenen Wahrscheinlichkeitsverteilung verglichen werden.

Eine MATLAB-Implementierung des Verfahrens (mit weiterführenden Kommentaren) ist im Anhang C zu finden. Abbildungen 3.66 bis 3.68 zeigen verschiedene Ausgaben des MATLAB-Programms, Abbildung 3.69 zeigt den Vergleich der Warteschlangenlänge in

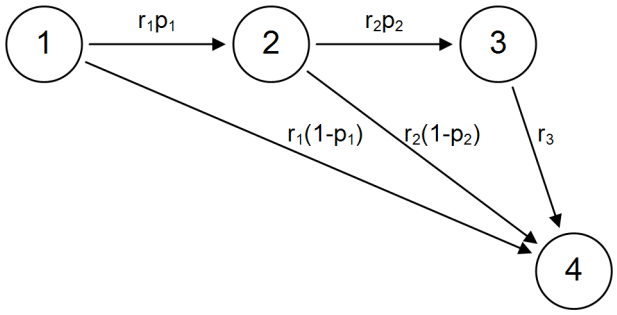


Abbildung 3.65.: Markov-Kette für den Durchflussprozess einer Anforderung im System zur Erzeugung der Cox-Verteilung.

einem GI/M/1/S-Warteschlangensystem zwischen einer gegebenen Verteilung der Zwischenankunftszeiten und der Näherungslösung mit Cox-verteilten Zwischenankunftszeiten.

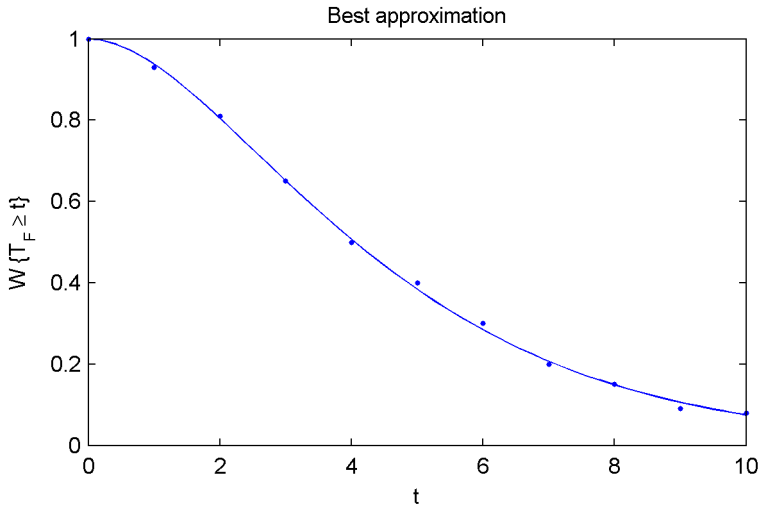


Abbildung 3.66.: Ausgabe des MATLAB-Programms: Die beste gefundene Cox-Verteilung zu einer gegebenen Wahrscheinlichkeitsverteilung.

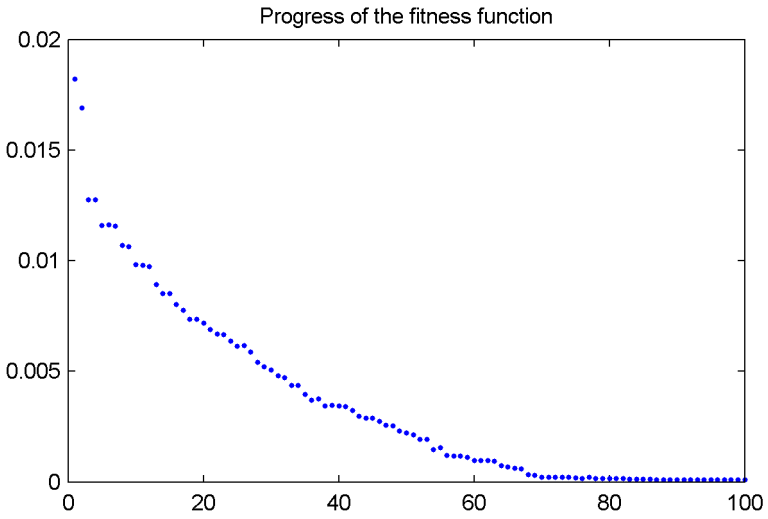


Abbildung 3.67.: Ausgabe des MATLAB-Programms: Der Verlauf der Fitness-Funktion des evolutionären Algorithmus.

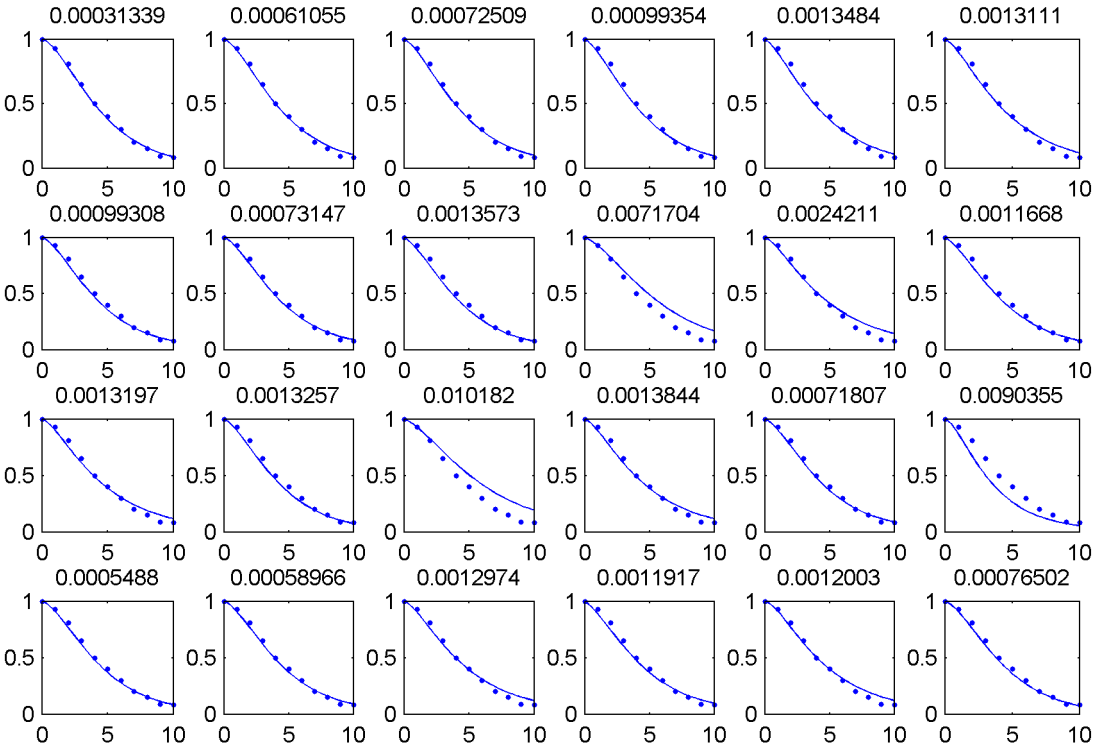


Abbildung 3.68.: Ausgabe des MATLAB-Programms: Eine Population des evolutionären Algorithmus.

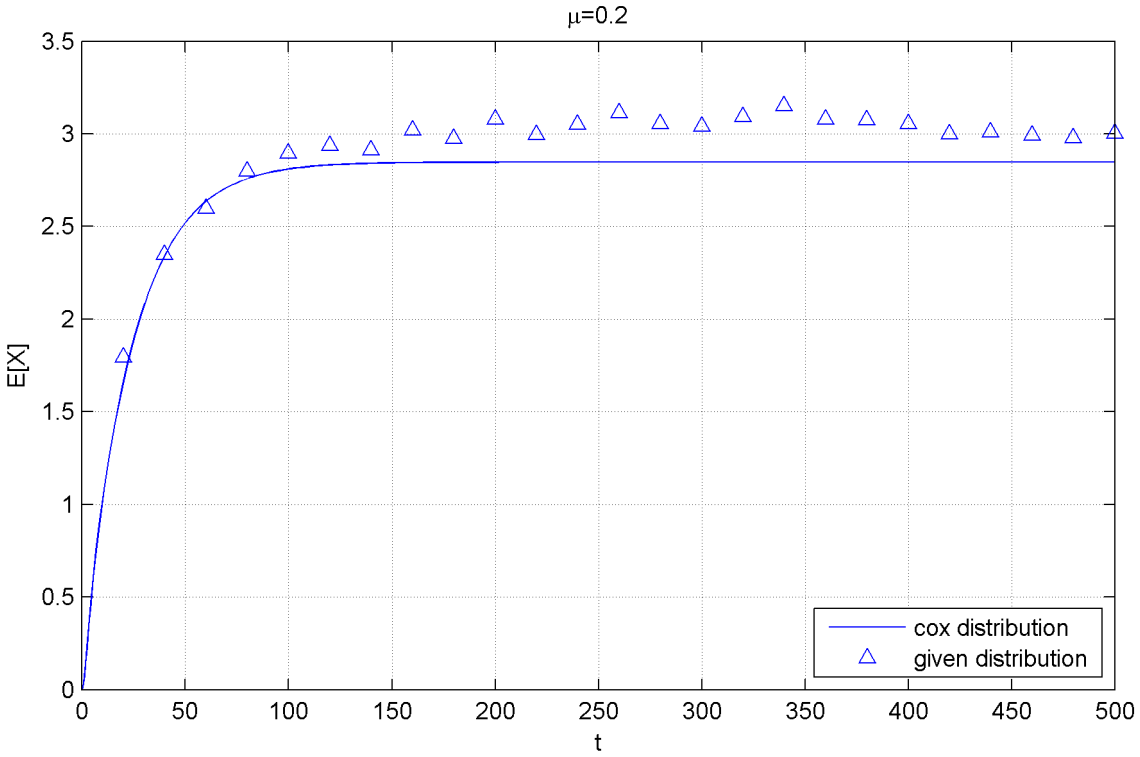


Abbildung 3.69.: Vergleich der Warteschlangenlänge in einem GI/M/1/S-Warteschlangensystem zwischen gegebener Verteilung der Zwischenankunftszeiten und Cox-verteilten Zwischenankunftszeiten.

4. Mathematische Verfahren

Das numerische Berechnen der stationären und transienten Zustandswahrscheinlichkeiten einer Markov-Kette ist in der Praxis sehr kompliziert. Die verwendeten Verfahren müssen schnell sein, sie sollen wenig Speicher benötigen, und sie müssen numerisch stabil sein.

Die Auswahl eines geeigneten Verfahrens aus der großen Anzahl der verfügbaren Verfahren, die richtige Anwendung des gewählten Verfahrens und die Beurteilung der Qualität der Ergebnisse erfordern tiefgehende mathematische Kenntnisse. Im Rahmen dieser Arbeit kann dieses Thema daher nur sehr oberflächlich behandelt werden: Es wird die prinzipielle Vorgehensweise zum Berechnen der Zustandswahrscheinlichkeiten beschrieben, und es werden einige einfache Verfahren vorgestellt.

4.1. Berechnen der stationären Zustandswahrscheinlichkeiten

Bei den Ausführungen in diesem Abschnitt wird davon ausgegangen, dass die stationäre Wahrscheinlichkeitsverteilung existiert und eindeutig ist.

4.1.1. Grundlagen

Die stationären Zustandswahrscheinlichkeiten π_i einer zeitkontinuierlichen Markov-Kette werden berechnet, indem das Gleichungssystem

$$\pi \; \mathcal{Q} = o \tag{4.1}$$

unter der Nebenbedingung

$$\sum_i \pi_i = 1 \tag{4.2}$$

gelöst wird. Dieses Gleichungssystem kann auch geschrieben werden als

$$\mathcal{Q}^T \; \pi^T = o \tag{4.3}$$

Da die Verfahren zum Lösen von Gleichungssystemen auf die zweite Form aufbauen, wird im folgenden diese Form verwendet und zur Vereinfachung $\mathcal{A} = \mathcal{Q}^T$ und $y = \pi^T$ geschrieben. Das Gleichungssystem und die Nebenbedingung werden damit zu

$$\mathcal{A} \; y = o \tag{4.4}$$

und

$$\sum_i y_i = 1 \tag{4.5}$$

Der Rang der Matrix $\mathcal{A}$ ist stets um 1 kleiner als ihre Größe. Die Lösung des Gleichungssystems (4.4) hat daher die Form

$$y = \lambda\, y^*,\ \lambda \in \mathbb{R} \tag{4.6}$$

wobei y^* eine einzelne, beliebige Lösung des Gleichungssystems ist. Es gibt nun zwei Möglichkeiten zum Berechnen der Zustandswahrscheinlichkeiten. Die eine Möglichkeit ist, eine Zeile des Gleichungssystems (4.4) durch die Nebenbedingung (4.5) zu ersetzen:

$$\begin{array}{rcrcrcccrcr} y_1 & + & y_2 & + & y_3 & + & \dots & + & y_N & = & 1 \\ a_{21}y_1 & + & a_{22}y_2 & + & a_{23}y_3 & + & \dots & + & a_{2N}y_N & = & 0 \\ a_{31}y_1 & + & a_{32}y_2 & + & a_{33}y_3 & + & \dots & + & a_{3N}y_N & = & 0 \\ \vdots & & \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{N1}y_1 & + & a_{N2}y_2 & + & a_{N3}y_3 & + & \dots & + & a_{NN}y_N & = & 0 \end{array} \tag{4.7}$$

Dadurch wird der Rang der Matrix $\mathcal{A}$ gleich ihrer Größe, und das Gleichungssystem kann eindeutig gelöst werden. Die andere Möglichkeit ist, zunächst eine beliebige Lösung y^* zu suchen und anschließend jeden Wert dieser Lösung durch die Summe $\sum_i y_i^*$ zu dividieren, damit die Nebenbedingung erfüllt ist. Welche der beiden Möglichkeiten zu bevorzugen ist, hängt vom gewählten Verfahren ab.

4.1.2. Direkte Verfahren

Bei den direkten Verfahren wird das zu lösende Gleichungssystem so lange umgeformt, bis ein äquivalentes Gleichungssystem entsteht, aus dem die Lösung abgelesen oder leicht ausgerechnet werden kann.

Dies erfolgt in einer endlichen und nach oben beschränkten Anzahl von Schritten. Das heißt, direkte Verfahren finden immer die Lösung des Gleichungssystems, wenn sie existiert. Die Anzahl der benötigten Schritte wächst jedoch mit steigender Größe des Gleichungssystems stark an, wodurch direkte Verfahren für die Lösung von sehr großen Gleichungssystemen im allgemeinen nicht gut geeignet sind.

Weiters ist zu beachten, dass während der Anwendung von direkten Verfahren die Koeffizientenmatrix des Gleichungssystems geändert wird, und dass dabei aus dünnbesetzten Matrizen (wie sie bei den in dieser Arbeit besprochenen Gleichungssystemen meist vorkommen) dichtbesetzte Matrizen entstehen können, was bei großen Gleichungssystemen zu Problemen beim Speicherbedarf führen kann.

Direkte Verfahren berechnen bei exakter Rechnung die exakte Lösung eines Gleichungssystems. Da am Computer jedoch aufgrund der endlichen Anzahl von darstellbaren Zahlen die Rechnungen im allgemeinen nicht exakt durchgeführt werden, ist dennoch eine Fehlerabschätzung notwendig.

Das Gaußsche Eliminationsverfahren

Das bekannteste direkte Verfahren zum Lösen linearer Gleichungssysteme ist das Gaußsche Eliminationsverfahren. Hier wird das Gleichungssystem

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_n \end{pmatrix} \tag{4.8}$$

zunächst so umgeformt, dass die Koeffizientenmatrix obere Dreiecksform hat:

$$\begin{pmatrix} a'_{11} & a'_{12} & a'_{13} & \cdots & a'_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a'_{33} & \cdots & a'_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a'_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_n \end{pmatrix} \tag{4.9}$$

Dazu können folgende Operationen verwendet werden:

- Addieren des Vielfachen einer Zeile zu einer anderen Zeile
- Multiplikation einer Zeile mit einem Faktor ungleich 0
- Vertauschen von Zeilen

Es dürfen auch Spalten vertauscht werden, das führt jedoch zu einer Vertauschung der entsprechenden Variablen.

Anschließend wird die Lösung durch „Rückwärtseinsetzen“ berechnet:

$$y_n = \frac{b'_n}{a'_{nn}} \qquad y_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{n} y_j \; a'_{ij} \right) \quad i = n-1 \text{ bis } 1 \tag{4.10}$$

Die Laufzeit des Gaußschen Eliminationsverfahrens ist $O(n^3)$.

4.1.3. Iterative Verfahren

Bei iterativen Verfahren wird eine Folge von Näherungslösungen erzeugt, die gegen die exakte Lösung des Gleichungssystems konvergiert. Normalerweise wird die exakte Lösung nicht erreicht, sondern die Berechnung wird abgebrochen, wenn der Fehler $\mathcal{A}\,y - b$ (das *Residuum*) einen gewissen Wert unterschreitet.

Iterative Verfahren sind gut für sehr große Gleichungssysteme geeignet, da ihre Laufzeit mit steigender Systemgröße nicht so rasch anwächst wie die der direkten Verfahren. Weiters wird die Koeffizientenmatrix des Gleichungssystems nicht verändert, sodass man auch sehr große Gleichungssysteme lösen kann, wenn die Koeffizientenmatrix dünn besetzt ist.

Es gibt mehrere Arten von iterativen Verfahren: Splitting-Methoden, Mehrgitter-Verfahren, Projektionsmethoden und Krylov-Unterraum-Verfahren. Eine Beschreibung einiger wichtiger iterativer Verfahren ist in [Meister 2005] zu finden.

Das Gauß-Seidel-Relaxationsverfahren

Ein oft verwendetes iteratives Verfahren ist das Gauß-Seidel-Relaxationsverfahren. Es gehört zu den Splitting-Methoden und wird oft auch SOR-Verfahren *(successive overrelaxation)* genannt.

Es wird wieder folgendes Gleichungssystem betrachtet:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_n \end{pmatrix}$$

Zunächst wird ein Startvektor $\left(y_1^{(0)}, y_2^{(0)}, \ldots, y_n^{(0)}\right)$ gewählt. Dann wird die Iteration

$$y_i^{(k+1)} = (1-\omega)\, y_i^{(k)} + \omega \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, y_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij}\, y_j^{(k)} \right) \quad i = 1 \text{ bis } n \quad (4.11)$$

solange durchgeführt, bis die gewünschte Genauigkeit erreicht ist.

ω wird Relaxationsparameter genannt. Die Wahl eines geeigneten Relaxationsparameters stellt oft ein Problem dar, denn wie in Abbildung 4.1 zu sehen ist, hängt die Anzahl der benötigten Iterationen (und somit die Laufzeit des Verfahrens) stark von der Wahl dieses Wertes ab. Man kann zwar den optimalen Relaxationsparameter für ein gegebenes Gleichungssystem aus den Eigenwerten der Koeffizientenmatrix berechnen, das ist jedoch sehr aufwendig. Es ist auch nicht möglich, aus dem Verlauf des Residuums im Laufe der ersten Iterationen auf die Qualität des verwendeten Relaxationsparameters zu schließen (siehe Abbildung 4.2).

Der Startvektor der Iteration sollte so gewählt werden, dass er in der Nähe einer Lösung liegt.

4.1.4. Rekursives Lösen

Durch die besondere Struktur der bei der Modellierung von Warteschlangensystemen entstehenden Markov-Ketten ist es oft möglich, durch die Zustandswahrscheinlichkeit eines Zustands die Zustandswahrscheinlichkeiten aller anderen Zustände auszudrücken.

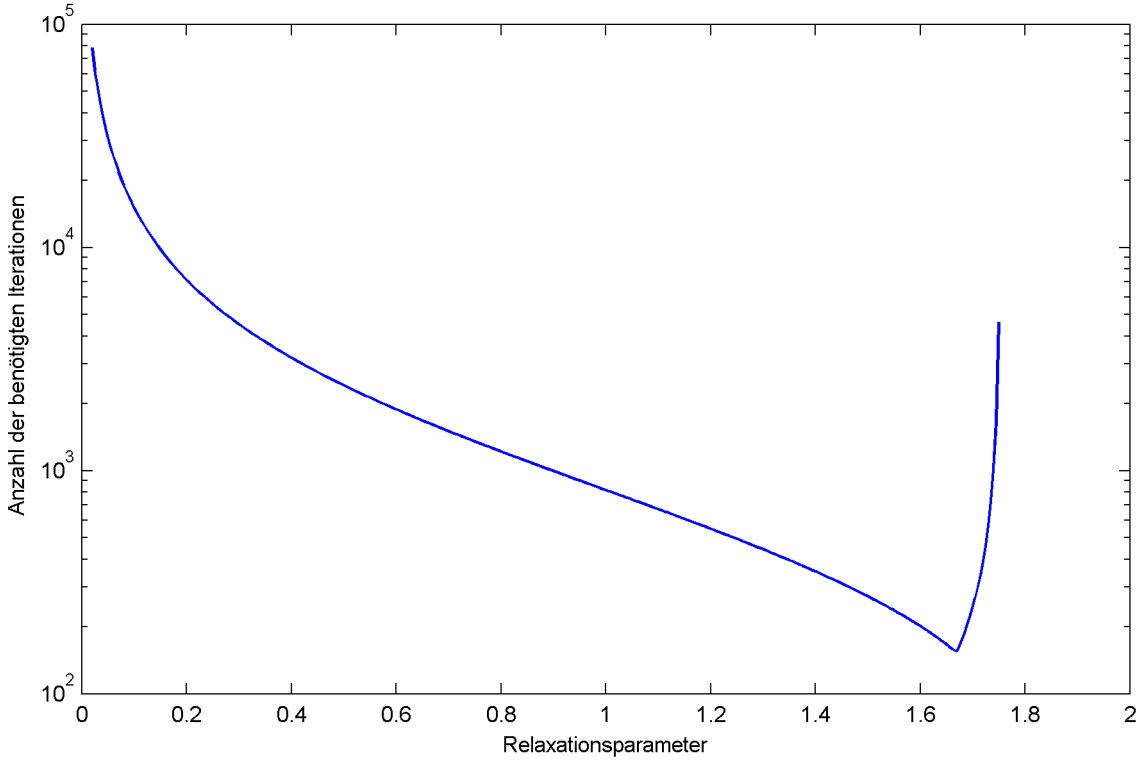


Abbildung 4.1.: Gauß-Seidel-Relaxationsverfahren: Anzahl der zum Lösen eines bestimmten Gleichungssystems benötigten Iterationen in Abhängigkeit vom Relaxationsparameter.

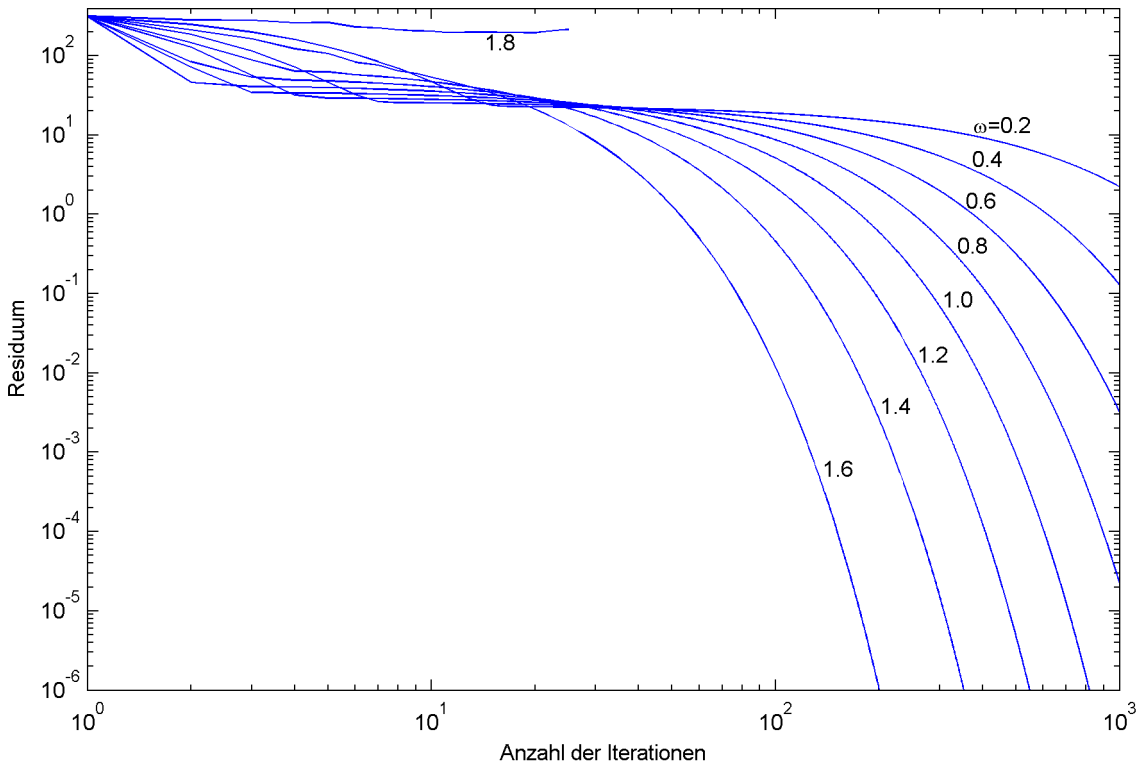


Abbildung 4.2.: Gauß-Seidel-Relaxationsverfahren: Verlauf des Residuums während der Lösung eines bestimmten Gleichungssystems für verschiedene Relaxationsparameter.

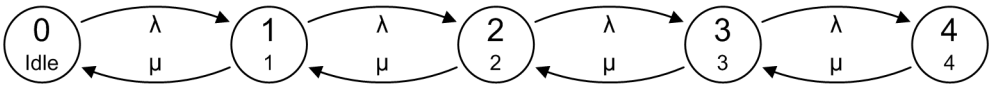


Abbildung 4.3.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems.

Beispielsweise können bei der in Abbildung 4.3 gezeigten Markov-Kette die Zustandswahrscheinlichkeiten π_1 bis π_4 wie folgt durch die Zustandswahrscheinlichkeit π_0 *(Rekursionsstartpunkt)* ausgedrückt werden:

$$\begin{aligned}
\pi_0\,\lambda = \pi_1\,\mu \quad &\Rightarrow \quad \pi_1 = \pi_0\,\tfrac{\lambda}{\mu} \\
\pi_1\,\lambda = \pi_2\,\mu \quad &\Rightarrow \quad \pi_2 = \pi_1\,\tfrac{\lambda}{\mu} = \pi_0\left(\tfrac{\lambda}{\mu}\right)^2 \\
\pi_2\,\lambda = \pi_3\,\mu \quad &\Rightarrow \quad \pi_3 = \pi_2\,\tfrac{\lambda}{\mu} = \pi_0\left(\tfrac{\lambda}{\mu}\right)^3 \\
\pi_3\,\lambda = \pi_4\,\mu \quad &\Rightarrow \quad \pi_4 = \pi_3\,\tfrac{\lambda}{\mu} = \pi_0\left(\tfrac{\lambda}{\mu}\right)^4
\end{aligned} \tag{4.12}$$

Da die Summe aller Zustandswahrscheinlichkeiten gleich 1 sein muss, kann nun der Wert von π_0 berechnet werden:

$$\pi_0 = \frac{1}{\sum_{k=0}^{4}\left(\frac{\lambda}{\mu}\right)^k} \tag{4.13}$$

In [Herzog/Woo/Chandy 1975] wird gezeigt, wie auch kompliziertere Markov-Ketten mit Hilfe dieser Methode berechnet werden können. Im folgenden Abschnitt wird zuerst dieses Verfahren überblicksmäßig beschrieben, anschließend wird eine im Rahmen dieser Diplomarbeit entstandene Erweiterung des Verfahrens vorgestellt, die es möglich macht, das Verfahren ohne händische Eingriffe automatisch ausführen zu lassen.

Original-Verfahren

Das Original-Verfahren funktioniert so:

Aus der Menge der Zustände $\{1, 2, 3, \dots\}$ werden geeignete Rekursionsstartpunkte r_1, r_2, r_3, ... ausgewählt. Dann werden die restlichen Zustandswahrscheinlichkeiten als (rekursive) Funktion der Zustandswahrscheinlichkeiten der Rekursionsstartpunkte ausgedrückt. Um die Zustandswahrscheinlichkeiten der Rekursionsstartpunkte p_{r_1}, p_{r_2}, ... zu berechnen, wird folgendermaßen vorgegangen: Wenn die Anzahl der Rekursionsstartpunkte k ist, werden in k Durchläufen die fiktiven Zustandswahrscheinlichkeiten aller Zustände c_i^k berechnet, basierend auf jeweils unterschiedlichen, fiktiven Werten für die Zustandswahrscheinlichkeiten der Rekursionsstartpunkte. Beim n-ten Durchlauf wird die Zustandswahrscheinlichkeit für den Rekursionsstartpunkt r_n auf 1, die Zustandswahrscheinlichkeiten für alle anderen Rekursionsstartpunkte r_1, ..., r_{n-1}, r_{n+1}, ..., r_k auf 0 gesetzt. Da die tatsächlichen Zustandswahrscheinlichkeiten p_i durch $p_i = \sum_{n=0}^{k} c_i^n p_{r_n}$ dargestellt werden können, können die tatsächlichen Zustandswahrscheinlichkeiten der Rekursionsstartpunkte (unter Zuhilfenahme der Normierungsbedingung $\sum p_i = 1$ und $k-1$ Zeilen aus dem Gleichungssystem der Zustandsübergangsgleichungen) leicht berechnet werden. Das dabei entstehende Gleichungssystem ist um viele Größenordnungen kleiner als das Gleichungssystem der Zustandsübergangsgleichungen und kann daher sehr schnell gelöst werden.

Erweiterung

Um das Verfahren ohne händische Eingriffe automatisch ausführen lassen zu können, mußte für die ersten beiden Punkte des Verfahrens, also das Auswählen der Rekursionsstartpunkte und das Aufstellen der Rekursionsformeln, ein Algorithmus enwickelt werden. Das neue Verfahren sieht nun so aus:

Finden der Rekursionsstartpunkte und des Lösungsweges

1. Initialisierung:
   - Die Menge der Rekursionsstartpunkte ist leer.
   - Die Liste der Lösungsschritte ist leer.
   - Alle Zustände sind als unbekannt markiert.
2. Es wird im Gleichungssystem eine Zeile gesucht, für die die Anzahl x der unbekannten Zustände, die von 0 unterschiedliche Koeffizienten haben, minimal, aber größer als 0 ist.
3. Aus dieser Zeile werden $x - 1$ (beliebige) Zustände zur Menge der Rekursionsstartpunkte hinzugefügt und als bekannt markiert.
4. Es wird im Gleichungssystem eine Zeile n gesucht, für die genau ein unbekannter Zustand a einen von 0 unterschiedlichen Koeffizienten hat. Das Paar $\langle n, a\rangle$ wird zur Liste der Lösungsschritte hinzugefügt und der Zustand a wird als bekannt markiert. Dieser Schritt wird solange wiederholt, bis keine passende Zeile mehr gefunden werden kann.
5. Wenn es noch als unbekannt markierte Zustände gibt, wird bei Schritt 2 weitergemacht.

Berechnen der Zustandswahrscheinlichkeiten

1. Die nächsten beide Schritte werden für jeden Rekursionsstartpunkt ein Mal durchgeführt. Dabei wird die Zustandswahrscheinlichkeit c_s^s für den aktuellen Rekursionsstartpunkt s auf 1 und die Zustandswahrscheinlichkeiten für alle anderen Rekursionsstartpunkte $c_i^s, i \neq s$ auf 0 gesetzt.
2. Die Liste der Lösungsschritte wird vom Anfang bis zum Ende abgearbeitet. Für jeden Eintrag $\langle n, a\rangle$ wird die Zustandswahrscheinlichkeit c_a^s nach der Formel

   $$c_a^s := -\frac{\sum_{i \neq a} c_i^s \; q_{ni}}{q_{na}}$$

   berechnet, wobei s die Nummer des aktuellen Rekursionsstartpunktes ist.
3. Für die Berechnung der tatsächlichen Zustandswahrscheinlichkeiten p_{r_1}, p_{r_2}, …, p_{r_k} der Rekursionsstartpunkte r_1, r_2, …, r_k wird ein Gleichungssystem erzeugt. Die erste Zeile dieses Gleichungssystems ist die Bedingung, dass die Summe aller Zustandswahrscheinlichkeiten gleich 1 sein muss:

   $$p_{r_1} \sum_i c_i^1 + p_{r_2} \sum_i c_i^2 + \cdots + p_{r_k} \sum_i c_i^k = 1$$

Die restlichen $k-1$ Zeilen werden aus dem Gleichungssystem für die Änderung der Zustandswahrscheinlichkeiten genommen. Sie haben das Format

$$p_{r_1} \sum_i q_{ni}\ c_i^1 + p_{r_2} \sum_i q_{ni}\ c_i^2 + \cdots + p_{r_k} \sum_i q_{ni}\ c_i^k = 0$$

Es sind nicht alle Zeilen geeignet. Bei manchen Zeilen ergeben sich alle Koeffizienten zu 0, das Gleichungssystem ist dann unterbestimmt.

4. Das erzeugte Gleichungssystem wird gelöst. Aufgrund der geringen Größe können direkte Verfahren (zum Beispiel das Gaußsche Eliminationsverfahren) verwendet werden.
5. Schritt 2 wird erneut durchgeführt. Die nun entstehenden Werte sind die tatsächlichen Zustandswahrscheinlichkeiten.

Beispiel

Das Verfahren wird anhand eines Beispiels erklärt. Als Beispielsystem dient das in Abschnitt 3.3.3 vorgestellte M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate ($S = 5, S_{\text{stop}} = 4, S_{\text{go}} = 1$). Abbildung 4.4 zeigt die Markov-Kette für den Systemzustand dieses Systems.

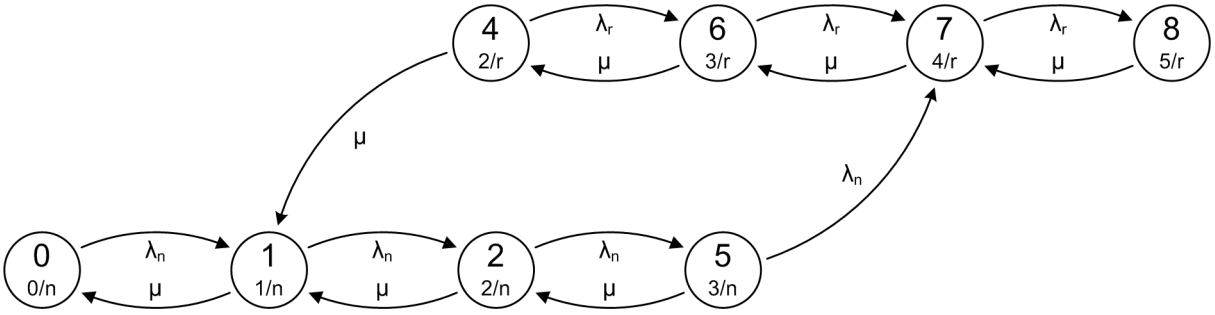


Abbildung 4.4.: Markov-Kette für den Systemzustand eines M/M/1/S-Warteschlangensystems mit geregelter Ankunftsrate.

Folgende Zahlenwerte werden angenommen: $\lambda_1 = 0.8, \lambda_2 = 0.4, \mu = 1$

Die transponierte Matrix der Zustandsübergangsraten lautet:

$$Q^T = \begin{pmatrix}
-\lambda_n & \mu & & & & & & \\
\lambda_n & -\lambda_n - \mu & \mu & \mu & & & & \\
 & \lambda_n & -\lambda_n - \mu & & \mu & & & \\
 & & & -\lambda_r - \mu & & \mu & & \\
 & & \lambda_n & & -\lambda_n - \mu & & & \\
 & & & \lambda_r & & -\lambda_r - \mu & \mu & \\
 & & & & \lambda_n & \lambda_r & -\lambda_r - \mu & \mu \\
 & & & & & & \lambda_r & -\mu
\end{pmatrix}$$

Finden der Rekursionsstartpunkte und des Lösungsweges:

1. Initialisierung, wie oben beschrieben.
2. In folgenden Zeilen ist die Anzahl der unbekannten Zustände, die von 0 unterschiedliche Koeffizienten haben, minimal, aber größer als 0 (nämlich 2): 1,4,5 und 8. Es wird willkürlich Zeile 1 gewählt.
3. Aus Zeile 1 wird willkürlich der Zustand 1 gewählt. Dieser Zustand ist der erste Rekursionsstartpunkt, und er wird als bekannt markiert.
4. Nun werden Zeilen gesucht, für die genau ein unbekannter Zustand einen von 0 unterschiedlichen Koeffizienten hat. Die einzige solche Zeile ist die Zeile 1, der unbekannte Zustand ist der Zustand 2. Das Paar $\langle 1, 2\rangle$ wird zur Liste der Lösungsschritte hinzugefügt.
5. Es wird nach weiteren passenden Zeilen gesucht, die Suche bleibt jedoch erfolglos. Nun muss ein neuer Rekursionsstartpunkt gefunden werden.
6. In folgenden Zeilen ist die Anzahl der unbekannten Zustände, die von 0 unterschiedliche Koeffizienten haben, minimal, aber größer als 0 (nämlich 2): 2, 3, 4, 5 und 8. Es wird willkürlich Zeile 2 gewählt.[1]
7. Aus Zeile 2 wird willkürlich der Zustand 3 gewählt. Dieser Zustand ist der zweite Rekursionsstartpunkt, und er wird als bekannt markiert.
8. Nun werden Zeilen gesucht, für die genau ein unbekannter Zustand einen von 0 unterschiedlichen Koeffizienten hat. Die erste solche Zeile ist die Zeile 2, der unbekannte Zustand ist der Zustand 4. Das Paar $\langle 2, 4\rangle$ wird zur Liste der Lösungsschritte hinzugefügt.
9. Die nächste passende Zeile ist die Zeile 3, der unbekannte Zustand ist der Zustand 5. Das Paar $\langle 3, 5\rangle$ wird zur Liste der Lösungsschritte hinzugefügt.
10. Die weiteren Lösungsschritte lauten: $\langle 4, 6\rangle$, $\langle 6, 7\rangle$, $\langle 7, 8\rangle$. Damit sind alle Zustände bekannt.

Berechnen der Zustandswahrscheinlichkeiten:

1. Es wird $p_1 = 1, p_3 = 0$ gesetzt.
2. Die Liste der Lösungsschritte wird vom Anfang bis zum Ende abgearbeitet:

$$c_2^1 := -\frac{\sum_{i\neq 2} c_i^1\ q_{1i}}{q_{1,2}},\ c_4^1 := -\frac{\sum_{i\neq 4} c_i^1\ q_{2i}}{q_{2,4}},\ c_5^1 := -\frac{\sum_{i\neq 5} c_i^1\ q_{3i}}{q_{3,5}},\ \ldots$$

c_1^1	c_2^1	c_3^1	c_4^1	c_5^1	c_6^1	c_7^1	c_8^1
1.000	0.800	0.000	0.640	-0.640	0.896	0.998	1.551

3. Es wird $p_1 = 0, p_3 = 1$ gesetzt.

[1] Wenn statt der Zeile 2 die Zeile 8 gewählt worden wäre, wären 3 Rekursionsstartpunkte ermittelt worden, obwohl 2 ausreichend sind. Das heißt, der Algorithmus findet nicht immer die minimale Anzahl von Rekursionsstartpunkten!

4. Die Liste der Lösungsschritte wird vom Anfang bis zum Ende abgearbeitet:

$$c_2^2 := -\frac{\sum_{i\neq 2} c_i^2 \; q_{1i}}{q_{1,2}},\; c_4^2 := -\frac{\sum_{i\neq 4} c_i^2 \; q_{2i}}{q_{2,4}},\; c_5^2 := -\frac{\sum_{i\neq 5} c_i^2 \; q_{3i}}{q_{3,5}},\; \ldots$$

c_1^1	c_2^1	c_3^1	c_4^1	c_5^1	c_6^1	c_7^1	c_8^1
0.000	0.000	1.000	-1.000	1.800	-1.400	-1.560	-3.064

5. Das Gleichungssystem für die Berechnung der tatsächlichen Zustandswahrscheinlichkeiten $p_{r_1} = p_1$ und $p_{r_2} = p_3$ wird erzeugt. Die erste Zeile des Gleichungssystems beinhaltet die Bedingung, dass die Summe aller Zustandswahrscheinlichkeiten gleich 1 sein muss. Die zweite Zeile ist die Zeile 5 aus der Koeffizientenmatrix:

$$\begin{array}{llll} p_1 & \sum_i c_i^1 & +p_3 & \sum_i c_i^2 = 1 \\ p_1 & q_{5,3}\; c_5^1 + q_{5,5}\; c_5^1 & +p_3 & q_{5,3}\; c_5^2 + q_{5,5}\; c_5^2 = 0 \end{array}$$

   Das Gleichungssystem wird mit dem Gaußschen Eliminationsverfahren gelöst.

6. Die Liste der Lösungsschritte wird vom Anfang bis zum Ende abgearbeitet:

$$p_2 := -\frac{\sum_{i\neq 2} p_i \; q_{1i}}{q_{1,2}},\; p_4 := -\frac{\sum_{i\neq 4} p_i \; q_{2i}}{q_{2,4}},\; p_5 := -\frac{\sum_{i\neq 5} p_i \; q_{3i}}{q_{3,5}},\; \ldots$$

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
0.308	0.246	0.145	0.052	0.065	0.072	0.081	0.032

4.2. Berechnen der transienten Zustandswahrscheinlichkeiten

Die in diesem Abschnitt gezeigten Verfahren lassen sich natürlich auch verwenden, um die Verteilung der Durchflusszeit durch ein Warteschlangensystem zu berechnen.

Zum Berechnen der transienten Zustandswahrscheinlichkeiten $y_i(x)$ einer zeitkontinuierlichen Markov-Kette muss ein gewöhnliches Differentialgleichungssystem 1. Ordnung gelöst werden. Dieses Differentialgleichungssystem hat folgende Form:

$$\begin{aligned} y_1(x)' &= \sum_{j}^{N} q_{1j}\ y_j(x) \\ y_2(x)' &= \sum_{j}^{N} q_{2j}\ y_j(x) \\ &\dots \\ y_N(x)' &= \sum_{j}^{N} q_{Nj}\ y_j(x) \end{aligned} \tag{4.14}$$

wobei N die Anzahl der Zustände ist. Die Anfangszustände $y_i(0)$ sind bekannt. Wenn die Anzahl der Zustände, wie es meistens der Fall ist, sehr groß ist, kann die exakte Lösung nicht mehr effizient berechnet werden. Stattdessen werden numerische Näherungsverfahren verwendet, die die Funktionswerte nur mehr an Zwischenstellen x_n (mit $x_0 = 0$, $x_{n+1} > x_n$ und $x_{n+1} - x_n = h$ ...„Schrittweite“) berechnen: Beginnend beim bekannten Anfangszustand wird der Funktionsverlauf Schritt für Schritt angenähert, indem versucht wird, aus dem Funktionswert y_n an der Stelle x_n den Funktionswert y_{n+1} an der Stelle x_{n+1} vorherzusagen. Dazu existieren verschiedene Verfahren, die sich durch den Rechenaufwand und die erreichte Genauigkeit unterscheiden. Im folgenden Abschnitt werden einige dieser Verfahren vorgestellt. Der Index i wird dabei der Einfachheit halber weggelassen, die angegebenen Formeln sind immer auf alle Differentialgleichungen des Systems anzuwenden. Weiters wird für die rechte Seite einer Zeile aus dem Differentialgleichungssystem $f\,(x, y(x))$ geschrieben.

4.2.1. Euler-Cauchy-Verfahren und Fehlerabschätzung

Das einfachste Verfahren zum Approximieren von Differentialgleichungen ist das Euler-Cauchy-Verfahren. Hier wird der Verlauf der Kurve zwischen x_n und x_{n+1} durch die Tangente der Kurve am Punkt x_n angenähert:

$$y_{n+1} = y_n + h\, f(x_n, y_n) \tag{4.15}$$

Das Euler-Cauchy-Verfahren ist sehr einfach, aber auch sehr ungenau. Es wird in der Praxis kaum verwendet.

Einen Anhaltspunkt, wie genau ein Verfahren die tatsächlichen Werte annähert, bietet die Ordnung des Verfahrens: Ein Näherungsverfahren hat die Ordnung n, wenn der

Fehler $O(h^{n+1})$ ist. Zum Beispiel wird beim Euler-Cauchy-Verfahren die Funktion durch die ersten beiden Terme der Taylor-Reihe

$$y(x+h) = \sum_{n=0}^{\infty} \frac{y^{(n)}(x)}{n!} h^n \tag{4.16}$$

angenähert:

$$y(x+h) = y(x) + y'(x)h \tag{4.17}$$

Der Fehler ist daher

$$\Delta y(x+h) = \sum_{n=2}^{\infty} \frac{y^{(n)}(x)}{n!} h^n = O\left(h^2\right) \tag{4.18}$$

und das Verfahren hat die Ordnung 1. Im allgemeinen ist der Fehler eines Verfahrens umso geringer, je höher seine Ordnung ist.

4.2.2. Runge-Kutta-Verfahren

Bei den Runge-Kutta-Verfahren wird der Verlauf der Kurve mit Hilfe mehrerer Hilfssteigungen angenähert, wodurch die Ordnung erhöht wird. Das gebräuchlichste Runge-Kutta-Verfahren ist das Runge-Kutta-Verfahren 4. Ordnung:

$$\begin{aligned}
k_1 &= hf\left(x_n, y_n\right) \\
k_2 &= hf\left(x_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}\right) \\
k_3 &= hf\left(x_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2}\right) \\
k_4 &= hf\left(x_n + h, y_n + k_3\right) \\
y_{n+1} &= y_n + \tfrac{k_1}{6} + \tfrac{k_2}{3} + \tfrac{k_3}{3} + \tfrac{k_4}{6} + O\left(h^5\right)
\end{aligned} \tag{4.19}$$

4.2.3. Runge-Kutta-Verfahren mit adaptiver Schrittweitensteuerung

Bei den beiden besprochenen Verfahren hängt die Genauigkeit der Näherung sowohl von der Kurvenform als auch von der (festen) Schrittweite ab. Um über den gesamten Kurvenverlauf einer beliebigen Kurve genaue Ergebnisse zu erzielen, muss man die Schrittweite so klein wählen, das die gewünschte Genauigkeit im „schwierigsten“ Teil der Kurve erreicht wird, was jedoch den Rechenaufwand für den Rest der Kurve unnötig erhöht. Eine Lösung für dieses Problem stellt die adaptive Schrittweitensteuerung dar, bei der die Schrittweite bei jedem Schritt so gewählt wird, dass die geforderte Genauigkeit eingehalten wird, die Schritte aber möglichst groß sind.

Das funktioniert folgendermaßen: Nach jedem Schritt wird die Genauigkeit der Näherung berechnet. Ist die Genauigkeit besser als die geforderte Genauigkeit, dann kann der nächste Schritt größer gewählt werden. Ist die Genauigkeit schlechter als die geforderte Genauigkeit, dann muss der letzte Schritt mit einer kleineren Schrittweite wiederholt werden. Das Ändern der Schrittweite kann entweder um einen festen Wert erfolgen (zum Beispiel Halbieren beziehungsweise Verdoppeln der Schrittweite), oder die neue

Schrittweite kann anhand der erreichten Genauigkeit berechnet werden, was die Anzahl der Schritte bei einer etwa gleichbleibenden Genauigkeit möglichst gering macht.

Zum Berechnen der Genauigkeit der Näherung gibt es im wesentlichen zwei Möglichkeiten. Die erste Möglichkeit ist, den letzten Schritt der Weite h ein zweites Mal mit zwei Schritten der Weite $\frac{h}{2}$ zu wiederholen und die Ergebnisse zu vergleichen. Die zweite (etwas schnellere) Möglichkeit verwendet die eingebetteten Runge-Kutta-Verfahren.

4.2.4. Eingebettete Runge-Kutta-Verfahren

Eingebettete Runge-Kutta-Verfahren beruhen auf der Tatsache, dass es möglich ist, mit bestimmten Mengen von Funktionsauswertungen (Hilfssteigungen) gleichzeitig zwei Runge-Kutta-Verfahren verschiedener Ordnung zu berechnen. Beispielsweise hat ein Runge-Kutta-Verfahren 5. Ordnung folgende Form:

$$\begin{aligned} k_1 &= hf\left(x_n, y_n\right) \\ k_2 &= hf\left(x_n + a_2 h, y_n + b_{21} k_1\right) \\ &\dots \\ k_6 &= hf\left(x_n + a_6\, h, y_n + b_{61}\, k_1 + \dots + b_{65}\, k_5\right) \\ y_{n+1} &= y_n + \sum_{i=1}^{6} c_i k_i + O\left(h^6\right) \end{aligned} \tag{4.20}$$

Bei geeigneter Wahl von a_i und b_{ij} kann mit den gleichen Werten k_i auch ein Runge-Kutta-Verfahren 4. Ordnung berechnet werden:

$$y^*_{n+1} = y_n + \sum_{i=1}^{6} c^*_i\, k_i + O\left(h^5\right) \tag{4.21}$$

Die Differenz der Ergebnisse der beiden Verfahren kann zur Fehlerabschätzung verwendet werden:

$$\Delta y_{n+1} = y_{n+1} - y^*_{n+1} \approx \sum_{i=1}^{6} \left(c_i - c^*_i\right) k_i \tag{4.22}$$

Erstmalig wurden eingebettete Runge-Kutta-Verfahren in [Fehlberg 1969] beschrieben. Seither wurden zahlreiche andere Verfahren entwickelt. Die Werte für das Verfahren nach J. R. Cash and A. H. Karp (5. und 4. Ordnung) können Tabelle 4.2.4 entnommen werden.

i	a_i	b_{i1}	b_{i2}	b_{i3}	b_{i4}	b_{i5}	c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

Tabelle 4.1.: Parameter für das eingebettete Runge-Kutta-Verfahren nach Cash und Karp. Entnommen aus [Press et al. 1992].

5. Zusammenfassung

In dieser Arbeit wurde ein wichtiges Verfahren zur Untersuchung von Warteschlangensystemen gezeigt: die Modellierung mit Markov-Ketten.

Nach einem Überblick über die theoretischen Grundlagen von zeitdiskreten und zeitkontinuierlichen Markov-Ketten in Kapitel 2 wurde in Kapitel 3 ausführlich gezeigt, wie man zu verschiedenen Fragestellungen über Warteschlangensysteme, wie

- Systemzustand (Anzahl der Anforderungen im System, Blockierwahrscheinlichkeit),
- Durchflussprozess (Durchflusszeit, Wartezeit) und
- Berechnen der Zeit bis zum Erreichen eines bestimmten Systemzustands,

die passenden Markov-Ketten erzeugt, und wie dann aus den Eigenschaften der Markov-Ketten Rückschlüsse auf das zugrundeliegende Warteschlangensystem gezogen werden können. Der zum Erstellen von Markov-Ketten benötigte Aufwand kann durch automatisches, regelbasiertes Erzeugen am Computer sehr verringert werden. Es wurde anhand von Beispielen erklärt, wie hier vorgegangen wird. Weiters wurde in diesem Kapitel gezeigt, wie auch Prozesse mit nicht-exponentialverteilten Zwischenereigniszeiten mit Markov-Ketten modelliert werden können.

Zuletzt wurden in Kapitel 4 mathematische Verfahren besprochen, die für das Berechnen der stationären und transienten Zustandswahrscheinlichkeiten von Markov-Ketten verwendet werden können. Dieses wichtige Thema konnte im Rahmen dieser Arbeit nur sehr oberflächlich behandelt werden. Bei der Anwendung in der Praxis muss man ihm, um zuverlässige Ergebnisse zu erhalten, unbedingt mehr Zeit und Aufmerksamkeit widmen.

Mit Markov-Ketten können nicht nur Warteschlangensysteme modelliert werden, sondern auch beispielsweise biologische, physikalische und wirtschaftliche Vorgänge.
Markov-Ketten kommt daher große Bedeutung zu!

A. C++-Programm

Im Rahmen dieser Diplomarbeit wurde ein Programm zur Untersuchung von Warteschlangensystemen mit Markov-Ketten entwickelt, das im folgenden gezeigt wird.

A.1. Abstrakte Basisklasse für Markov-Ketten

Die Klasse *CQueueingSystem* (Listings A.1 und A.2) stellt folgende Funktionen zur Verfügung:

- Erzeugen der Markov-Ketten für Systemzustand und Durchflussprozess in einem Warteschlangensystem
- Berechnen der transienten und stationären Systemzustandswahrscheinlichkeiten
- Berechnen der Verteilung der Durchflusszeit
- Verschiedene Debug-Funktionen (Ausgabe aller Zustände, Ausgabe der Matrix der Zustandsübergangsraten, . . .)

Von dieser Klasse werden alle Klassen für konkrete Warteschlangensysteme abgeleitet.

Listing A.1: QueueingSystem.h

```
1  #ifndef QUEUEINGSYSTEM_H
2  #define QUEUEINGSYSTEM_H QUEUEINGSYSTEM_H
3
4  #include "include.h"
5
6  #include <vector>
7  #include <list>
8  #include <map>
9
10 #include "Matrix/Matrix.h"
11
12 /* base class for queueing stations */
13
14 class CQueueingSystem
15 {
16 public:
17     CQueueingSystem();
18     virtual ~CQueueingSystem();
19
20     // initialization
21     void InitEqSysSystemState(unsigned int iNumberOfStates);
22     void InitEqSysFlowTime(unsigned int iNumberOfStates);
23
24     // Read and write the system state probabilities and the flow time CCDF
25     REAL GetSystemStateProbability(unsigned int iState) {VERIFY(iState<iNumberOfSystemStates)
           ; return aSystemStateProb[iState];};
26     void SetSystemStateProbability(unsigned int iState, REAL fProb) {VERIFY(iState<
           iNumberOfSystemStates); aSystemStateProb[iState]=fProb;};
```

```
27      REAL GetFlowStateSojournTimeCCDF(unsigned int iState) {VERIFY(iState<iNumberOfFlowStates)
            ; return aFlowStateSojournTimeCCDF[iState];};
28      void SetFlowStateSojournTimeCCDF(unsigned int iState, REAL fSojournTimeCDF){ VERIFY(
            iState<iNumberOfFlowStates); aFlowStateSojournTimeCCDF[iState]=fSojournTimeCDF;};
29
30      // Output of the system state probabilities and the flow time CCDF
31      void PrintSystemStateProbabilities(bool xString = false);
32      void WriteToFileSystemStateProbabilities(char * szFileName);
33      void PrintFlowStateSojournTimeCCDF(bool xString = false);
34      void WriteToFileFlowStateSojournTimeCDF(char * szFileName);
35
36      // Get the number of states
37      unsigned int GetNumberOfSystemStates() {return iNumberOfSystemStates;};
38      unsigned int GetNumberOfFlowStates() {return iNumberOfFlowStates;};
39
40      // Coefficient matrices
41      CMatrix matrixSystemState;        // system state probabilities
42      CMatrix matrixFlowTime;         // flow time CCDF
43
44      // Invalidate the matrices (e.g. because a rate has changed)
45      void InvalidateMatrices() {xMatrixSystemStateValid=false;  xMatrixFlowTimeValid=false;};
46
47      // calculate the stationary system state probabilities
48      void CalcStationarySystemStateProbabilities_SOR(bool bInitStartvector=true, float
            fRelaxationParameter=0, int iMaxIterations=1000);
49      void CalcStationarySystemStateProbabilities_Recursive();
50  #ifdef LAPACK
51      void CalcStationarySystemStateProbabilities_LU_Lapack();
52  #endif
53
54      // calculate the transient system state probabilities
55      void CalcTransientSystemStateProbabilities_RK4(REAL fStartX, REAL fStopX, REAL fStepSize,
             void (*Callback)(REAL fX)=NULL);
56      void CalcTransientSystemStateProbabilities_RK4Adapt(REAL fStartX, REAL fStopX, void (*
            Callback)(REAL fX)=NULL);
57
58      // calculate the ccdf of the flow time
59      void CalcFlowStateSojournTimeCCDF_RK4(REAL fStartX, REAL fStopX, REAL fStepSizeX, void (*
            Callback)(REAL fX)=NULL);
60      void CalcFlowStateSojournTimeCCDF_RK4Adapt(REAL fStartX, REAL fStopX, void (*Callback)(
            REAL fX)=NULL);
61
62      // initialization of the calculation of the flow time
63      void InitCalcFlowStateSojournTimeCCDF();
64
65      // Calculate the total flow time CCDF (single CCDFs weighted with the system state
66      // probabilities when the test customer arrived
67      REAL CalcWeightedFlowTimeCCDF();
68
69      // Output of correspondence between system states and flow states
70      void PrintCorrespondingStates();
71
72      // Calculate new coefficient matrices
73      void CalcCoeffMatrixSystemState();
74      void CalcCoeffMatrixFlowTime();
75
76      // Output the states, transition rates, ...
77      virtual const std::string SystemStateToString(unsigned int iStateNumber) {return "?";};
78      virtual const std::string FlowStateToString(unsigned int iStateNumber) {return "?";};
79      void PrintSystemStates();
80      void PrintFlowStates();
81      void PrintSystemStateTransitionRates(bool xString = true);
82      void PrintFlowStateTransitionRates(bool xString = true);
83      void FindAbsorbingSystemStates(bool xString = true);
84
85  protected:
86      // Calculation of a rate between 2 states
87      virtual REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
            iStateNumberDest) {return 0;};
88      virtual REAL GetRateFlowState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest
            ) {return 0;};
89
90      // Number of the states
```

```
91      unsigned int iNumberOfSystemStates;
92      unsigned int iNumberOfFlowStates;
93
94      // Mapping between corresponding system states and flow states
95      std::map<unsigned int,unsigned int> mapCorrespondingStates;
96
97  private:
98      // Probabilities of the system states at the considered point of time
99      REAL * aSystemStateProb;
100
101     // Probabilities of the system states at the point of time when the test customer arrived
102     REAL * aSystemStateProbTA;
103
104     // Probabilities P for all flow states F that no absorbing state is reached at
105     // the considered point of time T, given that the system was in state F at time 0.
106     // This means: if the flow process starts at flow state F, then the probability
107     // that the flowtime is greater then T is equal to P.
108     // "Sojourn" refers to the sojourn outside the absorbing states, not to
109     // the sojourn in the state F
110     REAL * aFlowStateSojournTimeCCDF;
111
112     // The matrices are valid and can be used for calculations
113     bool xMatrixSystemStateValid;
114     bool xMatrixFlowTimeValid;
115
116     // The sum of the system state probabilities which a test customer can
117     // see if it arrives
118     REAL fSystemStateProbSumTA;
119 };
120
121 #endif
```

Listing A.2: QueueingSystem.cpp

```
1  #include "QueueingSystem.h"
2
3  #include <iostream>
4  #include <iomanip>
5  #include <sstream>
6  #include <cmath>
7  #include <fstream>
8
9  #include "Matrix/LEQSolverSOR.h"
10 #include "Matrix/LEQSolverRecursive.h"
11 #include "Matrix/LEQSolverLU_Lapack.h"
12 #include "Matrix/ODESolverRK4.h"
13 #include "Matrix/ODESolverRK4Adapt.h"
14
15 #define VERB_QS_01   // error messages
16 #define VERB_QS_02   // info messages
17
18 //-------------------------------------------------------------------------
19 // Constructor
20 //-------------------------------------------------------------------------
21 CQueueingSystem::CQueueingSystem()
22 {
23     iNumberOfSystemStates = 0;
24     iNumberOfFlowStates = 0;
25     xMatrixSystemStateValid=false;
26     xMatrixFlowTimeValid=false;
27     aSystemStateProb = NULL;
28     aSystemStateProbTA = NULL;
29     aFlowStateSojournTimeCCDF = NULL;
30     fSystemStateProbSumTA = 0;
31 }
32
33 //-------------------------------------------------------------------------
34 // Destructor
35 //-------------------------------------------------------------------------
36 CQueueingSystem::~CQueueingSystem()
37 {
38     if (aSystemStateProb!=NULL) delete[] aSystemStateProb;
39     if (aSystemStateProbTA!=NULL) delete[] aSystemStateProbTA;
```

```
40      if (aFlowStateSojournTimeCCDF!=NULL) delete[] aFlowStateSojournTimeCCDF;
41  }
42
43  //------------------------------------------------------------------------------
44  // Initialization. This function must be called right after the creation of
45  // the state vector for the system states in the derived classes
46  //------------------------------------------------------------------------------
47  void CQueueingSystem::InitEqSysSystemState(unsigned int iNumberOfSystemStates)
48  {
49      VERIFY(iNumberOfSystemStates>0);
50      this->iNumberOfSystemStates = iNumberOfSystemStates;
51      if (aSystemStateProb!=NULL) delete[] aSystemStateProb;
52      if (aSystemStateProbTA!=NULL) delete[] aSystemStateProbTA;
53      aSystemStateProb = new REAL[this->iNumberOfSystemStates];
54      aSystemStateProbTA = new REAL[this->iNumberOfSystemStates];
55  }
56
57  //------------------------------------------------------------------------------
58  // Initialization. This function must be called right after the creation of
59  // the state vector for the flow states in the derived classes
60  //------------------------------------------------------------------------------
61  void CQueueingSystem::InitEqSysFlowTime(unsigned int iNumberOfFlowStates)
62  {
63      VERIFY(iNumberOfFlowStates>0);
64      this->iNumberOfFlowStates = iNumberOfFlowStates;
65      if (aFlowStateSojournTimeCCDF!=NULL) delete[] aFlowStateSojournTimeCCDF;
66      aFlowStateSojournTimeCCDF = new REAL[this->iNumberOfFlowStates];
67  }
68
69  //------------------------------------------------------------------------------
70  // Output of the system state probabilities
71  //------------------------------------------------------------------------------
72  void CQueueingSystem::PrintSystemStateProbabilities(bool xString)
73  {
74      int iItemsPerRow = (xString ? SCREEN_WIDTH / 27 : SCREEN_WIDTH / 20);
75      for (unsigned int i=0; i<iNumberOfSystemStates; i++)
76      {
77          if (xString)
78              std::cout << std::setw(12) << SystemStateToString(i) << "␣:";
79          else
80              std::cout << std::setw(5) << i << "␣:";
81          std::cout << std::setw(13) << aSystemStateProb[i];
82          if (i % iItemsPerRow == (iItemsPerRow-1)) std::cout << std::endl;
83      }
84      if (iNumberOfSystemStates % iItemsPerRow != 0) std::cout << std::endl;
85  }
86
87  //------------------------------------------------------------------------------
88  // Save the system state probabilities to a file
89  //------------------------------------------------------------------------------
90  void CQueueingSystem::WriteToFileSystemStateProbabilities(char * szFileName)
91  {
92      std::ofstream outfile;
93      outfile.open(szFileName,std::ios::out);
94      for (unsigned int i=0; i<iNumberOfSystemStates; i++)
95          outfile << aSystemStateProb[i] << '\n';
96      PRINTTEXT("The␣system␣state␣probabilities␣have␣been␣written␣to␣the␣file␣\'" << szFileName
            << "\'.␣Number␣of␣system␣states:␣" << iNumberOfSystemStates);
97      outfile.close();
98  }
99
100 //------------------------------------------------------------------------------
101 // Output of the sojourn time ccdf of the flow states
102 //------------------------------------------------------------------------------
103 void CQueueingSystem::PrintFlowStateSojournTimeCCDF(bool xString)
104 {
105     int iItemsPerRow = (xString ? SCREEN_WIDTH / 27 : SCREEN_WIDTH / 20);
106     for (unsigned int i=0; i<iNumberOfFlowStates; i++)
107     {
108         if (xString)
109             std::cout << std::setw(12) << FlowStateToString(i) << "␣:";
110         else
111             std::cout << std::setw(5) << i << "␣:";
```

```
112         std::cout << std::setw(13) << aFlowStateSojournTimeCCDF[i];
113         if (i % iItemsPerRow == (iItemsPerRow-1)) std::cout << std::endl;
114     }
115     if (iNumberOfSystemStates % iItemsPerRow != 0) std::cout << std::endl;
116 }
117
118 //-------------------------------------------------------------------------
119 // Save the sojourn time CCDF of the flow states to a file
120 //-------------------------------------------------------------------------
121 void CQueueingSystem::WriteToFileFlowStateSojournTimeCDF(char * szFileName)
122 {
123     std::ofstream outfile;
124     outfile.open(szFileName,std::ios::out);
125     for (unsigned int i=0; i<iNumberOfFlowStates; i++)
126         outfile << aFlowStateSojournTimeCCDF[i] << '\n';
127     PRINTTEXT("The CCDF of the sojourn time of the flow states have been written to the file
        '" << szFileName << "'. Number of flow states: " << iNumberOfFlowStates);
128     outfile.close();
129 }
130
131 //-------------------------------------------------------------------------
132 // Calculate the coefficient matrix for the system state probabilities
133 //-------------------------------------------------------------------------
134 void CQueueingSystem::CalcCoeffMatrixSystemState()
135 {
136     VERIFY(iNumberOfSystemStates>0);
137     #ifdef VERB_QS_02
138         PRINTTEXT("The coefficient matrix for the system state probabilities is calculated...
            ");
139     #endif
140     matrixSystemState.Clear();
141     // bool bError=false;
142     for (unsigned int iRow = 0; iRow < iNumberOfSystemStates; iRow++)
143     {
144         for (unsigned int iCol = 0; iCol < iNumberOfSystemStates; iCol++)
145             if (iRow == iCol)  // element of the main diagonal
146             {
147                 REAL fValue = 0;
148                 for (unsigned int c = 0; c < iNumberOfSystemStates; c++)
149                     fValue += GetRateSystemState (iCol, c);
150                 // bError |= (fValue==0);
151                 matrixSystemState.Set(iRow,iCol,-fValue);
152             }
153             else               // no element of the main diagonal
154             {
155                 REAL fValue = GetRateSystemState (iCol, iRow);
156                 if (fValue!=0)
157                     matrixSystemState.Set(iRow,iCol,fValue);
158             }
159     }
160
161     xMatrixSystemStateValid = true;
162     #ifdef VERB_QS_02
163         PRINTTEXT("Calculation successfully finished.");
164     #endif
165     // if (bError) ...        //
166 }
167
168 //-------------------------------------------------------------------------
169 // Calculate the coefficient matrix for the flow time CCDF
170 //-------------------------------------------------------------------------
171 void CQueueingSystem::CalcCoeffMatrixFlowTime()
172 {
173     VERIFY(iNumberOfSystemStates>0);
174     #ifdef VERB_QS_02
175         std::cout << "The coefficient matrix for the flow time CCDF is calculated..." << std
            ::endl;
176     #endif
177     matrixFlowTime.Clear();
178     bool bError=false;
179     for (unsigned int iRow = 0; iRow < iNumberOfFlowStates; iRow++)
180     {
181         for (int iCol = 0; iCol < iNumberOfFlowStates; iCol++)
```

```
182             if (iRow == iCol)  // element of the main diagonal
183             {
184                 REAL fValue = 0;
185                 for (unsigned int c = 0; c < iNumberOfFlowStates; c++)
186                     fValue += GetRateFlowState (iCol, c);
187                 fValue += GetRateFlowState(iCol, (unsigned int)-1);  // rate to absorbing ↩
                        states
188                 bError |= (fValue==0);
189                 matrixFlowTime.Set(iRow,iCol,-fValue);
190             }
191             else               // no element of the main diagonal
192             {
193                 REAL fValue = GetRateFlowState (iRow, iCol);
194                 if (fValue!=0)
195                     matrixFlowTime.Set(iRow,iCol,fValue);
196             }
197     }
198     xMatrixFlowTimeValid = true;
199     #ifdef VERB_QS_02
200         PRINTTEXT("Calculation␣successfully␣finished.");
201     #endif
202     if (bError)
203         std::cout << "***␣The␣coefficient␣matrix␣for␣the␣flow␣time␣is␣faulty.␣The␣program␣↩
                will␣not␣work.␣***" << std::endl;
204 }
205
206 //------------------------------------------------------------------------------
207 // Output of the correspondence between system states and flow states
208 //------------------------------------------------------------------------------
209 void CQueueingSystem::PrintCorrespondingStates()
210 {
211     for (unsigned int i=0; i<mapCorrespondingStates.size(); i++)
212         std::cout << std::setw(5) << i << std::setw(5) << mapCorrespondingStates[i] << std::↩
                endl;
213 }
214
215 //------------------------------------------------------------------------------
216 // Calculate the weighted flow time CCDF
217 //------------------------------------------------------------------------------
218 REAL CQueueingSystem::CalcWeightedFlowTimeCCDF()
219 {
220     REAL fResult = 0;
221     for( std::map<unsigned int, unsigned int>::iterator iter = mapCorrespondingStates.begin()↩
            ; iter != mapCorrespondingStates.end(); iter++)
222         fResult += aSystemStateProbTA[(*iter).first]*aFlowStateSojournTimeCCDF[(*iter).second↩
                ];
223     return fResult/fSystemStateProbSumTA;
224 }
225
226 //------------------------------------------------------------------------------
227 // Output of the states, transition rates, ...
228 //------------------------------------------------------------------------------
229 void CQueueingSystem::PrintSystemStates()
230 {
231     std::cout << "#␣of␣states:␣" << iNumberOfSystemStates << std::endl;
232     int iItemsPerRow = SCREEN_WIDTH / 25;
233     for (unsigned int i=0; i<iNumberOfSystemStates; i++)
234     {
235         std::cout << std::setw(10) << i << "␣:" << std::setw(15) << SystemStateToString(i);
236         if (i % iItemsPerRow == (iItemsPerRow-1)) std::cout << std::endl;
237     }
238     if (iNumberOfSystemStates % iItemsPerRow != 0) std::cout << std::endl;
239 }
240 //------------------------------------------------------------------------------
241 void CQueueingSystem::PrintFlowStates()
242 {
243     std::cout << "#␣of␣states:␣" << iNumberOfFlowStates << std::endl;
244     int iItemsPerRow = SCREEN_WIDTH / 25;
245     for (unsigned int i=0; i<iNumberOfFlowStates; i++)
246     {
247         std::cout << std::setw(10) << i << "␣:" << std::setw(15) << FlowStateToString(i);
248         if (i % iItemsPerRow == (iItemsPerRow-1)) std::cout << std::endl;
249     }
```

```
250     if (iNumberOfFlowStates % iItemsPerRow != 0) std::cout << std::endl;
251 }
252 //-------------------------------------------------------------------------
253 void CQueueingSystem::PrintSystemStateTransitionRates(bool xString)
254 {
255     for (unsigned int i=0; i<iNumberOfSystemStates; i++)
256         for (unsigned int j=0; j<iNumberOfSystemStates; j++)
257         {
258             if (xString)
259                 std::cout << std::setw(12) << SystemStateToString(i);
260             else
261                 std::cout << std::setw(5) << i;
262             std::cout << " --> ";
263             if (xString)
264                 std::cout << std::setw(12) << SystemStateToString(j);
265             else
266                 std::cout << std::setw(5) << j;
267             std::cout << std::setw(20) << GetRateSystemState(i,j) << std::endl;
268         }
269 }
270 //-------------------------------------------------------------------------
271 void CQueueingSystem::PrintFlowStateTransitionRates(bool xString)
272 {
273     for (unsigned int i=0; i<iNumberOfFlowStates; i++)
274         for (unsigned int j=0; j<iNumberOfFlowStates; j++)
275         {
276             if (xString)
277                 std::cout << std::setw(12) << FlowStateToString(i);
278             else
279                 std::cout << std::setw(5) << i;
280             std::cout << " --> ";
281             if (xString)
282                 std::cout << std::setw(12) << FlowStateToString(j);
283             else
284                 std::cout << std::setw(5) << j;
285             std::cout << std::setw(20) << GetRateFlowState(i,j) << std::endl;
286         }
287 }
288 //-------------------------------------------------------------------------
289 void CQueueingSystem::FindAbsorbingSystemStates(bool xString)
290 {
291     std::cout << "Searching for absorbing system states..." << std::endl;
292     int iItemsPerRow = (xString ? SCREEN_WIDTH / 27 : SCREEN_WIDTH / 20);
293     int iItemCounter = 0;
294     for (unsigned int i=0; i<iNumberOfSystemStates; i++)
295     {
296         if (matrixSystemState.Get(i,i)==0)
297         {
298             if (xString)
299                 std::cout << std::setw(12) << SystemStateToString(i);
300             else
301                 std::cout << std::setw(5) << i;
302             iItemCounter++;
303         }
304         if (iItemCounter % iItemsPerRow == (iItemsPerRow-1)) std::cout << std::endl;
305     }
306     if (iItemCounter % iItemsPerRow != 0) std::cout << std::endl;
307 }
308
309 //*************************************************************************
310 // Calculation of the stationary system state probabilities
311 //*************************************************************************
312
313 //-------------------------------------------------------------------------
314 // stationary system state probabilities - SOR
315 //-------------------------------------------------------------------------
316 void CQueueingSystem::CalcStationarySystemStateProbabilities_SOR(bool bInitStartvector, float
      fRelaxationParameter, int iMaxIterations)
317 {
318     if (!xMatrixSystemStateValid)
319         CalcCoeffMatrixSystemState();
320
321     #ifdef VERB_QS_02
```

```
322         std::cout << "Calculation␣of␣the␣stationary␣system␣state␣probabilities␣is␣started␣(
                SOR)." << std::endl;
323     #endif
324 
325     REAL *aSystemStateProbInit = new REAL[iNumberOfSystemStates];
326     if (bInitStartvector)
327         for (int i=0; i<iNumberOfSystemStates; aSystemStateProbInit[i++]=1);  // startvector:
                (1,1,1,1,...)
328     else
329         for (int i=0; i<iNumberOfSystemStates; aSystemStateProbInit[i++]=aSystemStateProb[i])
                ;
330 
331     CLEQSolverSOR solver(iNumberOfSystemStates, matrixSystemState, NULL, aSystemStateProb);
332     solver.SetParamEps(1e-6);
333     solver.SetParamMaxIterations(iMaxIterations);
334 
335     if (fRelaxationParameter == 0)
336     {
337         #define NUM_RELAX_PARAM 12
338         // if no relaxation parameter is given, try these
339         REAL aRelaxationParameters[NUM_RELAX_PARAM] = {1.2, 1.0, 1.4, 0.8, 1.6, 0.6, 1.8,
                0.4, 1.9, 0.3, 0.2, 0.1} ;
340         for (int j=0; j<NUM_RELAX_PARAM; j++)
341         {
342             for (int i=0; i<iNumberOfSystemStates; aSystemStateProb[i++]=aSystemStateProbInit
                    [i]);
343             solver.SetParamRelaxationParameter(aRelaxationParameters[j]);
344             if (solver.Solve()) goto SOR_ok;
345         }
346     }
347     else
348     {
349         for (int i=0; i<iNumberOfSystemStates; aSystemStateProb[i++]=aSystemStateProbInit[i])
                ;
350         solver.SetParamRelaxationParameter(fRelaxationParameter);
351         if (solver.Solve()) goto SOR_ok;
352     }
353 
354     ERROR("The␣equation␣system␣could␣not␣be␣solved.");
355 
356 SOR_ok:
357     REAL fSum = 0;
358     for (int i = 0; i < iNumberOfSystemStates; fSum += aSystemStateProb[i++]);
359     for (int i = 0; i < iNumberOfSystemStates; aSystemStateProb[i++] /= fSum);
360     #ifdef VERB_QS_02
361         std::cout << "Calculation␣of␣the␣stationary␣system␣state␣probabilities␣successfully␣
                finished" << std::endl;
362     #endif
363     delete[] aSystemStateProbInit;
364 }
365 
366 //-------------------------------------------------------------------------
367 // stationary system state probabilities - recursive
368 //-------------------------------------------------------------------------
369 void CQueueingSystem::CalcStationarySystemStateProbabilities_Recursive()
370 {
371     if (!xMatrixSystemStateValid)
372         CalcCoeffMatrixSystemState();
373     CLEQSolverRecursive solver(iNumberOfSystemStates, matrixSystemState, aSystemStateProb);
374     if (!solver.Solve())
375         ERROR("The␣equation␣system␣could␣not␣be␣solved.");
376 }
377 
378 #ifdef LAPACK
379 //-------------------------------------------------------------------------
380 // stationary system state probabilities - LU factorization (requires LAPACK)
381 //-------------------------------------------------------------------------
382 void CQueueingSystem::CalcStationarySystemStateProbabilities_LU_Lapack()
383 {
384     if (!bMatrixSystemStateValid)
385         CalcCoeffMatrixSystemState();
386     CLEQSolverLU_Lapack solver(iNumberOfSystemStates, matrixSystemState, aSystemStateProb);
387     if (!solver.Solve())
```

```
388         ERROR("The equation system could not be solved.");
389 }
390 #endif
391
392 //*****************************************************************************
393 // Calculation of the transient system state probabilities
394 //*****************************************************************************
395
396 //-----------------------------------------------------------------------------
397 // transient system state probabilities - Runge Kutta 4th order
398 //-----------------------------------------------------------------------------
399 void CQueueingSystem::CalcTransientSystemStateProbabilities_RK4(REAL fStartX, REAL fStopX,
        REAL fStepSize, void (*Callback)(REAL fX))
400 {
401     if (!xMatrixSystemStateValid)
402         CalcCoeffMatrixSystemState();
403     CODESolverRK4 solver(iNumberOfSystemStates, matrixSystemState, aSystemStateProb);
404     solver.SetParamStepSize(fStepSize);
405     solver.Solve(fStartX, fStopX, Callback);
406 }
407
408 //-----------------------------------------------------------------------------
409 // stationary system state probabilities - Runge Kutta 4th order with
410 // adaptive step size control
411 //-----------------------------------------------------------------------------
412 void CQueueingSystem::CalcTransientSystemStateProbabilities_RK4Adapt(REAL fStartX, REAL
        fStopX, void (*Callback)(REAL fX))
413 {
414     if (!xMatrixSystemStateValid)
415         CalcCoeffMatrixSystemState();
416     CODESolverRK4Adapt solver(iNumberOfSystemStates, matrixSystemState, aSystemStateProb);
417     solver.SetParamInitialStepSize(0.1);
418     solver.SetParamMaxError(1e-6);
419     solver.Solve(fStartX, fStopX, Callback);
420 }
421
422 //*****************************************************************************
423 // calculation of the sojourn time (CCDF) of the flow states
424 //*****************************************************************************
425
426 //-----------------------------------------------------------------------------
427 // sojourn time (CCDF) of the flow states - Runge Kutta 4th order
428 //-----------------------------------------------------------------------------
429 void CQueueingSystem::CalcFlowStateSojournTimeCCDF_RK4(REAL fStartX, REAL fStopX, REAL
        fStepSize, void (*Callback)(REAL fX))
430 {
431     if (!xMatrixFlowTimeValid)
432         CalcCoeffMatrixFlowTime();
433     CODESolverRK4 solver(iNumberOfFlowStates, matrixFlowTime, aFlowStateSojournTimeCCDF);
434     solver.SetParamStepSize(fStepSize);
435     solver.Solve(fStartX, fStopX, Callback);
436 }
437
438 //-----------------------------------------------------------------------------
439 // sojourn time (CCDF) of the flow states - Runge Kutta 4th order with
440 // adaptive step size control
441 //-----------------------------------------------------------------------------
442 void CQueueingSystem::CalcFlowStateSojournTimeCCDF_RK4Adapt(REAL fStartX, REAL fStopX, void
        (*Callback)(REAL fX))
443 {
444     if (!xMatrixFlowTimeValid)
445         CalcCoeffMatrixFlowTime();
446     CODESolverRK4Adapt solver(iNumberOfFlowStates, matrixFlowTime, aFlowStateSojournTimeCCDF)
        ;
447     solver.SetParamInitialStepSize(0.1);
448     solver.SetParamMaxError(1e-6);
449     solver.Solve(fStartX, fStopX, Callback);
450 }
451
452 //-----------------------------------------------------------------------------
453 // initialization of the calculation of the flow time
454 // this funtion must be called when the test customer arrives
455 //-----------------------------------------------------------------------------
```

```
456 void CQueueingSystem::InitCalcFlowStateSojournTimeCCDF()
457 {
458     // initialize the CCDF
459     for (unsigned int i=0; i<iNumberOfFlowStates; aFlowStateSojournTimeCCDF[i++]=1);
460 
461     // save the system state probabilities (for weightening)
462     for (unsigned int i=0; i<iNumberOfSystemStates; i++) aSystemStateProbTA[i]=
        aSystemStateProb[i];
463 
464     // calculate the sum of the probabilities of all system states which
465     // the test customer can see when it arrives
466     fSystemStateProbSumTA=0;
467     for( std::map<unsigned int, unsigned int>::iterator iter = mapCorrespondingStates.begin()
        ;
468          iter != mapCorrespondingStates.end(); iter++)
469         fSystemStateProbSumTA += aSystemStateProbTA[(*iter).first];
470     // TRACEVAR(fSystemStateProbSumTA);
471 }
```

A.2. Klassen für die Markov-Ketten von Warteschlangensystemen

Diese Klassen beinhalten die Informationen, die die Klasse *CQueueingSystem* zum Erstellen der Markov-Kette benötigt: eine Liste mit den gültigen Zuständen der Markov-Kette und eine Funktion, die die Zustandsübergangsraten berechnet. Weiters werden hier systemspezifische Funktionen wie das Berechnen der mittleren Anzahl der Anforderungen im System und der Blockierwahrscheinlichkeit bereitgestellt.

A.2.1. M/M/1/S-Warteschlangensystem mit mehreren Klassen von Anforderungen

Die Klasse *CM_M_1_S_Prio.cpp* ist eine abstrakte Basisklasse für Warteschlangensysteme mit mehreren Klassen von Anforderungen.

Listing A.3: M_M_1_S_Prio/M_M_1_S_Prio.h

```
1  #ifndef M_M_1_S_PRIO_H
2  #define M_M_1_S_PRIO_H M_M_1_S_PRIO_H
3  
4  #include "../QueueingSystem.h"
5  
6  #include <vector>
7  
8  #define MAX_CLASS 10         // maximum number of classes
9  
10 struct TSystemState
11 {
12     unsigned int n[MAX_CLASS + 1];   // number of customers for each class. class 0 is not
       used!
13     unsigned int k;                  // state of the server
14 };
15 
16 class M_M_1_S_Prio : public CQueueingSystem
17 {
18 public:
19     M_M_1_S_Prio(unsigned int iNumberOfClasses, unsigned int iSystemSize);
20     void InitStateSpace();
21     void SetArrivalRate(unsigned int iClass, float fArrivalRate);
22     void SetServiceRate(unsigned int iClass, float fServiceRate);
```

```
23      float GetArrivalRate(unsigned int iClass);
24      float GetServiceRate(unsigned int iClass);
25      unsigned int GetSystemSize() {return iSystemSize;};
26      unsigned int GetNumberOfClasses() {return iNumberOfClasses;};
27      unsigned int GetSystemStateNumber(const TSystemState & state);
28      const std::string SystemStateToString(unsigned int iStateNumber);
29      float GetNumberOfCustomersMean(unsigned int iClass);
30  protected:
31      virtual bool SystemStateIsValid(const TSystemState & state) = 0;
32      virtual REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
            iStateNumberDest) = 0;
33      unsigned int iSystemSize;
34      unsigned int iNumberOfClasses;
35      float fArrivalRate[MAX_CLASS+1];
36      float fServiceRate[MAX_CLASS+1];
37      std::vector<TSystemState> vecSystemStates;
38  };
39
40  #endif
```

Listing A.4: M_M_1_S_Prio/M_M_1_S_Prio.cpp

```
1
2   #include "M_M_1_S_Prio.h"
3
4   #include <iostream>
5   #include <iomanip>
6   #include <sstream>
7   #include <cmath>
8
9   //------------------------------------------------------------------------------
10  // Constructor
11  //------------------------------------------------------------------------------
12  M_M_1_S_Prio::M_M_1_S_Prio(unsigned int iNumberOfClasses, unsigned int iSystemSize)
13  : CQueueingSystem()
14  {
15      VERIFY(iNumberOfClasses <= MAX_CLASS);
16
17      this->iSystemSize = iSystemSize;
18      this->iNumberOfClasses = iNumberOfClasses;
19
20      for (unsigned int i = 1; i <= iNumberOfClasses; i++)
21      {
22          fArrivalRate[i] = 0;
23          fServiceRate[i] = 1;
24      }
25  }
26
27  //------------------------------------------------------------------------------
28  // Initialization of the state space. This function must be called at the
29  // end of the constructor of each derived class.
30  //------------------------------------------------------------------------------
31  void M_M_1_S_Prio::InitStateSpace()
32  {
33      TSystemState state;
34
35      for (unsigned int i = 0; i <= MAX_CLASS; state.n[i++] = 0);
36      state.k = 0;  // idle state
37      vecSystemStates.push_back(state);
38      //Debug_PrintSystemStates();
39
40      do
41      {
42          for (state.k = 1; state.k <= iNumberOfClasses; state.k++)
43              if (SystemStateIsValid(state))
44              {
45                  vecSystemStates.push_back(state);
46                  //Debug_PrintSystemStates();
47              }
48          state.n[iNumberOfClasses] ++;
49          for (unsigned int j = iNumberOfClasses; j > 1; j--)
50              if (state.n[j] > iSystemSize)
51              {
```

```
52                  state.n[j] = 0;
53                  state.n[j - 1] ++;
54              }
55      }
56      while (state.n[1] <= iSystemSize);
57
58      //Debug_PrintSystemStates();
59
60      // initialization of the equation system
61      InitEqSysSystemState(vecSystemStates.size());
62  }
63
64
65  //-----------------------------------------------------------------------------
66  // Set the arrival rate
67  //-----------------------------------------------------------------------------
68  void M_M_1_S_Prio::SetArrivalRate(unsigned int iClass, float fArrivalRate)
69  {
70      VERIFY(iClass >= 1 && iClass <= iNumberOfClasses);
71      if (this->fArrivalRate[iClass] != fArrivalRate) InvalidateMatrices();
72      this->fArrivalRate[iClass] = fArrivalRate;
73  }
74
75  //-----------------------------------------------------------------------------
76  // Set the service rate
77  //-----------------------------------------------------------------------------
78  void M_M_1_S_Prio::SetServiceRate(unsigned int iClass, float fServiceRate)
79  {
80      VERIFY(iClass >= 1 && iClass <= iNumberOfClasses);
81      if (this->fServiceRate[iClass] != fServiceRate) InvalidateMatrices();
82      this->fServiceRate[iClass] = fServiceRate;
83  }
84
85  //-----------------------------------------------------------------------------
86  // Get the arrival rate
87  //-----------------------------------------------------------------------------
88  float M_M_1_S_Prio::GetArrivalRate(unsigned int iClass)
89  {
90      VERIFY(iClass >= 1 && iClass <= iNumberOfClasses);
91      return this->fArrivalRate[iClass];
92  }
93
94  //-----------------------------------------------------------------------------
95  // Get the service rate
96  //-----------------------------------------------------------------------------
97  float M_M_1_S_Prio::GetServiceRate(unsigned int iClass)
98  {
99      VERIFY(iClass >= 1 && iClass <= iNumberOfClasses);
100     return this->fServiceRate[iClass];
101 }
102
103 //-----------------------------------------------------------------------------
104 // Get the number of a system state
105 //-----------------------------------------------------------------------------
106 unsigned int M_M_1_S_Prio::GetSystemStateNumber(const TSystemState & state)
107 {
108     for (unsigned int i = 0; i < iNumberOfSystemStates; i++)
109     {
110         TSystemState sv = vecSystemStates[i];
111         bool bFound = true;
112         for (unsigned int j = 1; j <= iNumberOfClasses; j++)
113         {
114             if (state.n[j] != sv.n[j])
115             {
116                 bFound = false;
117                 break;
118             }
119         }
120         bFound = bFound && (state.k == vecSystemStates[i].k);
121         if (bFound)
122             return i;
123     }
124     return (unsigned int)-1;  // not found
```

```
125 }
126
127 //------------------------------------------------------------------------
128 // Get the string representation of a system state
129 //------------------------------------------------------------------------
130 const std::string M_M_1_S_Prio::SystemStateToString(unsigned int iStateNumber)
131 {
132     #define SEPARATOR '/'
133     std::stringstream s;
134
135     TSystemState state = vecSystemStates[iStateNumber];
136     for (unsigned int i = 1; i <= iNumberOfClasses; s << vecSystemStates[iStateNumber].n[i++]
          << SEPARATOR);
137     s << vecSystemStates[iStateNumber].k;
138     return s.str();
139 }
140
141 //------------------------------------------------------------------------
142 // Get the mean number of customers in the system
143 //------------------------------------------------------------------------
144 float M_M_1_S_Prio::GetNumberOfCustomersMean(unsigned int iClass)
145 {
146     float fSum=0;
147     for (unsigned int p=0; p<iNumberOfSystemStates; p++)
148         fSum += vecSystemStates[p].n[iClass]*GetSystemStateProbability(p);
149     return fSum;
150 }
```

Common Buffer, Abfertigungsdisziplin *Nichtunterbrechende Prioritäten*

Listing A.5: M_M_1_S_Prio_CommonBuffer_Prio.h

```
1  #ifndef M_M_1_S_Prio_CommonBuffer_Prio_H
2  #define M_M_1_S_Prio_CommonBuffer_Prio_H
3
4  #include "../M_M_1_S_Prio.h"
5
6  class M_M_1_S_Prio_CommonBuffer_Prio : public M_M_1_S_Prio
7  {
8  public:
9      float GetBlockingProbability(unsigned int iClass);
10     M_M_1_S_Prio_CommonBuffer_Prio(unsigned int iNumberOfClasses, unsigned int iSystemSize);
11 protected:
12     virtual bool SystemStateIsValid(const TSystemState & state);
13     virtual REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
           iStateNumberDest);
14 };
15
16 #endif
```

Listing A.6: M_M_1_S_Prio_CommonBuffer_Prio.cpp

```
1  #include "M_M_1_S_Prio_CommonBuffer_Prio.h"
2
3  //------------------------------------------------------------------------
4  // Constructor
5  //------------------------------------------------------------------------
6
7  M_M_1_S_Prio_CommonBuffer_Prio::M_M_1_S_Prio_CommonBuffer_Prio(unsigned int iNumberOfClasses,
       unsigned int iSystemSize)
8  : M_M_1_S_Prio(iNumberOfClasses,iSystemSize)
9  {
10     InitStateSpace();
11 }
12
13 //------------------------------------------------------------------------
14 // Get the transition rate between two system states
15 // (priority service)
16 //------------------------------------------------------------------------
```

```
17 REAL M_M_1_S_Prio_CommonBuffer_Prio::GetRateSystemState(unsigned int iStateNumberSrc, )
       unsigned int iStateNumberDest)
18 {
19     VERIFY(iStateNumberSrc<iNumberOfSystemStates);
20     VERIFY(iStateNumberDest<iNumberOfSystemStates);
21
22     //PRINTVAR(iStateNumberSrc);
23     //PRINTVAR(iStateNumberDest);
24
25     TSystemState s = vecSystemStates[iStateNumberSrc];
26     TSystemState t = vecSystemStates[iStateNumberDest];
27
28     //std::cout << SystemStateToString(iStateNumberSrc,'/') << std::endl;
29     //std::cout << SystemStateToString(iStateNumberDest,'/') << std::endl;
30
31
32     // idle state, arrival of a customer of class i
33     if (iStateNumberSrc==0)
34     {
35         for (unsigned int i=1; i<=iNumberOfClasses; i++)
36         {
37             bool bMatch=true;
38             for (unsigned int j=1; j<i; j++)
39                 bMatch = bMatch && (t.n[j]==0);
40             bMatch = bMatch && (t.n[i]==1);
41             for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
42                 bMatch = bMatch && (t.n[j]==0);
43             if (bMatch) return fArrivalRate[i];
44         }
45         return 0;
46     }
47
48     // the last customer (class i) has been served, idle state is reached
49     if (iStateNumberDest==0)
50     {
51         for (unsigned int i=1; i<=iNumberOfClasses; i++)
52         {
53             bool bMatch=true;
54             for (unsigned int j=1; j<i; j++)
55                 bMatch = bMatch && (s.n[j]==0);
56             bMatch = bMatch && (s.n[i]==1);
57             for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
58                 bMatch = bMatch && (s.n[j]==0);
59             if (bMatch) return fServiceRate[i];
60         }
61         return 0;
62     }
63
64     // arrival of a customer of class i
65     if (t.k==s.k)
66     {
67         for (unsigned int i=1; i<=iNumberOfClasses; i++)
68             {
69                 bool bMatch=true;
70                 for (unsigned int j=1; j<i; j++)
71                     bMatch = bMatch && (t.n[j]==s.n[j]);
72                 bMatch = bMatch && (t.n[i]==s.n[i]+1);
73                 for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
74                     bMatch = bMatch && (t.n[j]==s.n[j]);
75                 if (bMatch)
76                     return fArrivalRate[i];
77             }
78     }
79
80     // a customer has been served
81     {
82         bool bMatch=true;
83         for (unsigned int j=1; j<s.k; j++)
84             bMatch = bMatch && (t.n[j]==s.n[j]);
85         bMatch = bMatch && (t.n[s.k]==s.n[s.k]-1);
86         for (unsigned int j=s.k+1; j<=iNumberOfClasses; j++)
87             bMatch = bMatch && (t.n[j]==s.n[j]);
88         if (bMatch)
```

```
89          {
90              for (unsigned int j=1; j<t.k; j++)
91                  if (t.n[j]!=0) return 0;
92              return fServiceRate[s.k];
93          }
94      }
95
96      // there are no other transitions
97      return 0;
98  }
99
100 //------------------------------------------------------------------------------
101 // Check if a system state is valid
102 //------------------------------------------------------------------------------
103 bool M_M_1_S_Prio_CommonBuffer_Prio::SystemStateIsValid(const TSystemState & state)
104 {
105     // the number of customers must not exceed the buffer size
106     unsigned int iSum=0;
107     for (unsigned int j=1; j<=iNumberOfClasses; j++)
108         iSum += state.n[j];
109     if (iSum>iSystemSize) return false;
110
111     // if there is no customer of class j in the system, the server
112     // cannot serve a customer of class j
113     for (unsigned int j=1; j<=iNumberOfClasses; j++)
114         if ((state.n[j]==0) && (state.k==j)) return false;
115
116     // if the server is idle, the system must be empty
117     if (state.k==0)
118         for (unsigned int j=1; j<iNumberOfClasses; j++)
119             if (state.n[j]!=0) return false;
120
121     return true;
122 }
123
124 //------------------------------------------------------------------------------
125 // Get the probability that the system is full (blocking probability)
126 //------------------------------------------------------------------------------
127 float M_M_1_S_Prio_CommonBuffer_Prio::GetBlockingProbability(unsigned int /*iClass*/)
128 {
129     TSystemState state;
130     unsigned int i;
131     float p=0;
132     for (i=0;i<iNumberOfSystemStates;i++)
133     {
134         unsigned int iSum=0;
135         for (unsigned int j=1; j<=iNumberOfClasses; j++)
136             iSum += vecSystemStates[i].n[j];
137         if (iSum==iSystemSize)
138             p+=GetSystemStateProbability(i);
139     }
140     return p;
141 }
```

Common Buffer, Abfertigungsdisziplin *Abfertigung in zufälliger Reihenfolge*

Listing A.7: M_M_1_S_Prio_CommonBuffer_Random.h

```
1  #ifndef M_M_1_S_Prio_CommonBuffer_Random_H
2  #define M_M_1_S_Prio_CommonBuffer_Random_H
3
4  #include "../M_M_1_S_Prio.h"
5
6  class M_M_1_S_Prio_CommonBuffer_Random : public M_M_1_S_Prio
7  {
8  public:
9      float GetBlockingProbability(unsigned int iClass);
10     M_M_1_S_Prio_CommonBuffer_Random(unsigned int iNumberOfClasses, unsigned int iSystemSize)
       ;
11 protected:
12     virtual bool SystemStateIsValid(const TSystemState & state);
```

```
13      virtual REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
            iStateNumberDest);
14  };
15
16  #endif
```

Listing A.8: M_M_1_S_Prio_CommonBuffer_Random.cpp

```
1   #include "M_M_1_S_Prio_CommonBuffer_Random.h"
2
3   //------------------------------------------------------------------------------
4   // Constructor
5   //------------------------------------------------------------------------------
6
7   M_M_1_S_Prio_CommonBuffer_Random::M_M_1_S_Prio_CommonBuffer_Random(unsigned int
        iNumberOfClasses, unsigned int iSystemSize)
8   : M_M_1_S_Prio(iNumberOfClasses,iSystemSize)
9   {
10      InitStateSpace();
11  }
12
13  //------------------------------------------------------------------------------
14  // Get the transition rate between two system states
15  // (random service)
16  //------------------------------------------------------------------------------
17  REAL M_M_1_S_Prio_CommonBuffer_Random::GetRateSystemState(unsigned int iStateNumberSrc,
        unsigned int iStateNumberDest)
18  {
19      VERIFY(iStateNumberSrc<iNumberOfSystemStates);
20      VERIFY(iStateNumberDest<iNumberOfSystemStates);
21
22      TSystemState s = vecSystemStates[iStateNumberSrc];
23      TSystemState t = vecSystemStates[iStateNumberDest];
24
25      // idle state, arrival of a customer of class i
26      if (iStateNumberSrc==0)
27      {
28          for (unsigned int i=1; i<=iNumberOfClasses; i++)
29          {
30              bool bMatch=true;
31              for (unsigned int j=1; j<i; j++)
32                  bMatch = bMatch && (t.n[j]==0);
33              bMatch = bMatch && (t.n[i]==1);
34              for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
35                  bMatch = bMatch && (t.n[j]==0);
36              if (bMatch) return fArrivalRate[i];
37          }
38          return 0;
39      }
40
41      // the last customer (class i) has been served, idle state is reached
42      if (iStateNumberDest==0)
43      {
44          for (unsigned int i=1; i<=iNumberOfClasses; i++)
45          {
46              bool bMatch=true;
47              for (unsigned int j=1; j<i; j++)
48                  bMatch = bMatch && (s.n[j]==0);
49              bMatch = bMatch && (s.n[i]==1);
50              for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
51                  bMatch = bMatch && (s.n[j]==0);
52              if (bMatch) return fServiceRate[i];
53          }
54          return 0;
55      }
56
57      // arrival of a customer of class i
58      if (t.k==s.k)
59      {
60          for (unsigned int i=1; i<=iNumberOfClasses; i++)
61              {
62                  bool bMatch=true;
63                  for (unsigned int j=1; j<i; j++)
```

```
64                      bMatch = bMatch && (t.n[j]==s.n[j]);
65                  bMatch = bMatch && (t.n[i]==s.n[i]+1);
66                  for (unsigned int j=i+1; j<=iNumberOfClasses; j++)
67                      bMatch = bMatch && (t.n[j]==s.n[j]);
68                  if (bMatch)
69                      return fArrivalRate[i];
70              }
71      }
72
73      // a customer has been served
74      {
75          bool bMatch=true;
76          for (unsigned int j=1; j<s.k; j++)
77              bMatch = bMatch && (t.n[j]==s.n[j]);
78          bMatch = bMatch && (t.n[s.k]==s.n[s.k]-1);
79          for (unsigned int j=s.k+1; j<=iNumberOfClasses; j++)
80              bMatch = bMatch && (t.n[j]==s.n[j]);
81          if (bMatch)
82          {
83              unsigned int iSum=0;
84              for (unsigned int j=1; j<=iNumberOfClasses; iSum += t.n[j++]);
85              return fServiceRate[s.k]*t.n[t.k]/iSum;
86          }
87      }
88
89      // there are no other transitions
90      return 0;
91  }
92
93  //---------------------------------------------------------------------------
94  // Check if a system state is valid
95  //---------------------------------------------------------------------------
96  bool M_M_1_S_Prio_CommonBuffer_Random::SystemStateIsValid(const TSystemState & state)
97  {
98      // the number of customers must not exceed the buffer size
99      unsigned int iSum=0;
100     for (unsigned int j=1; j<=iNumberOfClasses; j++)
101         iSum += state.n[j];
102     if (iSum>iSystemSize) return false;
103
104     // if there is no customer of class j in the system, the server
105     // cannot serve a customer of class j
106     for (unsigned int j=1; j<=iNumberOfClasses; j++)
107         if ((state.n[j]==0) && (state.k==j)) return false;
108
109     // if the server is idle, the station must be empty
110     if (state.k==0)
111         for (unsigned int j=1; j<iNumberOfClasses; j++)
112             if (state.n[j]!=0) return false;
113
114     return true;
115 }
116
117 //---------------------------------------------------------------------------
118 // Get the probability that the system is full (blocking probability)
119 //---------------------------------------------------------------------------
120 float M_M_1_S_Prio_CommonBuffer_Random::GetBlockingProbability(unsigned int /*iClass*/)
121 {
122     TSystemState state;
123     unsigned int i;
124     float p=0;
125     for (i=0;i<iNumberOfSystemStates;i++)
126     {
127         unsigned int iSum=0;
128         for (unsigned int j=1; j<=iNumberOfClasses; j++)
129             iSum += vecSystemStates[i].n[j];
130         if (iSum==iSystemSize)
131             p+=GetSystemStateProbability(i);
132     }
133     return p;
134 }
```

A.2.2. M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate

Listing A.9: M_M_1_S_TwoPointA/M_M_1_S_TwoPointA.h

```
#ifndef M_M_1_S_TWOPOINTA_H
#define M_M_1_S_TWOPOINTA_H M_M_1_S_TWOPOINTA_H

#include "../QueueingSystem.h"

struct TSystemState
{
    unsigned int n;     // number of customers in the system
    unsigned int s;     // arrival rate: 0 = normal, 1 = reduced
};

class M_M_1_S_TwoPointA : public CQueueingSystem
{
public:
    M_M_1_S_TwoPointA(unsigned int iSystemSize, unsigned int iThresholdStop, unsigned int
        iThresholdGo);
    void SetArrivalRateNormal(float fArrivalRate);
    void SetArrivalRateReduced(float fArrivalRate);
    void SetServiceRate(float fServiceRate);
    float GetArrivalRateNormal() {return fArrivalRateNormal;};
    float GetArrivalRateReduced() {return fArrivalRateReduced;};
    float GetServiceRate() {return fServiceRate;};
    unsigned int GetSystemSize() {return iSystemSize;};
    float GetNumberOfCustomersMean();
    float GetProbabilitySwitchReduced();
    float GetBlockingProbability();
    unsigned int GetSystemStateNumber(const TSystemState & state);
    const std::string SystemStateToString(unsigned int iStateNumber, char cSeperator='-');
protected:
    std::vector<struct TSystemState> vecSystemStates;
    bool SystemStateIsValid(const TSystemState & state);
    REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest);
    unsigned int iSystemSize;
    unsigned int iThresholdStop;
    unsigned int iThresholdGo;
    float fArrivalRateNormal;
    float fArrivalRateReduced;
    float fServiceRate;
};

#endif
```

Listing A.10: M_M_1_S_TwoPointA/M_M_1_S_TwoPointA.cpp

```
#include "M_M_1_S_TwoPointA.h"

#include <iostream>
#include <iomanip>
#include <sstream>
#include <cmath>

//------------------------------------------------------------------------------
// Constructor
//------------------------------------------------------------------------------
M_M_1_S_TwoPointA::M_M_1_S_TwoPointA(unsigned int iSystemSize, unsigned int iThresholdStop,
    unsigned int iThresholdGo)
: CQueueingSystem()
{
    this->iSystemSize = iSystemSize;
    this->iThresholdStop = iThresholdStop;
    this->iThresholdGo = iThresholdGo;

    fArrivalRateNormal = 0;
    fArrivalRateReduced = 0;
    fServiceRate = 1;

    // create the state space
    struct TSystemState state;
```

```
24      vecSystemStates.clear();
25      for (state.n = 0; state.n <= iSystemSize; state.n++)    // number of customers in the ↩
            system
26      {
27          state.s = 0;        // arrival rate normal
28          if (SystemStateIsValid(state))
29              vecSystemStates.push_back(state);
30          state.s = 1;        // arrival rate reduced
31          if (SystemStateIsValid(state))
32              vecSystemStates.push_back(state);
33      }
34
35      // initialize the equation system
36      InitEqSysSystemState(vecSystemStates.size());
37  }
38
39  //-------------------------------------------------------------------------
40  // Set the arrival rates
41  //-------------------------------------------------------------------------
42  void M_M_1_S_TwoPointA::SetArrivalRateNormal(float fArrivalRate)
43  {
44      if (fArrivalRateNormal != fArrivalRate) InvalidateMatrices();
45      fArrivalRateNormal = fArrivalRate;
46  }
47  //-------------------------------------------------------------------------
48  void M_M_1_S_TwoPointA::SetArrivalRateReduced(float fArrivalRate)
49  {
50     if (fArrivalRateReduced != fArrivalRate) InvalidateMatrices();
51      fArrivalRateReduced = fArrivalRate;
52  }
53
54  //-------------------------------------------------------------------------
55  // Set the service rate
56  //-------------------------------------------------------------------------
57  void M_M_1_S_TwoPointA::SetServiceRate(float fServiceRate)
58  {
59      if (this->fServiceRate != fServiceRate) InvalidateMatrices();
60      this->fServiceRate = fServiceRate;
61  }
62
63  //-------------------------------------------------------------------------
64  // Get the number of a given state
65  //-------------------------------------------------------------------------
66  unsigned int M_M_1_S_TwoPointA::GetSystemStateNumber(const TSystemState & state)
67  {
68      for (unsigned int i = 0; i < vecSystemStates.size(); i++)
69          if (vecSystemStates[i].n==state.n && vecSystemStates[i].s==state.s)
70              return i;
71      return (unsigned int)-1;
72  }
73
74  //-------------------------------------------------------------------------
75  // Get the string representation of a state
76  //-------------------------------------------------------------------------
77  const std::string M_M_1_S_TwoPointA::SystemStateToString(unsigned int iStateNumber, char ↩
        cSeperator)
78  {
79      std::stringstream s;
80      s << vecSystemStates[iStateNumber].n << cSeperator << vecSystemStates[iStateNumber].s;
81      return s.str();
82  }
83
84  //-------------------------------------------------------------------------
85  // Calcultate the transition rate between system states
86  //-------------------------------------------------------------------------
87  REAL M_M_1_S_TwoPointA::GetRateSystemState(unsigned int iStateNumberSrc, unsigned int ↩
        iStateNumberDest)
88  {
89      VERIFY(iStateNumberSrc<GetNumberOfSystemStates());
90      VERIFY(iStateNumberDest<GetNumberOfSystemStates());
91
92      TSystemState A = vecSystemStates[iStateNumberSrc];
93      TSystemState B = vecSystemStates[iStateNumberDest];
```

```
94
95      // arrival of a customer
96      if (B.n == A.n+1)
97      {
98          // normal rate, threshold not reached
99          if (A.s==0 && B.s==0 && B.n < iThresholdStop)
100             return fArrivalRateNormal;
101         // normal rate, threshold reached
102         if (A.s==0 && B.s==1 && B.n == iThresholdStop)
103             return fArrivalRateNormal;
104         // reduced rate
105         if (A.s==1 && B.s==1)
106             return fArrivalRateReduced;
107     }
108
109     // service of a customer
110     if (B.n == A.n-1)
111     {
112         // reduced rate, threshold not reached
113         if (A.s==1 && B.s==1 && B.n > iThresholdGo)
114           return fServiceRate;
115         // reduced rate, threshold reached
116         if (A.s==1 && B.s==0 && B.n == iThresholdGo)
117           return fServiceRate;
118         // normal rate
119         if (A.s==0 && B.s==0)
120           return fServiceRate;
121     }
122
123     // there are no other transitions
124     return 0;
125 }
126
127 //---------------------------------------------------------------------------
128 // Check if a system state is valid
129 //---------------------------------------------------------------------------
130 bool M_M_1_S_TwoPointA::SystemStateIsValid(const TSystemState & state)
131 {
132     // ThresholdStop is reached, but the arrival rate is still normal
133     if (state.n>=iThresholdStop && state.s==0) return false;
134     // ThresholdGo is reached, but the arrival rate is still reduced
135     if (state.n<=iThresholdGo && state.s==1) return false;
136     // all other states are valid
137     return true;
138 }
139
140 //---------------------------------------------------------------------------
141 // Get the probability that the system is full (blocking probability)
142 //---------------------------------------------------------------------------
143 float M_M_1_S_TwoPointA::GetBlockingProbability()
144 {
145     float fB=0;
146     for (unsigned int i=0;i<vecSystemStates.size();i++)
147         if (vecSystemStates[i].n==iSystemSize)
148             fB+=GetSystemStateProbability(i);
149     return fB;
150 }
151
152 //---------------------------------------------------------------------------
153 // Get the mean number of customers in the system
154 //---------------------------------------------------------------------------
155 float M_M_1_S_TwoPointA::GetNumberOfCustomersMean()
156 {
157     float fEX=0;
158     for (unsigned int i=0;i<vecSystemStates.size();i++)
159         fEX+= vecSystemStates[i].n*GetSystemStateProbability(i);
160     return fEX;
161 }
162
163 //---------------------------------------------------------------------------
164 // Get the probability that the arrival rate is reduced
165 //---------------------------------------------------------------------------
166 float M_M_1_S_TwoPointA::GetProbabilitySwitchReduced()
```

```
167 {
168     float fProbReduced=0;
169     for (unsigned int i=0;i<vecSystemStates.size();i++)
170         if (vecSystemStates[i].s==1)
171             fProbReduced+= GetSystemStateProbability(i);
172     return fProbReduced;
173 }
```

A.2.3. M/M/1/S-Warteschlangensystem mit geregelter Ankunftsrate und verzögerter Nachrichtenübertragung

Listing A.11: M_M_1_S_TwoPointADelay/M_M_1_S_TwoPointADelay.h

```
1  #ifndef M_M_1_S_TWOPOINTADELAY_H
2  #define M_M_1_S_TWOPOINTADELAY_H M_M_1_S_TWOPOINTADELAY_H
3
4  #include "../QueueingSystem.h"
5
6  #include <vector>
7  #include <map>
8
9  struct TSystemState
10 {
11     unsigned int n;      // number of customers in the system
12     unsigned int s;      // arrival rate: 0 = normal, 1 = reduced
13     unsigned int m;      // number of messages on the way
14 };
15
16 class M_M_1_S_TwoPointADelay : public CQueueingSystem
17 {
18 public:
19     M_M_1_S_TwoPointADelay(unsigned int iSystemSize, unsigned int iThresholdStop, unsigned ↩
          int iThresholdGo, int iMaxMessagesOnTheWay = 10);
20     void SetArrivalRateNormal(float fArrivalRate);
21     void SetArrivalRateReduced(float fArrivalRate);
22     void SetServiceRate(float fServiceRate);
23     void SetControlMessageTransferRate(float fControlMessageTransferRate);
24     void SetThresholdStop(int iThreshold);
25     void SetThresholdGo(int iThreshold);
26     float GetNumberOfCustomersMean();
27     float GetProbabilitySwitchReduced();
28     unsigned int GetSystemSize() {return iSystemSize;};
29     unsigned int GetSystemStateNumber(const TSystemState & state);
30     float GetBlockingProbability();
31     const std::string SystemStateToString(unsigned int iStateNumber);
32 protected:
33     bool SystemStateIsValid(const TSystemState & state);
34     REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest);
35     unsigned int iSystemSize;
36     unsigned int iThresholdStop;
37     unsigned int iThresholdGo;
38     unsigned int iMaxMessagesOnTheWay;
39     float fArrivalRateNormal;
40     float fArrivalRateReduced;
41     float fServiceRate;
42     float fControlMessageTransferRate;
43     std::vector<TSystemState> vecSystemStates;
44 };
45
46 #endif
```

Listing A.12: M_M_1_S_TwoPointADelay/M_M_1_S_TwoPointADelay.cpp

```
1 #include "M_M_1_S_TwoPointADelay.h"
2
3 #include <iostream>
4 #include <iomanip>
5 #include <sstream>
6 #include <cmath>
7
```

```
8  //------------------------------------------------------------------------------
9  // Constructor
10 //------------------------------------------------------------------------------
11 M_M_1_S_TwoPointADelay::M_M_1_S_TwoPointADelay(unsigned int iSystemSize, unsigned int
       iThresholdStop, unsigned int iThresholdGo, int iMaxMessagesOnTheWay)
12 {
13     this->iSystemSize = iSystemSize;
14     this-> iThresholdStop = iThresholdStop;
15     this-> iThresholdGo = iThresholdGo;
16     this->iMaxMessagesOnTheWay=iMaxMessagesOnTheWay;
17     fArrivalRateNormal = 0;
18     fArrivalRateReduced = 0;
19     fServiceRate = 1;
20
21     // create the state space
22     TSystemState state;
23
24     for (state.n = 0; state.n <= iSystemSize; state.n++)    // number of customers in the
           system
25         for (state.m = 0; state.m <= iMaxMessagesOnTheWay; state.m++)    // number of
               messages on the way
26         {
27             state.s = 0;        // arrival rate normal
28             if (SystemStateIsValid(state))
29                 vecSystemStates.push_back(state);
30             state.s = 1;        // arrival rate reduced
31             if (SystemStateIsValid(state))
32                 vecSystemStates.push_back(state);
33         }
34
35     // initialize the equation system
36     InitEqSysSystemState(vecSystemStates.size());
37 }
38
39 //------------------------------------------------------------------------------
40 // Set the arrival rate
41 //------------------------------------------------------------------------------
42 void M_M_1_S_TwoPointADelay::SetArrivalRateNormal(float fArrivalRate)
43 {
44     if (fArrivalRateNormal != fArrivalRate) InvalidateMatrices();
45     fArrivalRateNormal = fArrivalRate;
46 }
47 //------------------------------------------------------------------------------
48 void M_M_1_S_TwoPointADelay::SetArrivalRateReduced(float fArrivalRate)
49 {
50     if (fArrivalRateReduced != fArrivalRate) InvalidateMatrices();
51     fArrivalRateReduced = fArrivalRate;
52 }
53
54 //------------------------------------------------------------------------------
55 // Set the service rate
56 //------------------------------------------------------------------------------
57 void M_M_1_S_TwoPointADelay::SetServiceRate(float fServiceRate)
58 {
59     if (this->fServiceRate != fServiceRate) InvalidateMatrices();
60     this->fServiceRate = fServiceRate;
61 }
62
63 //------------------------------------------------------------------------------
64 // Set the transmission rate of the control messages
65 //------------------------------------------------------------------------------
66 void M_M_1_S_TwoPointADelay::SetControlMessageTransferRate(float fControlMessageTransferRate)
67 {
68     if (this->fControlMessageTransferRate != fControlMessageTransferRate) InvalidateMatrices
           ();
69     this->fControlMessageTransferRate = fControlMessageTransferRate;
70 }
71
72 //------------------------------------------------------------------------------
73 // Get the number of a given state
74 //------------------------------------------------------------------------------
75 unsigned int M_M_1_S_TwoPointADelay::GetSystemStateNumber(const TSystemState & state)
76 {
```

```
77      for (unsigned int i = 0; i < vecSystemStates.size(); i++)
78          if (vecSystemStates[i].n==state.n && vecSystemStates[i].s==state.s && vecSystemStates
                [i].m==state.m)
79              return i;
80      return (unsigned int)-1;
81  }
82
83  //---------------------------------------------------------------------------
84  // Get the string representation of a state
85  //---------------------------------------------------------------------------
86  const std::string M_M_1_S_TwoPointADelay::SystemStateToString(unsigned int iStateNumber)
87  {
88      #define SEPARATOR '/'
89      std::stringstream s;
90      s << vecSystemStates[iStateNumber].n << SEPARATOR << vecSystemStates[iStateNumber].s <<
            SEPARATOR << vecSystemStates[iStateNumber].m;
91      return s.str();
92  }
93
94  //---------------------------------------------------------------------------
95  // Calcultate the transition rate between system states
96  //---------------------------------------------------------------------------
97  REAL M_M_1_S_TwoPointADelay::GetRateSystemState(unsigned int iStateNumberSrc, unsigned int
        iStateNumberDest)
98  {
99
100     VERIFY(iStateNumberSrc<iNumberOfSystemStates);
101     VERIFY(iStateNumberDest<iNumberOfSystemStates);
102
103     TSystemState A = vecSystemStates[iStateNumberSrc];
104     TSystemState B = vecSystemStates[iStateNumberDest];
105
106     // arrival of a customer
107     if (B.n == A.n+1 && B.s==A.s)
108     {
109         if ((B.n!=iThresholdStop && B.m==A.m) ||
110             (B.n==iThresholdStop && B.m==A.m && (A.s+A.m)%2==1) ||   // threshold stop
                    reached, message already sent
111             (B.n==iThresholdStop && B.m==A.m+1 && (A.s+A.m)%2==0) ||      // threshold stop
                    reached, sending message
112             (B.n==iThresholdStop && B.m==A.m && A.m==iMaxMessagesOnTheWay && (A.s+A.m)%2==0))
                    // --"--, message buffer full
113             // if A.s+A.m is even, then the last message that has been sent was "go".
114             // we send "stop"-messages only in this case
115             if (B.s==0)
116                 return fArrivalRateNormal;
117             else
118                 return fArrivalRateReduced;
119     }
120
121     // service of a customer
122     else if (B.n == A.n-1 && B.s==A.s)
123     {
124         if ((B.n!=iThresholdGo && B.m==A.m) ||
125             (B.n==iThresholdGo && B.m==A.m && (A.s+A.m)%2==0) ||    // threshold go reached,
                    message already sent
126             (B.n==iThresholdGo && B.m==A.m+1 && (A.s+A.m)%2==1) ||        // threshold go
                    reached, sending message
127             (B.n==iThresholdGo && B.m==A.m && A.m==iMaxMessagesOnTheWay && (A.s+A.m)%2==1))
                    // --"--, message buffer full
128             // if A.s+A.m is odd, then the last message that has been sent was "stop".
129             // we send "go"-messages only in this case
130             return fServiceRate;
131     }
132     // arrival of a message at the source
133     else if (B.n == A.n && B.s==1-A.s && B.m == A.m-1 )
134     {
135         return fControlMessageTransferRate;
136     }
137
138     // there are no other transitions
139     return 0;
140 }
```

```
141
142 //------------------------------------------------------------------------------
143 // Check if a system state is valid
144 //------------------------------------------------------------------------------
145 bool M_M_1_S_TwoPointADelay::SystemStateIsValid(const TSystemState & state)
146 {
147     VERIFY (state.n<=iSystemSize);
148     VERIFY (state.m<=iMaxMessagesOnTheWay);
149     VERIFY (state.s<=1);
150
151     // ThresholdStop is reached, but the last message sent was "go"
152     //if (state.n>=iThresholdStop && (state.s+state.m)%2==0) return false;
153     // ThresholdGo is reached, but the last message sent was "stop"
154     //if (state.n<=iThresholdGo && (state.s+state.m)%2==1) return false;
155     // all other states are valid
156
157     // if the reduced arrival rate is not 0, all states are valid
158     return true;
159 }
160
161 //------------------------------------------------------------------------------
162 // Get the probability that the system is full (blocking probability)
163 //------------------------------------------------------------------------------
164 float M_M_1_S_TwoPointADelay::GetBlockingProbability()
165 {
166     float fB=0;
167     for (unsigned int i=0;i<vecSystemStates.size();i++)
168         if (vecSystemStates[i].n==iSystemSize)
169             fB+=GetSystemStateProbability(i);
170     return fB;
171 }
172
173 //------------------------------------------------------------------------------
174 // Get the mean number of customers in the system
175 //------------------------------------------------------------------------------
176 float M_M_1_S_TwoPointADelay::GetNumberOfCustomersMean()
177 {
178     REAL fEX=0;
179     for (unsigned int i=0;i<vecSystemStates.size();i++)
180         fEX+= vecSystemStates[i].n*GetSystemStateProbability(i);
181     return fEX;
182 }
183
184 //------------------------------------------------------------------------------
185 // Get the probability that the arrival rate is reduced
186 //------------------------------------------------------------------------------
187 float M_M_1_S_TwoPointADelay::GetProbabilitySwitchReduced()
188 {
189     float fProbReduced=0;
190     for (unsigned int i=0;i<vecSystemStates.size();i++)
191         if (vecSystemStates[i].s==1)
192             fProbReduced+= GetSystemStateProbability(i);
193     return fProbReduced;
194 }
```

A.2.4. M/M/1/S-Warteschlangensystem mit geregelter Bedienrate

Listing A.13: M_M_1_S_OnePointS/M_M_1_S_OnePointS.h

```
1  #ifndef M_M_1_S_ONEPOINTS_H
2  #define M_M_1_S_ONEPOINTS_H M_M_1_S_ONEPOINTS_H
3
4  #include "../QueueingSystem.h"
5
6  struct TSystemState
7  {
8      unsigned int n;      // number of customers in the system
9  };
10
11 struct TFlowState
12 {
```

```
13      unsigned int n;      // number of customers in the system
14      unsigned int m;      // number of customers which are served before the test customer ist
            served
15  };
16
17  class M_M_1_S_OnePointS : public CQueueingSystem
18  {
19  public:
20      M_M_1_S_OnePointS(unsigned int iSystemSize, unsigned int iThreshold);
21      void SetArrivalRate(float fArrivalRate);
22      void SetServiceRateNormal(float fServiceRate);
23      void SetServiceRateReduced(float fServiceRate);
24      float GetArrivalRate() {return fArrivalRate;};
25      float GetServiceRateNormal() {return fServiceRateNormal;};
26      float GetServiceRateReduced() {return fServiceRateReduced;};
27      unsigned int GetSystemSize() {return iSystemSize;};
28      float GetNumberOfCustomersMean();
29      float GetProbabilityServiceRateReduced();
30      float GetBlockingProbability();
31      unsigned int GetSystemStateNumber(const TSystemState & state);
32      unsigned int GetFlowStateNumber(const TFlowState & state);
33      const std::string SystemStateToString(unsigned int iStateNumber, char cSeperator='-');
34      const std::string FlowStateToString(unsigned int iStateNumber, char cSeperator='-');
35      void Debug_PrintSystemStates();
36      void Debug_PrintFlowStates();
37  protected:
38      std::vector<TSystemState> vecSystemStates;
39      std::vector<TFlowState> vecFlowStates;
40      bool SystemStateIsValid(const TSystemState & state);
41      bool FlowStateIsValid(const TFlowState & state);
42      REAL GetRateSystemState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest);
43      REAL GetRateFlowState(unsigned int iStateNumberSrc, unsigned int iStateNumberDest);
44      unsigned int iSystemSize;
45      unsigned int iThreshold;
46      float fServiceRateNormal;
47      float fServiceRateReduced;
48      float fArrivalRate;
49  };
50
51  #endif
```

Listing A.14: M_M_1_S_OnePointS/M_M_1_S_OnePointS.cpp

```
1
2   #include "M_M_1_S_OnePointS.h"
3
4   #include <iostream>
5   #include <iomanip>
6   #include <sstream>
7   #include <cmath>
8
9   //--------------------------------------------------------------------------
10  // Constructor
11  //--------------------------------------------------------------------------
12  M_M_1_S_OnePointS::M_M_1_S_OnePointS(unsigned int iSystemSize, unsigned int iThreshold)
13  : CQueueingSystem()
14  {
15      this->iSystemSize = iSystemSize;
16      this->iThreshold = iThreshold;
17
18      fServiceRateNormal = 1;
19      fServiceRateReduced = 1;
20      fArrivalRate = 0;
21
22      // create the space of the system states
23      TSystemState state;
24
25      for (state.n = 0; state.n <= iSystemSize; state.n++)
26          if (SystemStateIsValid(state))
27              vecSystemStates.push_back(state);
28
29      // create the space of the flow states
30      TFlowState state2;
```

```
31
32      for (state2.n = 0; state2.n <= iSystemSize; state2.n++)
33          for (state2.m = 0; state2.m <= iSystemSize; state2.m++)
34              if (FlowStateIsValid(state2))
35                  vecFlowStates.push_back(state2);
36
37      Debug_PrintSystemStates();
38      Debug_PrintFlowStates();
39
40      // initialize the equation systems
41      InitEqSysSystemState(vecSystemStates.size());
42      InitEqSysFlowTime(vecFlowStates.size());
43
44
45      // for each system state S find the corresponding flow state
46      // (this is the flow state which is reached when the test customer arrives,
47      // when the system is in state S)
48      for (unsigned int i = 0; i<vecSystemStates.size(); i++)
49      {
50          state = vecSystemStates[i];
51          state2.n = state.n+1;  // the arrival of the test customer increases the number of
                customers by 1
52          state2.m = state.n;    // the customers which are in the system will be served before
                 the test customer is served
53          if (FlowStateIsValid(state2))
54              mapCorrespondingStates.insert(std::pair<unsigned int,unsigned int>(i,
                    GetFlowStateNumber(state2)));
55      }
56  }
57
58  //---------------------------------------------------------------------------
59  // Set the service rates
60  //---------------------------------------------------------------------------
61  void M_M_1_S_OnePointS::SetServiceRateNormal(float fServiceRate)
62  {
63      if (fServiceRateNormal != fServiceRate) InvalidateMatrices();
64      fServiceRateNormal = fServiceRate;
65  }
66  //---------------------------------------------------------------------------
67  void M_M_1_S_OnePointS::SetServiceRateReduced(float fServiceRate)
68  {
69     if (fServiceRateReduced != fServiceRate) InvalidateMatrices();
70      fServiceRateReduced = fServiceRate;
71  }
72
73  //---------------------------------------------------------------------------
74  // Set the arrival rates
75  //---------------------------------------------------------------------------
76  void M_M_1_S_OnePointS::SetArrivalRate(float fArrivalRate)
77  {
78      if (this->fArrivalRate != fArrivalRate) InvalidateMatrices();
79      this->fArrivalRate = fArrivalRate;
80  }
81
82  //---------------------------------------------------------------------------
83  // Get the number of a system state
84  //---------------------------------------------------------------------------
85  unsigned int M_M_1_S_OnePointS::GetSystemStateNumber(const TSystemState & state)
86  {
87      for (unsigned int i = 0; i < vecSystemStates.size(); i++)
88          if (vecSystemStates[i].n==state.n)
89              return i;
90      return (unsigned int)-1;
91  }
92
93  //---------------------------------------------------------------------------
94  // Get the number of a flow state
95  //---------------------------------------------------------------------------
96  unsigned int M_M_1_S_OnePointS::GetFlowStateNumber(const TFlowState & state)
97  {
98      for (unsigned int i = 0; i < vecFlowStates.size(); i++)
99          if (vecFlowStates[i].n==state.n && vecFlowStates[i].m==state.m)
100             return i;
```

```
101     return (unsigned int)-1;
102 }
103 
104 //------------------------------------------------------------------------------
105 // Get the string representation of a system state
106 //------------------------------------------------------------------------------
107 const std::string M_M_1_S_OnePointS::SystemStateToString(unsigned int iStateNumber, char ↩
        cSeperator)
108 {
109     std::stringstream s;
110     s << vecSystemStates[iStateNumber].n;
111     return s.str();
112 }
113 
114 //------------------------------------------------------------------------------
115 // Get the string representation of a flow state
116 //------------------------------------------------------------------------------
117 const std::string M_M_1_S_OnePointS::FlowStateToString(unsigned int iStateNumber, char ↩
        cSeperator)
118 {
119     std::stringstream s;
120     s << vecFlowStates[iStateNumber].n << cSeperator << vecFlowStates[iStateNumber].m;
121     return s.str();
122 }
123 
124 //------------------------------------------------------------------------------
125 // Get the transition rate between two system states
126 //------------------------------------------------------------------------------
127 REAL M_M_1_S_OnePointS::GetRateSystemState(unsigned int iStateNumberSrc, unsigned int ↩
        iStateNumberDest)
128 {
129     VERIFY(iStateNumberSrc<GetNumberOfSystemStates());
130     VERIFY(iStateNumberDest<GetNumberOfSystemStates());
131 
132     TSystemState A = vecSystemStates[iStateNumberSrc];
133     TSystemState B = vecSystemStates[iStateNumberDest];
134 
135     // arrival of a customer
136     if (B.n == A.n+1)
137         return fArrivalRate;
138 
139     // service of a customer
140     if (B.n == A.n-1)
141         if (A.n>iThreshold)
142                 return fServiceRateNormal;
143             else
144                 return fServiceRateReduced;
145 
146     // there are no other state transitions
147     return 0;
148 }
149 
150 //------------------------------------------------------------------------------
151 // Get the transition rate between two flow states
152 //------------------------------------------------------------------------------
153 REAL M_M_1_S_OnePointS::GetRateFlowState(unsigned int iStateNumberSrc, unsigned int ↩
        iStateNumberDest)
154 {
155     //PRINTTEXT("GetRateFlowState");
156     //PRINTVAR(iStateNumberSrc);
157     //PRINTVAR(iStateNumberDest);
158 
159     VERIFY(iStateNumberSrc<GetNumberOfFlowStates());
160     TFlowState A = vecFlowStates[iStateNumberSrc];
161 
162     if (iStateNumberDest!=(unsigned int)-1)
163     {
164         VERIFY(iStateNumberDest<GetNumberOfFlowStates());
165         TFlowState B = vecFlowStates[iStateNumberDest];
166 
167         // arrival of a customer
168         if (B.n == A.n+1 && B.m == A.m)
169             return fArrivalRate;
```

```
170
171         // service of a customer
172         if (B.n == A.n-1 && B.m == A.m-1)
173         {
174             if (A.n>iThreshold)
175                 return fServiceRateNormal;
176             else
177                 return fServiceRateReduced;
178         }
179
180         // there are no other state transitions
181         return 0;
182     }
183     else        // transition to an absorbing state
184     {
185         // service of a customer
186         if (A.m == 0)
187         {
188             if (A.n>iThreshold)
189                 return fServiceRateNormal;
190             else
191                 return fServiceRateReduced;
192         }
193         return 0;
194     }
195 }
196
197 //---------------------------------------------------------------------------
198 // Check if a system state is valid
199 //---------------------------------------------------------------------------
200 bool M_M_1_S_OnePointS::SystemStateIsValid(const TSystemState & state)
201 {
202     return (state.n <= iSystemSize);
203 }
204
205 //---------------------------------------------------------------------------
206 // Check if a flow state is valid
207 //---------------------------------------------------------------------------
208 bool M_M_1_S_OnePointS::FlowStateIsValid(const TFlowState & state)
209 {
210     return (state.n <= iSystemSize && state.m < state.n);
211 }
212
213 //---------------------------------------------------------------------------
214 // Get the probability that the system is full (blocking probability)
215 //---------------------------------------------------------------------------
216 float M_M_1_S_OnePointS::GetBlockingProbability()
217 {
218     for (unsigned int i=0;i<vecSystemStates.size();i++)
219         if (vecSystemStates[i].n==iSystemSize)
220             return GetSystemStateProbability(i);
221     return 0;
222 }
223
224 //---------------------------------------------------------------------------
225 // Get the mean number of customers in the system
226 //---------------------------------------------------------------------------
227 float M_M_1_S_OnePointS::GetNumberOfCustomersMean()
228 {
229     float fEX=0;
230     for (unsigned int i=0;i<vecSystemStates.size();i++)
231         fEX+= vecSystemStates[i].n*GetSystemStateProbability(i);
232     return fEX;
233 }
234
235 //---------------------------------------------------------------------------
236 // Get the probability, that the service rate is increased
237 //---------------------------------------------------------------------------
238 float M_M_1_S_OnePointS::GetProbabilityServiceRateReduced()
239 {
240     float fProbReduced=0;
241     for (unsigned int i=0;i<vecSystemStates.size();i++)
242         if (vecSystemStates[i].n<=iThreshold)
```

```
243             fProbReduced+= GetSystemStateProbability(i);
244     return fProbReduced;
245 }
246 
247 //--------------------------------------------------------------------------
248 // Print the system states
249 //--------------------------------------------------------------------------
250 void M_M_1_S_OnePointS::Debug_PrintSystemStates()
251 {
252     std::cout << "# of states: " << vecSystemStates.size() << std::endl;
253     for (int i=0; i<vecSystemStates.size(); i++)
254         std::cout << SystemStateToString(i,'/') << " ";
255     std::cout << std::endl;
256 }
257 
258 //--------------------------------------------------------------------------
259 // Print the flow states
260 //--------------------------------------------------------------------------
261 void M_M_1_S_OnePointS::Debug_PrintFlowStates()
262 {
263     std::cout << "# of states: " << vecFlowStates.size() << std::endl;
264     for (int i=0; i<vecFlowStates.size(); i++)
265         std::cout << FlowStateToString(i,'/') << " ";
266     std::cout << std::endl;
267 }
```

A.3. Mathematik

A.3.1. Speichern von dünnbesetzten Matrizen

Die Klasse *CMatrix* wird zur Speicherung von dünnbesetzten Matrizen verwendet. Statt alle Elemente der Matrix in einem zweidimensionalen Array zu speichern, werden nur diejenigen Elemente gespeichert, deren Wert ungleich 0 ist. Das Speichern erfolgt zeilenweise in verketteten Listen, wodurch auch eine schnelle Matrix-Vektor-Multiplikation möglich ist. (Diese wird beispielsweise für das Runge-Kutta-Verfahren und das Gauß-Seidel-Relaxationsverfahren benötigt.)

Listing A.15: Matrix.h

```
1  #ifndef MATRIX_H
2  #define MATRIX_H MATRIX_H
3  
4  #include "../config.h"
5  #include "../include.h"
6  
7  #include <vector>
8  #include <list>
9  
10 /* class for the storage of sparse matrices */
11 
12 struct TMatrixElement
13 {
14     unsigned int iCol;      // the column of the element, starts with 0
15     REAL fValue;            // the value of the element
16 };
17 
18 typedef std::list<TMatrixElement> TMatrixRow ;
19 
20 class CMatrix
21 {
22     public:
23         CMatrix();
24 
25         // Set an element of the matrix
26         void Set(int iRow, int iCol, REAL fValue);
27 
```

```
28          // Get an element of the matrix
29          REAL Get(int iRow, int iCol);
30
31
32          // Clear the matrix
33          void Clear();
34
35          // Input and Output of the matrix
36          void ReadMatrixFromFile(char * szFilename);
37          void WriteMatrixToFile(char *szFilename);
38          void PrintMatrix(bool bCompressed = false);
39          void PrintStatistics();
40
41          // Get the size of the matrix
42          void GetSize(int & iRows, int & iCols);
43
44          // the rows of the matrix
45          std::vector<TMatrixRow> vecMatrixRows;
46
47      private:
48          // the number of columns of the matrix
49          int iMatrixCols;
50  };
51
52  #endif
```

Listing A.16: Matrix.cpp

```
1   #include "Matrix.h"
2
3   #include <iomanip>
4   #include <sstream>
5   #include <fstream>
6
7   #define MAX_SIZE 100000
8
9   //---------------------------------------------------------------------------
10  // Constructor
11  //---------------------------------------------------------------------------
12  CMatrix::CMatrix()
13  {
14      this->iMatrixCols = 0;
15  }
16
17  //---------------------------------------------------------------------------
18  // Set an element of the matrix
19  //---------------------------------------------------------------------------
20  void CMatrix::Set(int iRow, int iCol, REAL fValue)
21  {
22      // dynamic growing
23      if (iCol>=iMatrixCols) iMatrixCols=iCol+1;
24      if (iRow >= MAX_SIZE || iCol >= MAX_SIZE) ERROR("The matrix is too large.");
25      while (iRow>=vecMatrixRows.size())
26      {
27          TMatrixRow r;
28          vecMatrixRows.push_back(r);
29      }
30
31      // find the correct position and set the element
32      TMatrixRow::iterator iterMatrixRow = vecMatrixRows[iRow].begin();
33      while (iterMatrixRow != vecMatrixRows[iRow].end())
34      {
35          // insert a new element
36          if (iterMatrixRow->iCol > iCol)
37          {
38              if (fValue != 0)
39              {
40                  TMatrixElement e;
41                  e.iCol = iCol;
42                  e.fValue = fValue;
43                  vecMatrixRows[iRow].insert(iterMatrixRow, e);
44              }
45              return;
```

```
46          }
47          else
48          // change an element which is already there
49          if (iterMatrixRow->iCol == iCol)
50          {
51              if (fValue != 0)
52                  iterMatrixRow->fValue = fValue;
53              else
54                  vecMatrixRows[iRow].erase(iterMatrixRow);
55              return;
56          }
57          else
58          iterMatrixRow++;
59      }
60      // insert a new element at the end of the row
61      if (fValue != 0)
62      {
63          TMatrixElement e;
64          e.iCol = iCol;
65          e.fValue = fValue;
66          vecMatrixRows[iRow].push_back(e);
67      }
68  }
69
70  //-------------------------------------------------------------------------
71  // Get an element of the matrix
72  //-------------------------------------------------------------------------
73  REAL CMatrix::Get(int iRow, int iCol)
74  {
75      VERIFY(iRow<vecMatrixRows.size());
76      VERIFY(iCol<iMatrixCols);
77      TMatrixRow::iterator iterMatrixRow = vecMatrixRows[iRow].begin();
78      while (iterMatrixRow != vecMatrixRows[iRow].end())
79      {
80          if (iterMatrixRow->iCol==iCol) return iterMatrixRow->fValue;
81          iterMatrixRow++;
82      }
83      return 0;
84  }
85
86  //-------------------------------------------------------------------------
87  // Output of the matrix
88  //-------------------------------------------------------------------------
89  void CMatrix::PrintMatrix(bool bCompressed)
90  {
91      if (vecMatrixRows.size()==0)
92      {
93          std::cout << "The␣matrix␣is␣empty!" << std::endl;
94          return;
95      }
96      // compressed output
97      if (bCompressed)
98      {
99          for (unsigned int iRow=0; iRow<vecMatrixRows.size(); iRow++)
100         {
101             std::cout << "R" << iRow << "␣";
102             TMatrixRow::iterator iterMatrixRow = vecMatrixRows[iRow].begin();
103             while (iterMatrixRow != vecMatrixRows[iRow].end())
104             {
105                 std::cout << "C" << iterMatrixRow->iCol << "=" << iterMatrixRow->fValue << "␣"
                        ";
106                 iterMatrixRow++;
107             }
108             std::cout << std::endl;
109         }
110     }
111     else
112     // normal output
113     {
114         const int COLWIDTH = 10;
115         for (unsigned int iRow=0; iRow<vecMatrixRows.size(); iRow++)
116         {
117             TMatrixRow::iterator iterMatrixRow = vecMatrixRows[iRow].begin();
```

```
118             for (unsigned int iCol=0; iCol<iMatrixCols; iCol++)
119                 if ((iterMatrixRow != vecMatrixRows[iRow].end()) && (iterMatrixRow->iCol==
                        iCol))
120                     std::cout << std::setw(COLWIDTH) << iterMatrixRow++->fValue;
121                 else
122                     std::cout  << std::setw(COLWIDTH) << 0;
123             std::cout << std::endl;
124         }
125     }
126 }
127
128 //---------------------------------------------------------------------------
129 // Write the matrix to a file
130 //---------------------------------------------------------------------------
131
132 void CMatrix::WriteMatrixToFile(char * szFileName)
133 {
134     const int COLWIDTH = 10;
135     std::ofstream outfile;
136     outfile.open(szFileName,std::ios::out);
137     for (unsigned int iRow=0; iRow<vecMatrixRows.size(); iRow++)
138     {
139         TMatrixRow::iterator iterMatrixRow = vecMatrixRows[iRow].begin();
140         for (unsigned int iCol=0; iCol<iMatrixCols; iCol++)
141             if ((iterMatrixRow != vecMatrixRows[iRow].end()) && (iterMatrixRow->iCol==iCol))
142                 outfile << std::setw(COLWIDTH) << iterMatrixRow++->fValue;
143             else
144                 outfile  << std::setw(COLWIDTH) << 0;
145         outfile << std::endl;
146     }
147     outfile.close();
148 }
149
150 //---------------------------------------------------------------------------
151 // Clear the matrix
152 //---------------------------------------------------------------------------
153 void CMatrix::Clear()
154 {
155     for (int i=0; i<vecMatrixRows.size(); vecMatrixRows[i++].clear());
156     vecMatrixRows.clear();
157     this->iMatrixCols = 0;
158 }
159
160 //---------------------------------------------------------------------------
161 // Get the size of the matrix
162 //---------------------------------------------------------------------------
163 void CMatrix::GetSize(int & iRows, int & iCols)
164 {
165     iRows = vecMatrixRows.size();
166     iCols = this->iMatrixCols;
167 }
168
169 //---------------------------------------------------------------------------
170 // Read the matrix from a file (columns separated by blanks)
171 //---------------------------------------------------------------------------
172 void CMatrix::ReadMatrixFromFile(char * szFileName)
173 {
174     std::ifstream infile;
175     infile.open(szFileName,std::ios::in);
176     Clear();
177
178     char c;
179     int iRow = 0;
180     int iCol = -1;
181     REAL fValue;
182
183     while (!infile.eof())
184     {
185         while (std::isspace(c=infile.get()) && c != '\n');
186         if ((c=='\n') && (iCol>=0))
187         {
188             iRow++;
189             iCol=-1;
```

```
190         }
191         else
192         {
193             infile.putback(c);
194             infile >> fValue;
195             iCol++;
196             Set(iRow,iCol,fValue);
197         }
198     }
199     infile.close();
200 }
201
202 //--------------------------------------------------------------------------
203 // Output of statistics about the matrix
204 //--------------------------------------------------------------------------
205 void CMatrix::PrintStatistics()
206 {
207     unsigned int iSum=0;
208     for (unsigned int iRow=0; iRow<vecMatrixRows.size(); iSum +=  vecMatrixRows[iRow++].size
        ());
209     std::cout << "#rows = " << vecMatrixRows.size() << "   #cols = " << iMatrixCols <<
210         "   #elements = " << vecMatrixRows.size() * iMatrixCols * 1.0 <<
211         "   #non-zero elements = " << iSum * 1.0 <<
212         "   fullness = " << 100.0*iSum/vecMatrixRows.size()/iMatrixCols << "%"  <<
213         std::endl;
214 }
```

A.3.2. Lösen von linearen Gleichungssystemen

Die Klasse *CLEQSolver* (Listings A.17 und A.18) dient als Basisklasse für verschiedene Löser von linearen Gleichungssystemen. Die Klassen *CLEQSolverSOR* (Listings A.19 und A.20), *CLEQSolverRecursive* (Listings A.21 und A.22) und *CLEQSolverLU_Lapack* (Listings A.23 und A.24) lösen lineare Gleichungssysteme mit dem Gauß-Seidel-Relaxationsverfahren, der rekursiven Methode beziehungsweise durch LU-Zerlegung. Die Klasse *CLEQSolverLU_Lapack* benötigt die LAPACK-Bibliothek.

Listing A.17: LEQSolver.h

```
1  #ifndef LEQSOLVER_H
2  #define LEQSOLVER_H LEQSOLVER_H
3
4  #include "Matrix.h"
5
6  /* base class for solver of linear equation systems */
7  class CLEQSolver
8  {
9      public:
10         CLEQSolver(int iSystemSize, CMatrix mCoeffMatrix, REAL * aRHS, REAL * aSolution);
11         virtual ~CLEQSolver();
12         virtual bool Solve() = 0;
13     protected:
14         int iSystemSize;
15         CMatrix mCoeffMatrix;
16         REAL * aRHS;
17         REAL * aSolution;
18         bool xNullArray;
19 };
20 #endif
```

Listing A.18: LEQSolver.cpp

```
1  #include "LEQSolver.h"
2
3  //--------------------------------------------------------------------------
4  // Constructor
5  //--------------------------------------------------------------------------
```

```
6  CLEQSolver::CLEQSolver(int iSystemSize, CMatrix mCoeffMatrix, REAL * aRHS, REAL * aSolution)
7  {
8      this->iSystemSize = iSystemSize;
9      this->mCoeffMatrix = mCoeffMatrix;
10     if (aRHS!=NULL)
11     {
12         this->aRHS = aRHS;
13         xNullArray = false;
14     }
15     else
16     {
17         this->aRHS = new REAL[iSystemSize];
18         for (int i=0; i<iSystemSize; this->aRHS[i++]=0);
19         xNullArray = true;
20     }
21     this->aSolution = aSolution;
22 }
23
24 //--------------------------------------------------------------------------
25 // Destructor
26 //--------------------------------------------------------------------------
27 CLEQSolver::~CLEQSolver()
28 {
29     if (xNullArray)
30         delete[] aRHS;
31 }
```

Listing A.19: LEQSolverSOR.h

```
1  #ifndef LEQSOLVERSOR_H
2  #define LEQSOLVERSOR_H LEQSOLVERSOR_H
3
4  #include "LEQSolver.h"
5
6  #include <vector>
7  #include <list>
8
9  /* class for solving linear equation systems with SOR */
10
11 class CLEQSolverSOR : CLEQSolver
12 {
13     public:
14         CLEQSolverSOR(int iSystemSize, CMatrix mCoeffMatrix, REAL * aRHS, REAL * aSolution);
15         void SetParamRelaxationParameter(float fRelaxationParameter);
16         void SetParamEps(float fEps) {this->fEps = fEps;};
17         void SetParamMaxIterations(int iMaxIterations) {this->iMaxIterations = iMaxIterations
               ;};
18         bool Solve();
19         bool Solve(REAL & fResidual, int & iNumberOfIterations);
20         float FindBestRelaxationParameter(float fMin, float fMax, float fStepsize, int
               iNumberOfIterations);
21     private:
22         float fRelaxationParameter;
23         float fEps;
24         int iMaxIterations;
25 };
26
27 #endif
```

Listing A.20: LEQSolverSOR.cpp

```
1  #include "LEQSolverSOR.h"
2
3  #include <iomanip>
4  #include <sstream>
5  #include <fstream>
6  #include <cmath>
7  #include <algorithm>
8  #include <ctime>
9
10
11 // #define VERB_SOR_FINDBEST_01
12 #define VERB_SOR_SOLVE_01   // error messages
```

```
13 #define VERB_SOR_SOLVE_02   // info messages
14 //#define VERB_SOR_SOLVE_03   // debug messages
15
16 //-----------------------------------------------------------------------------
17 // Construcor
18 //-----------------------------------------------------------------------------
19 CLEQSolverSOR::CLEQSolverSOR(int iSystemSize, CMatrix mCoeffMatrix,
20     REAL * aRHS, REAL * aSolution)
21 : CLEQSolver(iSystemSize, mCoeffMatrix, aRHS, aSolution)
22 {
23     fRelaxationParameter = 1.0;
24     fEps = 1e-6;
25     iMaxIterations = 1000;
26 }
27
28 //-----------------------------------------------------------------------------
29 // Set the relaxation parameter
30 //-----------------------------------------------------------------------------
31 void CLEQSolverSOR::SetParamRelaxationParameter(float fRelaxationParameter)
32 {
33     if (fRelaxationParameter <= 0 || fRelaxationParameter >= 2)
34         ERROR("SOR: The relaxation parameter must be in (0,2)");
35     this->fRelaxationParameter = fRelaxationParameter;
36 }
37
38 //-----------------------------------------------------------------------------
39 // Solve
40 //-----------------------------------------------------------------------------
41 bool CLEQSolverSOR::Solve(REAL & fResidual, int & iNumberOfIterations)
42 {
43     //PRINTVAR(fRelaxationParameter);
44
45     #ifdef VERB_SOR_SOLVE_02
46         std::cout << "CLEQSolverSOR::Solve(...)" << std::endl;
47     #endif
48
49     #ifdef VERB_SOR_SOLVE_03
50         PRINTVAR(iSystemSize);
51
52         std::cout << "RHS = ";
53         for (int i=0; i<iSystemSize; i++)
54             std::cout << aRHS[i] << "  ";
55         std::cout << std::endl;
56
57         std::cout << "Startvector = ";
58         for (int i=0; i<iSystemSize; i++)
59             std::cout << aSolution[i] << "  ";
60         std::cout << std::endl;
61
62         std::cout << "Matrix = " << std::endl;
63         mCoeffMatrix.PrintMatrix();
64     #endif
65
66     int m,n;
67     mCoeffMatrix.GetSize(m,n);
68     VERIFY(iSystemSize == m && iSystemSize == n);
69
70     bool xNewIteration;
71     int iResCounter = 0;
72     iNumberOfIterations = 0;
73     REAL fResOld = -1;
74     do
75     {
76         if (iNumberOfIterations >= iMaxIterations)
77         {
78             #ifdef VERB_SOR_SOLVE_01
79                 std::cout << "SOR: aborted - too many iterations" << std::endl;
80             #endif
81             return false;
82         }
83         iNumberOfIterations++;
84         fResidual=0;
85         xNewIteration = false;
```

```
86
87          for (unsigned int iRow = 0; iRow < iSystemSize; iRow++)
88          {
89              REAL aii=0;
90              REAL fSum=0;
91
92              std::list<TMatrixElement>::const_iterator iterMatrixRow = mCoeffMatrix.
                    vecMatrixRows[iRow].begin();
93              while (iterMatrixRow != mCoeffMatrix.vecMatrixRows[iRow].end())
94              {
95                  if (iterMatrixRow->iCol!=iRow)
96                      fSum += aSolution[iterMatrixRow->iCol]*iterMatrixRow->fValue;
97                  else
98                      aii = iterMatrixRow->fValue;
99                  iterMatrixRow++;
100             }
101             if (aii==0)
102             {
103                 #ifdef VERB_SOR_SOLVE_01
104                     std::cout << "SOR: aborted - division by zero" << std::endl;
105                 #endif
106                 return false;
107             }
108             REAL U = aSolution[iRow] * (1 - fRelaxationParameter) + fRelaxationParameter /
                    aii * (aRHS[iRow]-fSum);
109             fResidual += std::fabs(aRHS[iRow]-fSum-aSolution[iRow]*aii);
110             if (U==0)
111             {
112                 #ifdef VERB_SOR_SOLVE_01
113                     std::cout << "SOR: aborted - division by zero" << std::endl;
114                 #endif
115                 return false;
116             }
117             xNewIteration |= fResidual > fEps;
118             aSolution[iRow] = U;
119         }
120         // std::cout << std::setw(13) << fResidual << " ";  // TEST_03
121
122         if (fResidual>fResOld)
123         {
124             iResCounter++;
125             //#define MAX_RES_GROW 50     // TEST_04
126             #define MAX_RES_GROW 10
127             if (iResCounter>MAX_RES_GROW)
128             {
129                 #ifdef VERB_SOR_SOLVE_01
130                     std::cout << "SOR: aborted - the residual has been growing too long" <<
                            std::endl;
131                 #endif
132                 return false;
133             }
134         }
135         else
136         if (iResCounter>0)
137         {
138             iResCounter--;
139         }
140 //        PRINTVAR(fRes/fResOld);
141         fResOld = fResidual;
142         // PRINTVAR(fResidual);
143
144     }
145     while (xNewIteration);
146     #ifdef VERB_SOR_SOLVE_01
147         std::cout << "SOR: successfully finished - number of iterations: " <<
148             iNumberOfIterations << ", residual: " << fResidual << std::endl;
149     #endif
150
151     #ifdef VERB_SOR_SOLVE_03
152         std::cout << "Solution = ";
153         for (int i=0; i<iSystemSize; i++)
154             std::cout << aSolution[i] << "  ";
155         std::cout << std::endl;
```

```
156     #endif
157
158     return true;
159 }
160
161 //----------------------------------------------------------------------------
162 // Solve
163 //----------------------------------------------------------------------------
164 bool CLEQSolverSOR::Solve(void)
165 {
166     int iNumberOfIterations;
167     REAL fRes;
168     return Solve(fRes, iNumberOfIterations);
169 }
170
171 //----------------------------------------------------------------------------
172 // Find the best relaxation parameter
173 //----------------------------------------------------------------------------
174 float CLEQSolverSOR::FindBestRelaxationParameter(float fMin, float fMax,
175     float fStepsize, int iNumberOfIterations)
176 {
177     #ifdef VERB_SOR_FINDBEST_01
178         std::cout << "CLEQSolverSOR::FindBestRelaxationParameter(...)" << std::endl;
179     #endif
180
181     // save the current values
182     int iOldMaxIterations = iMaxIterations;
183     float fOldRelaxationParameter = fRelaxationParameter;
184     float fOldEps = fEps;
185     REAL * aOldSolution = aSolution;
186     REAL * aTestSolution = new REAL[iSystemSize];
187     aSolution = aTestSolution;
188
189     iMaxIterations = iNumberOfIterations;
190     fEps = 0;
191     REAL fMinRes = 1e30;
192     REAL fBestRelaxationParameter;
193
194     for (fRelaxationParameter = fMin; fRelaxationParameter <= fMax; fRelaxationParameter +=
            fStepsize)
195     {
196         #ifdef VERB_SOR_FINDBEST_01
197             std::cout << std::setw(13) << fRelaxationParameter;
198         #endif
199
200         REAL fRes;
201         for (int i=0; i<iSystemSize; aTestSolution[i++]=1);
202         int iNumberOfIterationsDone;
203         Solve(fRes,iNumberOfIterationsDone);
204         #ifdef VERB_SOR_FINDBEST_01
205             std::cout << std::setw(13) << fRes << std::endl;
206         #endif
207         if (iNumberOfIterationsDone >= iNumberOfIterations && fRes < fMinRes)
208         {
209             fMinRes = fRes;
210             fBestRelaxationParameter = fRelaxationParameter;
211         }
212     }
213     #ifdef VERB_SOR_FINDBEST_01
214         std::cout << "Best:␣" << fBestRelaxationParameter << std::endl;
215     #endif
216
217     // restore the old values
218     fRelaxationParameter = fOldRelaxationParameter;
219     iMaxIterations = iOldMaxIterations;
220     fEps = fOldEps;
221     delete[] aTestSolution;
222     aSolution = aOldSolution;
223
224     return(fBestRelaxationParameter);
225 }
```

Listing A.21: LEQSolverRecursive.h

```
#ifndef LEQSOLVERREC_H
#define LEQSOLVERREC_H LEQSOLVERREC_H

#include "LEQSolver.h"

#include <vector>
#include <list>

struct TRecursionStep
{
    unsigned int iRow;
    unsigned int iNewState;
};

/* class for solving the linear equation system for the stationary state
  probabilities of a Markov chain with the recursive method */

class CLEQSolverRecursive : CLEQSolver
{
    public:
        CLEQSolverRecursive(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution);
        bool Solve();
    private:
        void RecursiveCalcStateProb(REAL * aStateProb);
        void RecursiveFindSolutionPath();
        std::vector<struct TRecursionStep> vecSolutionPath;
        std::vector<int> vecRecursionStart;
        std::list<TMatrixElement>::iterator iterMatrixRow;
};

#endif
```

Listing A.22: LEQSolverRecursive.cpp

```
#include "LEQSolverRecursive.h"

#include "Gauss.h"

#include <iomanip>
#include <sstream>
#include <fstream>
#include <cmath>
#include <algorithm>
//#include <ctime>

#define VERB_REC_SOLVE_01   // error messages
#define VERB_REC_SOLVE_02   // info messages
#define VERB_REC_SOLVE_03   // show all computations

#define ITEMS_PER_ROW 8      // maximum number of items to be displayed in one display row


//------------------------------------------------------------------------------
// Construcor
//------------------------------------------------------------------------------
CLEQSolverRecursive::CLEQSolverRecursive(int iSystemSize, CMatrix mCoeffMatrix, REAL * ↩
    aSolution)
: CLEQSolver(iSystemSize, mCoeffMatrix, NULL, aSolution)
{
}

//------------------------------------------------------------------------------
// Solve
//------------------------------------------------------------------------------
bool CLEQSolverRecursive::Solve(void)
{
    unsigned int i,j;

    // if necessary, find the solution path (will be stored in vecSolutionPath
    if (vecSolutionPath.size()==0)
        RecursiveFindSolutionPath();

    VERIFY(vecSolutionPath.size()!=0);
```

```
39      VERIFY(vecRecursionStart.size()!=0);
40
41      // create arrays for the fictitious state probabilities
42      REAL ** aC = new REAL*[vecRecursionStart.size()];
43      for (i=0; i<vecRecursionStart.size(); i++)
44          aC[i] = new REAL[iSystemSize];
45
46      // calculate the fictitious state probabilities for the start vectors
47      // 1000... 0100... 0010... etc.
48      for (i=0; i<vecRecursionStart.size(); i++)
49      {
50          for (j=0; j<vecRecursionStart.size(); aC[i][vecRecursionStart[j++]]=0);
51          aC[i][vecRecursionStart[i]]=1;
52          RecursiveCalcStateProb(aC[i]);
53      }
54
55      #ifdef VERB_REC_SOLVE_03
56      for (i=0; i<vecRecursionStart.size(); i++)
57      {
58          std::cout << "starting␣point␣#" << i << "␣␣␣";
59          for (j=0; j<iSystemSize; j++)
60              {
61                  std::cout << std::setw(5) << j << ":" << std::setw(13) << aC[i][j];
62                  if (j % ITEMS_PER_ROW == (ITEMS_PER_ROW-1)) std::cout << std::endl;
63              }
64              if (iSystemSize % ITEMS_PER_ROW != 0) std::cout << std::endl;
65      }
66      #endif
67
68      // create matrix and solution vector for the Gauss elimination
69      REAL ** M = new REAL*[vecRecursionStart.size()];
70      for (i=0; i<vecRecursionStart.size(); i++)
71          M[i] = new REAL[vecRecursionStart.size()+1];
72      REAL * R = new REAL[vecRecursionStart.size()];
73
74      // fill the first row of the matrix (sum of all state probabilities = 1)
75      for (unsigned int iCol=0; iCol<vecRecursionStart.size(); iCol++)
76      {
77          REAL fSum=0;
78          for (j=0; j<iSystemSize;fSum+=aC[iCol][j++]);
79          M[0][iCol]=fSum;
80      }
81      M[0][vecRecursionStart.size()]=1;
82
83      unsigned int m=1;
84
85      // fill the other rows
86      for (unsigned int iRow=0; iRow<iSystemSize; iRow++)
87      {
88          if (m>vecRecursionStart.size()-1) break;  // matrix full
89          REAL fRowSum=0;
90          for (unsigned int n=0; n<vecRecursionStart.size(); n++)
91          {
92              M[m][n]=0;
93              for (iterMatrixRow = mCoeffMatrix.vecMatrixRows[iRow].begin(); iterMatrixRow != ⤸
                    mCoeffMatrix.vecMatrixRows[iRow].end(); iterMatrixRow++)
94                  M[m][n]+=iterMatrixRow->fValue*aC[n][iterMatrixRow->iCol];
95              fRowSum+=std::fabs((double)M[m][n]);
96          }
97          if (fRowSum < 1e-10) continue; // row with only zeros
98          M[m][vecRecursionStart.size()]=0; // right hand side
99          m++;
100     }
101
102     #ifdef VERB_REC_SOLVE_03
103     PRINTTEXT("remaining␣matrix:");
104     ShowMatrix(M, vecRecursionStart.size()+1, vecRecursionStart.size());
105     #endif
106
107     // calculate the real state probabilities of the starting points
108     if (!GaussianElimination(M,vecRecursionStart.size(),R))
109         ERROR("Gauss␣elimination␣does␣not␣work.")
110
```

```
111     for (i=0; i<vecRecursionStart.size(); i++)
112         aSolution[vecRecursionStart[i]]=R[i];
113 
114     // calculate the real state probabilities of the other states
115     RecursiveCalcStateProb(aSolution);
116 
117     // clean up
118     for (i=0; i<vecRecursionStart.size(); delete[] aC[i++]);
119     delete[] aC;
120 
121     for (i=0; i<vecRecursionStart.size(); delete[] M[i++]);
122     delete[] M;
123     delete[] R;
124 }
125 
126 
127 //--------------------------------------------------------------------------
128 // Calculate the fictitious state probabilities
129 //--------------------------------------------------------------------------
130 void CLEQSolverRecursive::RecursiveCalcStateProb(REAL * aStateProb)
131 {
132     for (unsigned int iStep=0; iStep<vecSolutionPath.size(); iStep++)
133     {
134         REAL fSum=0;
135         REAL fNewStateCoeff;
136 
137         for (iterMatrixRow = mCoeffMatrix.vecMatrixRows[vecSolutionPath[iStep].iRow].begin();
              iterMatrixRow != mCoeffMatrix.vecMatrixRows[vecSolutionPath[iStep].iRow].end();
             iterMatrixRow++)
138             if (iterMatrixRow->iCol!=vecSolutionPath[iStep].iNewState)
139                 fSum+=aStateProb[iterMatrixRow->iCol]*iterMatrixRow->fValue;
140             else fNewStateCoeff =iterMatrixRow->fValue;
141         aStateProb[vecSolutionPath[iStep].iNewState]=-fSum/fNewStateCoeff;
142     }
143 }
144 
145 //--------------------------------------------------------------------------
146 // Find the solution path
147 //--------------------------------------------------------------------------
148 void CLEQSolverRecursive::RecursiveFindSolutionPath()
149 {
150     #ifdef VERB_REC_SOLVE_03
151     PRINTTEXT("Lösungspfad␣wird␣gesucht...");
152     #endif
153 
154     // in bStateProbKnown we save which state probabilities are known
155     bool * bStateProbKnown = new bool[iSystemSize];
156     for (unsigned int i=0; i<iSystemSize; bStateProbKnown[i++]=false);
157 
158     // in bMatrixRowUsed we save how many unknown states each matrix row contains
159     bool * bMatrixRowUsed = new bool[iSystemSize];
160     for (unsigned int i=0; i<iSystemSize; bMatrixRowUsed[i++]=false);
161 
162     unsigned int iNumberOfKnownStates=0;
163     vecRecursionStart.clear();
164     vecSolutionPath.clear();
165     do
166     {
167         // find the minimum of unknown states per row
168         int iMinRow; int iMinValue=iSystemSize+1;
169         for (unsigned int iRow=0; iRow<iSystemSize; iRow++)
170         {
171             if (bMatrixRowUsed[iRow]) continue;
172             int iUnknownStates=0;
173             for (iterMatrixRow = mCoeffMatrix.vecMatrixRows[iRow].begin(); iterMatrixRow !=
                  mCoeffMatrix.vecMatrixRows[iRow].end(); iterMatrixRow++)
174                 if (!bStateProbKnown[iterMatrixRow->iCol])
175                     iUnknownStates++;
176             if (iUnknownStates<iMinValue && iUnknownStates>0)
177             {
178                 iMinValue=iUnknownStates;
179                 iMinRow=iRow;
180             }
```

```
181         }
182         // TRACEVAR(iMinRow); TRACEVAR(iMinValue);
183 
184         // from the row with the fewest unknown states n we choose n-1 new start points
185         int n=0;
186         for(iterMatrixRow = mCoeffMatrix.vecMatrixRows[iMinRow].begin(); iterMatrixRow != ⤸
                mCoeffMatrix.vecMatrixRows[iMinRow].end(); iterMatrixRow++)
187             if (!bStateProbKnown[iterMatrixRow->iCol])
188             {
189                 bStateProbKnown[iterMatrixRow->iCol]=true;
190                 vecRecursionStart.push_back(iterMatrixRow->iCol);
191                 // TRACETEXT("New start point!"); TRACEVAR(vecRecursionStart[⤸
                        vecRecursionStart.size()-1]);
192                 iNumberOfKnownStates++;
193                 n++;
194                 if (n==iMinValue-1)
195                     break;
196             }
197 
198         // find the solution path, until we cannot proceed any more
199         unsigned int iNumberOfKnownStatesOld;
200         do
201         {
202             iNumberOfKnownStatesOld=iNumberOfKnownStates;
203             for (unsigned int iRow=0; iRow<iSystemSize; iRow++)
204             {
205                 if (bMatrixRowUsed[iRow]) continue;   // all states are known
206                 int iNewState=-1;
207 
208                 for (iterMatrixRow = mCoeffMatrix.vecMatrixRows[iRow].begin(); iterMatrixRow ⤸
                        != mCoeffMatrix.vecMatrixRows[iRow].end(); iterMatrixRow++)
209                     if (!bStateProbKnown[iterMatrixRow->iCol])
210                     {
211                         if (iNewState!=-1) goto L1;  // more than 1 unknown state
212                         iNewState=iterMatrixRow->iCol;
213                     }
214 
215                 if (iNewState>-1)   // we found 1 new state ==> save to solution path
216                 {
217                     TRecursionStep step; step.iRow=iRow; step.iNewState=iNewState; ⤸
                            vecSolutionPath.push_back(step);
218                     bMatrixRowUsed[iRow]=true;
219                     bStateProbKnown[iNewState]=true;
220                     iNumberOfKnownStates++;
221                     // TRACEVAR(step.iRow=iRow); TRACEVAR(step.iNewState); TRACEVAR(⤸
                            iNumberOfKnownStates);
222                 }
223             L1:;
224             }
225         }
226         while (iNumberOfKnownStates>iNumberOfKnownStatesOld);
227     }
228     while (iNumberOfKnownStates<iSystemSize); // repeat, until all states are known
229 
230     delete[] bStateProbKnown;
231     delete[] bMatrixRowUsed;
232 
233     #ifdef VERB_REC_SOLVE_03
234     std::cout << "starting␣points:␣";
235     for (unsigned int i=0; i<vecRecursionStart.size(); i++)
236         std::cout << vecRecursionStart[i] << "␣";
237     std::cout << std::endl;
238     std::cout << "solution␣path:␣";
239     for (unsigned int i=0; i<vecSolutionPath.size(); i++)
240         std::cout << vecSolutionPath[i].iRow << "/" << vecSolutionPath[i].iNewState << "␣";
241     std::cout << std::endl;
242     #endif
243 }
```

Listing A.23: LEQSolverLU_Lapack.h

```
1 #ifndef LEQSOLVERLULAPACK_H
2 #define LEQSOLVERLULAPACK_H LEQSOLVERLULAPACK_H
```

```
3
4  #include "LEQSolver.h"
5
6  /* class for solving linear equation systems with lapack */
7
8  class CLEQSolverLU_Lapack : CLEQSolver
9  {
10     public:
11         CLEQSolverLU_Lapack(int iSystemSize, CMatrix mCoeffMatrix, REAL * aRHS, REAL * ↩
               aSolution);
12         bool Solve();
13
14 };
15
16 #endif
```

Listing A.24: LEQSolverLU_Lapack.cpp

```
1  #ifdef LAPACK       // the Lapack library is not available on all systems
2
3  #include <iostream>
4  #include <cmath>
5
6  #include "LEQSolverLU_Lapack.h"
7
8
9  extern "C"
10 {
11   void dgesv_(
12     long int&,  // N, dimension des Problems (i)
13     long int&,  // NRHS, number right hand sides (i)
14     double*,    // A, pointer to matrix (i)
15     long int&,  // LDA, leading dimension of A (i)
16     long int*,  // IPIV, pointer to ipiv (o)
17     double*,    // B, pointer to vector b/solution (i/o)
18     long int&,  // LDB, leading dimension of b (i)
19     long int&); // INFO (o)
20 };
21
22 //-------------------------------------------------------------------------
23 // Construcor
24 //-------------------------------------------------------------------------
25 CLEQSolverLU_Lapack::CLEQSolverLU_Lapack(int iSystemSize, CMatrix mCoeffMatrix, REAL * aRHS, ↩
       REAL * aSolution)
26 : CLEQSolver(iSystemSize, mCoeffMatrix, aRHS, aSolution)
27 {
28 }
29
30 //-------------------------------------------------------------------------
31 // Solve
32 //-------------------------------------------------------------------------
33 bool CLEQSolverLU_Lapack::Solve()
34 {
35     int m,n;
36     mCoeffMatrix.GetSize(m,n);
37     VERIFY(iSystemSize == m && iSystemSize == n);
38
39     long int N = iSystemSize;
40     long int nrhs=1;
41     long int info;
42     long int * ipiv = new long int[iSystemSize];
43
44     // copy the RHS to the solution
45     for (int i=0; i<iSystemSize; aSolution[i]=aRHS[i++]);
46
47     // copy the compressed matrix to an array
48     REAL * A = new REAL[iSystemSize*iSystemSize];
49     for (int i=0; i<iSystemSize*iSystemSize; A[i++]=0);
50     for (int iRow=0; iRow<mCoeffMatrix.vecMatrixRows.size(); iRow++)
51     {
52         std::list<TMatrixElement>::const_iterator iter = mCoeffMatrix.vecMatrixRows[iRow].↩
               begin();
53         while (iter != mCoeffMatrix.vecMatrixRows[iRow].end())
```

```
54              A[iSystemSize*iter->iCol+iRow]=iter++->fValue;
55      }
56
57      // solve
58      dgesv_(N, nrhs, A, N, ipiv, aSolution, N, info);
59
60      delete[] ipiv;
61      delete[] A;
62
63      if (info)
64      {
65          std::cout << "Lapack Error: dgesv: " << info << std::endl;
66          return 0;
67      }
68      return true;
69  }
70
71  #endif
```

A.3.3. Lösen von gewöhnlichen Differentialgleichungen

Die Klasse *CODESolver* (Listings A.25 und A.26) dient als Basisklasse für verschiedene numerische Löser von gewöhnlichen Differentialgleichungen. Die Klassen *CODESolverRK4* (Listings A.27 und A.28) und *CODESolverRK4Adapt* (Listings A.29 und A.30) lösen gewöhnliche Differentialgleichungen mit dem Runge-Kutta-Verfahren ohne beziehungsweise mit adaptiver Schrittweitensteuerung.

Listing A.25: ODESolver.h

```
1  #ifndef ODESOLVER_H
2  #define ODESOLVER_H ODESOLVER_H
3
4  #include "Matrix.h"
5
6  /* base class for solver of ordinary differential equations */
7
8  class CODESolver
9  {
10     public:
11         CODESolver(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution);
12         virtual bool Solve(REAL fStartX, REAL fStopX, void(* Callback) (REAL fX)=NULL) = 0;
13     protected:
14         int iSystemSize;
15         CMatrix mCoeffMatrix;
16         REAL * aSolution;
17 };
18
19 #endif
```

Listing A.26: ODESolver.cpp

```
1  #include "ODESolver.h"
2
3  CODESolver::CODESolver(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution)
4  {
5      this->iSystemSize = iSystemSize;
6      this->mCoeffMatrix = mCoeffMatrix;
7      this->aSolution = aSolution;
8  }
```

Listing A.27: ODESolverRK4.h

```
1  #ifndef ODESOLVERRK4_H
2  #define ODESOLVERRK4_H ODESOLVERRK4_H
3
4  #include "ODESolver.h"
```

```
5
6  /* ODE solver: Runge Kutta 4th order */
7
8  /* some of the code used in this class has been taken from the book
9     "Numerical Recipes in C" (http://www.library.cornell.edu/nr/bookcpdf.html) */
10
11 class CODESolverRK4 : CODESolver
12 {
13     public:
14         CODESolverRK4(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution);
15         bool Solve(REAL fStartX, REAL fStopX, void(* Callback) (REAL fX));
16         void SetParamStepSize(REAL fStepSize);
17     private:
18         void Derivation(REAL x, REAL y[], REAL dxdy[]);
19         void RK4(REAL y[],REAL dydx[],int n,REAL x, REAL h,
20             REAL yout[], void(CODESolverRK4::* derivs) (REAL x, REAL y[], REAL dxdy[]));
21         REAL fStepSize;
22 };
23
24 #endif
```

Listing A.28: ODESolverRK4.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  /* some of the code used in this class has been taken from the book
5     "Numerical Recipes in C" (http://www.library.cornell.edu/nr/bookcpdf.html) */
6
7  #include "ODESolverRK4.h"
8
9  //---------------------------------------------------------------------------
10 // Constructor
11 //---------------------------------------------------------------------------
12 CODESolverRK4::CODESolverRK4(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution)
13 : CODESolver(iSystemSize, mCoeffMatrix, aSolution)
14 {
15     fStepSize = 0.1;
16 }
17
18 //---------------------------------------------------------------------------
19 // Set the step size
20 //---------------------------------------------------------------------------
21 void CODESolverRK4::SetParamStepSize(REAL fStepSizeX)
22 {
23     this->fStepSize = fStepSize;
24 }
25
26 //---------------------------------------------------------------------------
27 // Calculate the derivation y'(x) = f(x,y(x))
28 //---------------------------------------------------------------------------
29 void CODESolverRK4::Derivation(REAL /*x*/, REAL y[], REAL dxdy[])
30 {
31     for (unsigned int iRow = 0; iRow < iSystemSize; iRow++)
32     {
33         dxdy[iRow] = 0;
34         std::list<TMatrixElement>::iterator iterMatrixRow;
35         for(iterMatrixRow = mCoeffMatrix.vecMatrixRows[iRow].begin(); iterMatrixRow != ↩
               mCoeffMatrix.vecMatrixRows[iRow].end(); iterMatrixRow++)
36             dxdy[iRow] += y[iterMatrixRow->iCol]*iterMatrixRow->fValue;
37     }
38 }
39
40 //---------------------------------------------------------------------------
41 // Solve
42 // The Callback function is called after each step
43 //---------------------------------------------------------------------------
44 bool CODESolverRK4::Solve(REAL fStartX, REAL fStopX, void(* Callback) (REAL fX))
45 {
46     REAL fX = fStartX;
47     REAL *dxdy = new REAL[iSystemSize];
48     if (Callback!=NULL) (*Callback)(fX);
49     while (fX < fStopX)
```

```
50      {
51          Derivation(fX,aSolution,dxdy);
52          RK4(aSolution,dxdy,iSystemSize,fX,fStepSize,aSolution,&CODESolverRK4::Derivation);
53          if (Callback!=NULL) (*Callback)(fX);
54          fX += fStepSize;
55      }
56      delete [] dxdy;
57      return true;
58  }
59
60  //------------------------------------------------------------------------------
61  // Runge-Kutta 4th order
62  //------------------------------------------------------------------------------
63
64  void CODESolverRK4::RK4(
65      REAL y[],           // IN: values of y(x)
66      REAL dydx[],        // IN: values of y'(x)
67      int n,              // IN: number of elements in y
68      REAL x,             // IN: x
69      REAL h,             // IN: stepsize
70      REAL yout[],        // OUT: estimated values of y(x+h) (can show to the same array as y)
71      void(CODESolverRK4::* derivs) (REAL x, REAL y[], REAL dxdy[])   // calculation of y'(x)=f(
            x,y(x))
72      )
73  {
74      int i;
75      REAL * dym, * dyt, * yt;
76      dym = new REAL[n];        // TODO: this arrays should not be created new in every step
77      dyt = new REAL[n];
78      yt = new REAL[n];
79      for (i = 0; i < n; i++)
80          yt[i] = y[i] + h/2 * dydx[i];
81      (this->* derivs) (x + h/2, yt, dyt);
82      for (i = 0; i < n; i++)
83          yt[i] = y[i] + h/2 * dyt[i];
84      (this->* derivs) (x + h / 2, yt, dym);
85      for (i = 0; i < n; i++)
86      {
87          yt[i] = y[i] + h * dym[i];
88          dym[i] += dyt[i];
89      }
90      (this->* derivs) (x + h, yt, dyt);
91      for (i = 0; i < n; i++)
92              yout[i] = y[i] + h/6 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
93      delete[] dym;
94      delete[] dyt;
95      delete[] yt;
96  }
```

Listing A.29: ODESolverRK4Adapt.h

```
1   #ifndef ODESOLVERRK4ADAPT_H
2   #define ODESOLVERRK4ADAPT_H ODESOLVERRK4ADAPT_H
3
4   #include "ODESolver.h"
5
6   /* ODE solver: Runge Kutta 4th order with adaptive step size control */
7
8   /* some of the code used in this class has been taken from the book
9      "Numerical Recipes in C" (http://www.library.cornell.edu/nr/bookcpdf.html) */
10
11
12  class CODESolverRK4Adapt : CODESolver
13  {
14      public:
15          CODESolverRK4Adapt(int iSystemSize, CMatrix mCoeffMatrix, REAL * aSolution);
16          void SetParamInitialStepSize(REAL fInitialStepSize);
17          void SetParamMaxError(REAL fMaxError);
18
19          bool Solve(REAL fStartX, REAL fStopX, void(* Callback) (REAL fX));
20      private:
21          REAL fInitialStepSize;
22          REAL fMaxError;
```

```
23          void Derivation(REAL x, REAL y[], REAL dxdy[]);   // Ableitung der
                Zustandswahrscheinlichkeiten (für RK)
24          REAL * dxdy, * yerr, * ytemp, * ak2, * ak3, * ak4, * ak5, * ak6;   // Hilfsfelder für
                RK
25          void rkqs(REAL y[], REAL dydx[], int n, REAL *x, REAL htry, REAL eps,
26          REAL yscal[], REAL *hdid, REAL *hnext, void(CODESolverRK4Adapt::*derivs) (REAL x,
                REAL y[], REAL dxdy[]));
27          void rkck(REAL y[], REAL dydx[], int n, REAL x, REAL h, REAL yout[],
28          REAL yerr[], void(CODESolverRK4Adapt::*derivs) (REAL x, REAL y[], REAL dxdy[]));
29
30  };
31
32  #endif
```

Listing A.30: ODESolverRK4Adapt.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   /* some of the code used in this class has been taken from the book
5      "Numerical Recipes in C" (http://www.library.cornell.edu/nr/bookcpdf.html) */
6
7   #include "ODESolverRK4Adapt.h"
8
9
10  #define VERB_RK4A_SOLVE_01   // error messages
11  #define VERB_RK4A_SOLVE_02   // info messages
12  //#define VERB_RK4A_SOLVE_03   // debug messages
13
14  //---------------------------------------------------------------------------
15  // Constructor
16  //---------------------------------------------------------------------------
17  CODESolverRK4Adapt::CODESolverRK4Adapt(int iSystemSize, CMatrix mCoeffMatrix, REAL *
        aSolution)
18  : CODESolver(iSystemSize, mCoeffMatrix, aSolution)
19  {
20      this->fInitialStepSize = 0.1;
21      this->fMaxError = 1e-4;
22  }
23
24  //---------------------------------------------------------------------------
25  // Set the step size and the precision
26  //---------------------------------------------------------------------------
27  void CODESolverRK4Adapt::SetParamInitialStepSize(REAL fInitialStepSize)
28  {
29      this->fInitialStepSize = fInitialStepSize;
30  }
31  //---------------------------------------------------------------------------
32  void CODESolverRK4Adapt::SetParamMaxError(REAL fMaxError)
33  {
34      this->fMaxError = fMaxError;
35  }
36
37  //---------------------------------------------------------------------------
38  // Calculate the derivation y'(x) = f(x,y(x))
39  //---------------------------------------------------------------------------
40  void CODESolverRK4Adapt::Derivation(REAL /*x*/, REAL y[], REAL dxdy[])
41  {
42      for (unsigned int iRow = 0; iRow < iSystemSize; iRow++)
43      {
44          dxdy[iRow] = 0;
45          std::list<TMatrixElement>::iterator iterMatrixRow;
46          for(iterMatrixRow = mCoeffMatrix.vecMatrixRows[iRow].begin(); iterMatrixRow !=
                mCoeffMatrix.vecMatrixRows[iRow].end(); iterMatrixRow++)
47              dxdy[iRow] += y[iterMatrixRow->iCol]*iterMatrixRow->fValue;
48      }
49  }
50
51  //---------------------------------------------------------------------------
52  // Solve
53  // The Callback function is called after each step
54  //---------------------------------------------------------------------------
55  bool CODESolverRK4Adapt::Solve(REAL fStartX, REAL fStopX, void(* Callback) (REAL fX))
```

```
{
    #ifdef VERB_RK4A_SOLVE_02
        std::cout << "CODESolverRK4Adapt::Solve(...)" << std::endl;
        PRINTVAR(iSystemSize);
    #endif
    dxdy = new REAL[iSystemSize];
    yerr = new REAL[iSystemSize];
    ytemp = new REAL[iSystemSize];
    ak2 = new REAL[iSystemSize];
    ak3 = new REAL[iSystemSize];
    ak4 = new REAL[iSystemSize];
    ak5 = new REAL[iSystemSize];
    ak6 = new REAL[iSystemSize];

    REAL fX = fStartX;
    if (Callback != NULL) (* Callback) (fX);

    REAL fStepSizeNext = fInitialStepSize;
    REAL fStepSizeDid;
    bool bStop=false;
    while (!bStop)
    {
        if (fX+fStepSizeNext>fStopX)
        {
            fStepSizeNext = fStopX-fX;
            bStop = true;
        }
        Derivation(fX,aSolution, dxdy);
        rkqs(aSolution,dxdy,iSystemSize, &fX, fStepSizeNext, fMaxError,
            aSolution, &fStepSizeDid, &fStepSizeNext, &CODESolverRK4Adapt::Derivation);
        if (Callback != NULL) (* Callback) (fX);
    }
    delete[] ytemp;
    delete[] yerr;
    delete[] dxdy;
    delete[] ak2;
    delete[] ak3;
    delete[] ak4;
    delete[] ak5;
    delete[] ak6;
}

inline static REAL FMAX(REAL x, REAL y) { return (y>x ? y : x); }
inline static REAL FMIN(REAL x, REAL y) { return (y<x ? y : x); }


//------------------------------------------------------------------------------
// Runge-Kutta 4th order with adaptive step size control
//------------------------------------------------------------------------------

#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4
// The value ERRCON equals (5/SAFETY) raised to the power (1/PGROW), see use below.
void CODESolverRK4Adapt::rkqs(REAL y[], REAL dydx[], int n, REAL *x, REAL htry, REAL eps,
            REAL yscal[], REAL *hdid, REAL *hnext, void(CODESolverRK4Adapt::*derivs) (REAL x,
                REAL y[], REAL dxdy[]))
// Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy
    and
// adjust stepsize. Input are the dependent variable vector y[1..n] and its derivative dydx
    [1..n]
// at the starting value of the independent variable x. Also input are the stepsize to be
    attempted
// htry, the required accuracy eps, and the vector yscal[1..n] against which the error is
// scaled. On output, y and x are replaced bythei r new values, hdid is the stepsize that was
// actuallyac complished, and hnext is the estimated next stepsize. derivs is the user-
    supplied
// routine that computes the right-hand side derivatives.
{
    int i;
    REAL errmax,h,htemp,xnew;

```

```
124     h=htry; // Set stepsize to the initial trial value.
125     //PRINTVAR(h);
126     for (;;)
127     {
128         rkck(y,dydx,n,*x,h,ytemp,yerr,derivs); // Take a step.
129         errmax=0.0; // Evaluate accuracy.
130         for (i=0;i<n;i++) if (yscal[i]!=0) errmax=FMAX(errmax,std::fabs(yerr[i]/yscal[i]));
131         errmax /= eps; // Scale relative to required tolerance.
132         if (errmax <= 1.0) break; // Step succeeded. Compute size of next step.
133         htemp=SAFETY*h*std::pow((double)errmax,(double)PSHRNK); // Truncation error too large
                , reduce stepsize.
134         h= (h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h)); // No more than a factor of
                10.
135         xnew=(*x)+h;
136         if (xnew == *x) ERROR("stepsize␣underflow␣in␣rkqs");
137     }
138     if (errmax > ERRCON)
139         *hnext=SAFETY*h*std::pow((double)errmax,(double)PGROW);
140     else
141         *hnext=5.0*h; // No more than a factor of 5 increase.
142     *x += (*hdid=h);
143     for (i=0;i<n;i++) y[i]=ytemp[i];
144 }
145 //-------------------------------------------------------------------------
146 void CODESolverRK4Adapt::rkck(REAL y[], REAL dydx[], int n, REAL x, REAL h, REAL yout[],
147         REAL yerr[],  void(CODESolverRK4Adapt::*derivs) (REAL x, REAL y[], REAL dxdy[]))
148
149 // Given values for n variables y[1..n] and their derivatives dydx[1..n] known at x, use
150 // the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval h
151 // and return the incremented variables as yout[1..n]. Also return an estimate of the local
152 // truncation error in yout using the embedded fourth-order method. The user supplies the
        routine
153 // derivs(x,y,dydx), which returns derivatives dydx at x.
154 {
155     int i;
156     static REAL a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,
157         b21=0.2,  b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
158         b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
159         b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
160         b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
161         c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
162         dc5 = -277.00/14336.0;
163     REAL dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
164         dc4=c4-13525.0/55296.0,dc6=c6-0.25;
165
166     for (i=0;i<n;i++) // First step.
167     ytemp[i]=y[i]+b21*h*dydx[i];
168     (this->*derivs)(x+a2*h,ytemp,ak2); // Second step.
169     for (i=0;i<n;i++)
170     ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
171     (this->*derivs)(x+a3*h,ytemp,ak3); // Third step.
172     for (i=0;i<n;i++)
173     ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
174     (this->*derivs)(x+a4*h,ytemp,ak4); // Fourth step.
175     for (i=0;i<n;i++)
176     ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
177     (this->*derivs)(x+a5*h,ytemp,ak5); // Fifth step.
178     for (i=0;i<n;i++)
179     ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]);
180     (this->*derivs)(x+a6*h,ytemp,ak6); // Sixth step.
181     for (i=0;i<n;i++) // Accumulate increments with proper weights.
182     yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
183     for (i=0;i<n;i++)
184     yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
185     // Estimate error as difference between fourth and fifth order methods.
186 }
```

A.3.4. Verschiedene Hilfsfunktionen

In den Dateien *Gauss.h* (Listing A.31) und *Gauss.cpp* (Listing A.32) ist ein Template für das Gaußsche Eliminationsverfahren implementiert. Das Template wird von der Klasse *CLEQSolverRecursive* verwendet.

Listing A.31: Gauss.h

```
/* Gaussian elimination */

#ifndef gauss_h
#define gauss_h gauss_h

template <typename T> bool GaussianElimination(T A, int N, float X[]);
template <typename T> void ShowMatrix(T aMatrix, int iCols, int iRows);

#include "Gauss.cpp"

#endif
```

Listing A.32: Gauss.cpp

```
#include <cmath>
#include <cassert>
#include <iostream>
#include <iomanip>
#include <typeinfo>

#include "../config.h"   // for typedef of REAL

template <typename T> void ShowMatrix(T aMatrix, int iCols, int iRows)
{
    for (int i=0; i<iRows; i++)
    {
        for (int j=0; j<iCols; j++)
            std::cout << std::setw(12) << aMatrix[i][j];
        std::cout << std::endl;
    }
}

template <typename T>void change(T & A, T & B) {T C = A; A = B; B = C;}

//-------------------------------------------------------------------------
// Gaussian Elimination
//
// Input:   A ... augmented coefficient matrix
//                -> the type of the matrix elements must be REAL
//                -> the elements will be changed
//          N ... number of variables
// Output:  X ... solution vector
//
//-------------------------------------------------------------------------
template <typename T> bool GaussianElimination(T A, int N, REAL X[])
{
    assert(typeid(A[0][0])==typeid(REAL));
    for (int iStep = 0; iStep < N; iStep++)
    {
        REAL fMax =std::fabs(A[iStep][iStep]);
        int iPivotRow = iStep;
        for (int i = iStep+1; i < N; i++)
            if (std::fabs(A[i][iStep]) > fMax)
                fMax = std::fabs(A[iPivotRow = i][iStep]);
        if (fMax==0) return false;
        if (iPivotRow!=iStep)
            for (int j = iStep; j < N+1; j++)
                change(A[iStep][j],A[iPivotRow][j]);
        for (int i = iStep+1; i < N; i++)
        {
            REAL fFactor = A[i][iStep]/A[iStep][iStep];
            //A[i][iStep]=0;
            for (int j = iStep+1; j < N+1; j++)
```

```
50                  A[i][j] -= fFactor * A[iStep][j];
51          }
52      }
53      for (int k = N-1; k >= 0; k--)
54      {
55          for (int j = N-1; j > k; j--)
56              A[k][N] -= X[j]*A[k][j];
57          X[k] = A[k][N]/A[k][k];
58      }
59      return true;
60  }
```

A.4. Verschiedenes

In *include.h* (Listing A.33) werden oft benötigte Makros definiert. Die Datei *config.h* (Listing A.34) enthält Konfigurationsdaten. *Makefile*(Listing A.35) ist wie üblich die Konfigurationsdatei für die automatische Übersetzung.

Listing A.33: include.h

```
1
2   // macros
3
4   #include <iostream>
5
6   #ifdef _DEBUG
7       #define TRACEVAR(X) std::cout << (#X) << "␣=␣" << (X) << std::endl;
8       #define TRACETEXT(X) std::cout << (X) << std::endl;
9   #else
10      #define TRACEVAR(X)
11      #define TRACETEXT(X)
12  #endif
13
14  #define ERROR(X) { std::cerr << "ERROR:␣" << X << std::endl; throw; }
15  #define VERIFY(X) if (!(X)) { std::cerr << "ERROR:␣" << (#X) << "␣is␣FALSE" << std::endl; ↩
        throw; }
16
17  #ifdef NO_VERBOSE_OUTPUT
18      #define PRINTVAR(X)
19      #define PRINTTEXT(X)
20  #else
21      #define PRINTVAR(X) std::cout << #X << "␣=␣" << X << std::endl;
22      #define PRINTTEXT(X) std::cout << X << std::endl;
23  #endif
24
25  // adaptions for Microsoft Visual Studio
26
27  // #define MICROSOFT_VS_6
28  #ifdef MICROSOFT_VS_6
29  namespace std
30  {
31      using ::abs;
32      using ::fabs;
33      using ::pow;
34      using ::isspace;
35  }
36  #endif
37
38  // definitions used by some debug functions
39  #define ITEMS_PER_ROW 8
40  #define SCREEN_WIDTH 120
```

Listing A.34: config.h

```
1
2   /* configuration data */
```

```
3
4 // data type for floating point operations
5 #ifdef REALTYPE
6     typedef REALTYPE REAL;
7 #else
8     typedef double REAL;
9 #endif
```

Listing A.35: Makefile

```
1  COMPILER = g++
2  CFLAGS = -O3
3
4  CPPFILES = src/*.cpp src/Matrix/*.cpp
5
6  HFILES = src/*.h src/Matrix/*.h
7
8  CPPFILES_M_M_1_S_Prio = \
9          src/M_M_1_S_Prio/*.cpp \
10         src/M_M_1_S_Prio/M_M_1_S_Prio_CommonBuffer/*.cpp
11
12 HFILES_M_M_1_S_Prio = \
13         src/M_M_1_S_Prio/*.h \
14         src/M_M_1_S_Prio/M_M_1_S_Prio_CommonBuffer/*.h
15
16 CPPFILES_M_M_1_S_TwoPointA = \
17         src/M_M_1_S_TwoPointA/*.cpp
18
19 HFILES_M_M_1_S_TwoPointA = \
20         src/M_M_1_S_TwoPointA/*.h
21
22 CPPFILES_M_M_1_S_OnePointS = \
23         src/M_M_1_S_OnePointS/*.cpp
24
25 HFILES_M_M_1_S_OnePointS = \
26         src/M_M_1_S_OnePointS/*.h
27
28 CPPFILES_M_M_1_S_TwoPointADelay = \
29         src/M_M_1_S_TwoPointADelay/*.cpp
30
31 HFILES_M_M_1_S_TwoPointADelay = \
32         src/M_M_1_S_TwoPointADelay/*.h
33
34
35 M_M_1_S_Prio_T001: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
36         rm -f *.o
37         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_T001
38         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_T001
39
40 M_M_1_S_Prio_T002: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
41         rm -f *.o
42         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_T002
43         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_T002
44
45 M_M_1_S_Prio_T003: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
46         rm -f *.o
47         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_T003
48         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_T003
49
50 M_M_1_S_Prio_T004: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
51         rm -f *.o
52         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_T004
53         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_T004
54
```

```
55 M_M_1_S_Prio_S001: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
56         rm -f *.o
57         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_S001
58         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_S001
59
60 M_M_1_S_Prio_S002: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
61         rm -f *.o
62         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_S002
63         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_S002
64
65 M_M_1_S_Prio_S003: $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) $(HFILES) $(HFILES_M_M_1_S_Prio) ↵
       Makefile
66         rm -f *.o
67         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_Prio) -DMAKEFILE -↵
               DM_M_1_S_PRIO -DTEST_S003
68         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_Prio_S003
69
70 M_M_1_S_TwoPointA_S001: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointA) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointA) Makefile
71         rm -f *.o
72         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointA) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTA -DTEST_S001
73         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointA_S001
74
75 M_M_1_S_TwoPointA_T002: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointA) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointA) Makefile
76         rm -f *.o
77         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointA) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTA -DTEST_T002
78         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointA_T002
79
80 M_M_1_S_OnePointS_F001: $(CPPFILES) $(CPPFILES_M_M_1_S_OnePointS) $(HFILES) $(↵
       HFILES_M_M_1_S_OnePointS) Makefile
81         rm -f *.o
82         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_OnePointS) -DMAKEFILE -↵
               DM_M_1_S_ONEPOINTS -DTEST_F001
83         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_OnePointS_F001
84
85 M_M_1_S_TwoPointADelay_T001: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointADelay) Makefile
86         rm -f *.o
87         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTADELAY -DTEST_T001
88         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointADelay_T001
89
90 M_M_1_S_TwoPointADelay_T002: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointADelay) Makefile
91         rm -f *.o
92         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTADELAY -DTEST_T002
93         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointADelay_T002
94
95 M_M_1_S_TwoPointADelay_T003: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointADelay) Makefile
96         rm -f *.o
97         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTADELAY -DTEST_T003
98         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointADelay_T003
99
100 M_M_1_S_TwoPointADelay_T004: $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) $(HFILES) $(↵
       HFILES_M_M_1_S_TwoPointADelay) Makefile
101         rm -f *.o
102         $(COMPILER) $(CFLAGS) -c $(CPPFILES) $(CPPFILES_M_M_1_S_TwoPointADelay) -DMAKEFILE -↵
               DM_M_1_S_TWOPOINTADELAY -DTEST_T004
103         $(COMPILER) $(CFLAGS) *.o -oM_M_1_S_TwoPointADelay_T004
104
105 clean:
106         rm -f *.o \
107         M_M_1_S_Prio_T001 \
```

```
        M_M_1_S_Prio_T002 \
        M_M_1_S_Prio_T003 \
        M_M_1_S_Prio_T004 \
        M_M_1_S_Prio_S001 \
        M_M_1_S_Prio_S002 \
        M_M_1_S_Prio_S003 \
        M_M_1_S_TwoPointA_S001 \
        M_M_1_S_TwoPointA_T002 \
        M_M_1_S_OnePointS_F001 \
        M_M_1_S_TwoPointADelay_T001 \
        M_M_1_S_TwoPointADelay_T002 \
        M_M_1_S_TwoPointADelay_T003 \
        M_M_1_S_TwoPointADelay_T004
```

A.5. Anwendung

In *main.cpp* (Listing A.36) ist zu sehen, wie die gezeigten Klassen verwendet werden.

Listing A.36: main.cpp

```
#include "config.h"

#ifndef MAKEFILE

    #define M_M_1_S_PRIO
    //#define M_M_1_S_TwoPointA
    //#define M_M_1_S_OnePointS

    #define TEST_T001
    //#define TEST_S001
    //#define TEST_F001

#endif

#include <iomanip>
#include <iostream>
#include <fstream>
#include <sstream>

//######################################################################
//
// M/M/1/S-Prio
//
// M/M/1/S queueing system with classes of customers
//
//######################################################################

#ifdef M_M_1_S_PRIO

#include "M_M_1_S_Prio/M_M_1_S_Prio_CommonBuffer/M_M_1_S_Prio_CommonBuffer_Random.h"
#include "M_M_1_S_Prio/M_M_1_S_Prio_CommonBuffer/M_M_1_S_Prio_CommonBuffer_Prio.h"

//**********************************************************************
// T001 - 2 classes, system is flooded with customers of one class
//**********************************************************************

#ifdef TEST_T001
    #define BLOCK_T001
#endif

#ifdef BLOCK_T001

//----------------------------------------------------------------------

#define CLASS 2
#define BUFFER 20

M_M_1_S_Prio_CommonBuffer_Prio system1(CLASS,BUFFER);
```

```
49
50 //------------------------------------------------------------------------
51
52 const float fStartTime = 10;
53 float fStopTime;
54 std::ofstream outfile;
55
56 void FigureCallback(REAL fX)
57 {
58     outfile <<
59         std::setw(20) << fX <<
60         std::setw(20) << system1.GetArrivalRate(1) <<
61         std::setw(20) << system1.GetServiceRate(1) <<
62         std::setw(20) << system1.GetArrivalRate(2) <<
63         std::setw(20) << system1.GetServiceRate(2) <<
64         std::setw(20) << system1.GetNumberOfCustomersMean(1) <<
65         std::setw(20) << system1.GetNumberOfCustomersMean(2) <<
66         std::setw(20) << system1.GetBlockingProbability(1) <<
67         std::setw(20) << system1.GetBlockingProbability(2) <<
68         std::endl;
69 }
70
71 void Figure(REAL fDuration, char * szFileName)
72 {
73     std::cout << "Dauer:␣" << fDuration << std::endl;
74
75     outfile.open(szFileName,std::ios::out);
76     outfile << "%␣" << szFileName << std::endl;
77     outfile << "%" <<
78         std::setw(19) << "time" <<
79         std::setw(20) << "lambda␣1" <<
80         std::setw(20) << "mu␣1" <<
81         std::setw(20) << "lambda␣2" <<
82         std::setw(20) << "mu␣2" <<
83         std::setw(20) << "E[X]␣1" <<
84         std::setw(20) << "E[X]␣2" <<
85         std::setw(20) << "pBlocking␣1" <<
86         std::setw(20) << "pBlocking␣2" <<
87         std::endl;
88
89     fStopTime = fStartTime+fDuration;
90
91     #ifdef TEST_T001
92         system1.SetArrivalRate(1,0.2);
93         system1.SetArrivalRate(2,0.2);
94         system1.SetServiceRate(1,1.0);
95         system1.SetServiceRate(2,1.0);
96     #endif
97
98     system1.CalcStationarySystemStateProbabilities_SOR(true);
99
100    //system1.PrintSystemStateProbabilities();
101
102    system1.CalcTransientSystemStateProbabilities_RK4Adapt(0, fStartTime, &FigureCallback);
103
104    #ifdef TEST_T001
105        system1.SetArrivalRate(1,6.0);
106    #endif
107
108    system1.CalcTransientSystemStateProbabilities_RK4Adapt(fStartTime, fStopTime, &
           FigureCallback);
109    #ifdef TEST_T001
110        system1.SetArrivalRate(1,0.2);
111    #endif
112    system1.CalcTransientSystemStateProbabilities_RK4Adapt(fStopTime, 120, &FigureCallback);
113
114    outfile.close();
115 }
116
117 int main(void)
118 {
119     Figure(30,"results/M_M_1_S_Prio_T001.dat");
120     return 0;
```

```
121 }
122
123 #endif
124
125
126 //**************************************************************************
127 // T002 - 2 classes, system is flooded with customers of one class -- MOVIE
128 //**************************************************************************
129
130 #ifdef TEST_T002
131     #define BLOCK_T002
132 #endif
133
134 #ifdef BLOCK_T002
135
136 //--------------------------------------------------------------------------
137
138 #define CLASS 2
139 #define BUFFER 20
140
141 //--------------------------------------------------------------------------
142
143 #define TIME_START 10
144 #define TIME_STOP 80
145 #define TIME_END 150
146 #define INTERVAL 1
147 #define ARRIVAL_RATE_HIGH_1 5
148 #define ARRIVAL_RATE_HIGH_2 0.8
149 #define ARRIVAL_RATE_LOW_1 0.2
150 #define ARRIVAL_RATE_LOW_2 0.7
151 #define SERVICE_RATE 1
152 //#define PRIO
153 #define RANDOM
154
155 #ifdef PRIO
156     M_M_1_S_Prio_CommonBuffer_Prio system1(CLASS,BUFFER);
157 #endif
158
159 #ifdef RANDOM
160     M_M_1_S_Prio_CommonBuffer_Random system1(CLASS,BUFFER);
161 #endif
162
163 void MovieCallback(REAL fX)
164 {
165     static float fTrigger=0;
166     static int iNumber = 0;
167
168     if (fX>=fTrigger)
169     {
170         PRINTVAR(fTrigger);
171         std::stringstream s;
172         s << "results/M_M_1_S_Prio_T002_pic_" << iNumber++ << ".dat";
173         std::ofstream outfile(s.str().c_str(),std::ios::out);
174
175         for (unsigned int i=0; i<=system1.GetSystemSize(); i++)
176         {
177             for (unsigned int j=0; j<=system1.GetSystemSize(); j++)
178             {
179                 REAL p = 0;
180                 for (unsigned int k=0; k<=2; k++)
181                 {
182                     TSystemState state;
183                     state.n[1]=i;
184                     state.n[2]=j;
185                     state.k=k;
186                     unsigned int iStateNumber = system1.GetSystemStateNumber(state);
187                     if (iStateNumber != (unsigned int)-1)
188                         p += system1.GetSystemStateProbability(iStateNumber);
189                 }
190                 outfile << std::setw(15) << p;
191             }
192             outfile << std::endl;
193         }
```

```
194         outfile.close();
195 
196         fTrigger += INTERVAL;
197     }
198 }
199 
200 
201 void movie()
202 {
203      // we should not use adaptive step size control (the stepsize could be greater than
         INTERVAL)
204      system1.SetServiceRate(1,SERVICE_RATE);
205      system1.SetServiceRate(2,SERVICE_RATE);
206 
207      system1.SetArrivalRate(1,ARRIVAL_RATE_LOW_1);
208      system1.SetArrivalRate(2,ARRIVAL_RATE_LOW_2);
209      system1.CalcStationarySystemStateProbabilities_SOR();
210      system1.CalcTransientSystemStateProbabilities_RK4(0, TIME_START, 0.05,  &MovieCallback);
211 
212      system1.SetArrivalRate(1,ARRIVAL_RATE_HIGH_1);
213      system1.SetArrivalRate(2,ARRIVAL_RATE_HIGH_2);
214      system1.CalcTransientSystemStateProbabilities_RK4(TIME_START, TIME_STOP, 0.05, &
         MovieCallback);
215 
216      system1.SetArrivalRate(1,ARRIVAL_RATE_LOW_1);
217      system1.SetArrivalRate(2,ARRIVAL_RATE_LOW_2);
218      system1.CalcTransientSystemStateProbabilities_RK4(TIME_STOP, TIME_END, 0.05, &
         MovieCallback);
219 }
220 
221 
222 int main(void)
223 {
224     movie();
225     return 0;
226 }
227 
228 #endif
229 
230 
231 //***************************************************************************
232 // T003 - 2 classes, system is flooded with customers of one class (prio)
233 //***************************************************************************
234 
235 #ifdef TEST_T003
236     #define BLOCK_T003_T004
237 #endif
238 
239 //***************************************************************************
240 // T004 - 2 classes, system is flooded with customers of one class (random)
241 //***************************************************************************
242 
243 #ifdef TEST_T004
244     #define BLOCK_T003_T004
245 #endif
246 
247 #ifdef BLOCK_T003_T004
248 
249 //---------------------------------------------------------------------------
250 
251 #define CLASS 2
252 #define BUFFER 20
253 
254 #define TIME_START 10
255 #define TIME_STOP 80
256 #define TIME_END 150
257 #define INTERVAL 1
258 #define ARRIVAL_RATE_HIGH_1 5
259 #define ARRIVAL_RATE_HIGH_2 0.4
260 #define ARRIVAL_RATE_LOW_1 0.4
261 #define ARRIVAL_RATE_LOW_2 0.4
262 #define SERVICE_RATE 1
263 
```

```
264 #ifdef TEST_T003
265     M_M_1_S_Prio_CommonBuffer_Prio system1(CLASS,BUFFER);
266 #endif
267
268 #ifdef TEST_T004
269     M_M_1_S_Prio_CommonBuffer_Random system1(CLASS,BUFFER);
270 #endif
271
272 std::ofstream outfile;
273
274 void Callback(REAL fX)
275 {
276     outfile <<
277         std::setw(20) << fX <<
278         std::setw(20) << system1.GetArrivalRate(1) <<
279         std::setw(20) << system1.GetServiceRate(1) <<
280         std::setw(20) << system1.GetArrivalRate(2) <<
281         std::setw(20) << system1.GetServiceRate(2) <<
282         std::setw(20) << system1.GetNumberOfCustomersMean(1) <<
283         std::setw(20) << system1.GetNumberOfCustomersMean(2) <<
284         std::setw(20) << system1.GetBlockingProbability(1) <<
285         std::setw(20) << system1.GetBlockingProbability(2) <<
286         std::endl;
287 }
288
289 void start(char * szFileName)
290 {
291
292     outfile.open(szFileName,std::ios::out);
293     outfile << "%␣" << szFileName << std::endl;
294     outfile << "%" <<
295         std::setw(19) << "time" <<
296         std::setw(20) << "lambda␣1" <<
297         std::setw(20) << "mu␣1" <<
298         std::setw(20) << "lambda␣2" <<
299         std::setw(20) << "mu␣2" <<
300         std::setw(20) << "E[X]␣1" <<
301         std::setw(20) << "E[X]␣2" <<
302         std::setw(20) << "pBlocking␣1" <<
303         std::setw(20) << "pBlocking␣2" <<
304         std::endl;
305
306      system1.SetServiceRate(1,SERVICE_RATE);
307      system1.SetServiceRate(2,SERVICE_RATE);
308
309      system1.SetArrivalRate(1,ARRIVAL_RATE_LOW_1);
310      system1.SetArrivalRate(2,ARRIVAL_RATE_LOW_2);
311      system1.CalcStationarySystemStateProbabilities_SOR();
312      system1.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_START, &Callback);
313
314      system1.SetArrivalRate(1,ARRIVAL_RATE_HIGH_1);
315      system1.SetArrivalRate(2,ARRIVAL_RATE_HIGH_2);
316      system1.CalcTransientSystemStateProbabilities_RK4Adapt(TIME_START, TIME_STOP, &Callback)
         ;
317
318      system1.SetArrivalRate(1,ARRIVAL_RATE_LOW_1);
319      system1.SetArrivalRate(2,ARRIVAL_RATE_LOW_2);
320      system1.CalcTransientSystemStateProbabilities_RK4Adapt(TIME_STOP, TIME_END, &Callback);
321 }
322
323 int main(void)
324 {
325     #ifdef TEST_T003
326         start("results/M_M_1_S_Prio_T003_T004/M_M_1_S_Prio_T003.dat");
327     #endif
328     #ifdef TEST_T004
329         start("results/M_M_1_S_Prio_T003_T004/M_M_1_S_Prio_T004.dat");
330     #endif
331     return 0;
332 }
333
334 #endif
335
```

```
336
337
338 //*****************************************************************************
339 // S001 - 2 classes, create 16 3d-plots for number of customers
340 //
341 //            .
342 //  # of      . .
343 //  custo-    . . .
344 //  mers 1    o . . .
345 //            o o . . .
346 //            O O o . . .
347 //
348 //            # of customers 2
349 //
350 //*****************************************************************************
351
352 #ifdef TEST_S001
353     #define BLOCK_S001
354 #endif
355
356 #ifdef BLOCK_S001
357
358 //-----------------------------------------------------------------------------
359
360 #define CLASS 2
361 #define BUFFER 20
362
363 M_M_1_S_Prio_CommonBuffer_Random system1(CLASS,BUFFER);
364
365 //-----------------------------------------------------------------------------
366
367 void PlotOneStateProb(char * szFilename)
368 {
369     std::ofstream outfile(szFilename,std::ios::out);
370
371     system1.CalcStationarySystemStateProbabilities_SOR();
372
373     for (unsigned int i=0; i<=system1.GetSystemSize(); i++)
374     {
375         for (unsigned int j=0; j<=system1.GetSystemSize(); j++)
376         {
377             REAL p = 0;
378             for (unsigned int k=0; k<=2; k++)
379             {
380                 TSystemState state;
381                 state.n[1]=i;
382                 state.n[2]=j;
383                 state.k=k;
384                 unsigned int iStateNumber = system1.GetSystemStateNumber(state);
385                 if (iStateNumber != (unsigned int)-1)
386                     p += system1.GetSystemStateProbability(iStateNumber);
387             }
388             outfile << std::setw(15) << p;
389         }
390         outfile << std::endl;
391     }
392 }
393
394 void PlotStateProb()
395 {
396     float L1[4] = {0.57,0.58,0.75,1.5};
397     float L2[4] = {0.57,0.58,0.75,1.5};
398
399     system1.SetServiceRate(1,1);
400     system1.SetServiceRate(2,1);
401
402     system1.SetArrivalRate(1,L1[0]);
403     system1.SetArrivalRate(2,L2[0]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic01.dat");
404     system1.SetArrivalRate(2,L2[1]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic02.dat");
405     system1.SetArrivalRate(2,L2[2]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic03.dat");
406     system1.SetArrivalRate(2,L2[3]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic04.dat");
407
408     system1.SetArrivalRate(1,L1[1]);
```

```
409     system1.SetArrivalRate(2,L2[0]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic05.dat");
410     system1.SetArrivalRate(2,L2[1]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic06.dat");
411     system1.SetArrivalRate(2,L2[2]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic07.dat");
412     system1.SetArrivalRate(2,L2[3]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic08.dat");
413 
414     system1.SetArrivalRate(1,L1[2]);
415     system1.SetArrivalRate(2,L2[0]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic09.dat");
416     system1.SetArrivalRate(2,L2[1]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic10.dat");
417     system1.SetArrivalRate(2,L2[2]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic11.dat");
418     system1.SetArrivalRate(2,L2[3]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic12.dat");
419 
420     system1.SetArrivalRate(1,L1[3]);
421     system1.SetArrivalRate(2,L2[0]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic13.dat");
422     system1.SetArrivalRate(2,L2[1]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic14.dat");
423     system1.SetArrivalRate(2,L2[2]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic15.dat");
424     system1.SetArrivalRate(2,L2[3]); PlotOneStateProb("results/M_M_1_S_Prio_S001_pic16.dat");
425 }
426 
427 int main()
428 {
429     //system1.PrintSystemStates();
430     PlotStateProb();
431     return 0;
432 }
433 #endif
434 
435 //*****************************************************************************
436 // S002 - 2 classes, number of customers depending on arrival rate of class 1
437 //          priority service
438 //*****************************************************************************
439 
440 #ifdef TEST_S002
441     #define BLOCK_S002_S003
442 #endif
443 
444 //*****************************************************************************
445 // S003 - 2 classes, number of customers depending on arrival rate of class 1
446 //          random service
447 //*****************************************************************************
448 
449 #ifdef TEST_S003
450     #define BLOCK_S002_S003
451 #endif
452 
453 #ifdef BLOCK_S002_S003
454 
455 //-----------------------------------------------------------------------------
456 
457 #define CLASS 2
458 #define BUFFER 20
459 
460 #ifdef TEST_S002
461     M_M_1_S_Prio_CommonBuffer_Prio system1(CLASS,BUFFER);
462 #endif
463 
464 #ifdef TEST_S003
465     M_M_1_S_Prio_CommonBuffer_Random system1(CLASS,BUFFER);
466 #endif
467 
468 int main()
469 {
470     system1.SetServiceRate(1,1);
471     system1.SetServiceRate(2,1);
472     system1.SetArrivalRate(2,0.2);
473 
474     system1.SetArrivalRate(1,0.01);
475     system1.CalcStationarySystemStateProbabilities_SOR();
476 
477     std::ofstream outfile;
478 
479     #ifdef TEST_S002
480         #define FILENAME "results/M_M_1_S_Prio_S002_S003/M_M_1_S_Prio_S002.dat"
481     #endif
```

```
482     #ifdef TEST_S003
483         #define FILENAME "results/M_M_1_S_Prio_S002_S003/M_M_1_S_Prio_S003.dat"
484     #endif
485 
486     outfile.open(FILENAME,std::ios::out);
487     std::cout << "Opening␣'" << FILENAME << "'␣..." << std::endl;
488 
489     outfile << "%" <<
490         std::setw(20) << "lambda1" <<
491         std::setw(20) << "lambda2" <<
492         std::setw(20) << "mu1" <<
493         std::setw(20) << "mu2" <<
494         std::setw(20) << "E[X1]" <<
495         std::setw(20) << "E[X2]" <<
496         std::setw(20) << "p_Blocking" <<
497         std::endl;
498 
499     for (int i=0; i<=200; i++)
500     {
501         system1.SetArrivalRate(1,i/100.0);
502 
503         system1.CalcStationarySystemStateProbabilities_SOR(false);
504         outfile <<
505             std::setw(20) << system1.GetArrivalRate(1) <<
506             std::setw(20) << system1.GetArrivalRate(2) <<
507             std::setw(20) << system1.GetServiceRate(1) <<
508             std::setw(20) << system1.GetServiceRate(2) <<
509             std::setw(20) << system1.GetNumberOfCustomersMean(1) <<
510             std::setw(20) << system1.GetNumberOfCustomersMean(2) <<
511             std::setw(20) << system1.GetBlockingProbability(1) <<
512             std::endl;
513     }
514     outfile.close();
515     return 0;
516 }
517 
518 #endif
519 #endif
520 
521 //##########################################################################
522 //
523 // M/M/1/S-TwoPointA
524 //
525 // M/M/1/S queueing system with two-point control of the arrival rate
526 //
527 //##########################################################################
528 
529 #ifdef M_M_1_S_TWOPOINTA
530 
531 #include "M_M_1_S_TwoPointA/M_M_1_S_TwoPointA.h"
532 
533 //**************************************************************************
534 // S001 - comparison of recursive solving and SOR
535 //**************************************************************************
536 
537 #ifdef TEST_S001
538     #define BLOCK_S001
539 #endif
540 
541 #ifdef BLOCK_S001
542 
543 
544 //--------------------------------------------------------------------------
545 
546 #define SYSTEMSIZE 5
547 #define THRESHOLD_GO 1
548 #define THRESHOLD_STOP 4
549 
550 M_M_1_S_TwoPointA system1(SYSTEMSIZE,THRESHOLD_STOP,THRESHOLD_GO);
551 
552 //--------------------------------------------------------------------------
553 
554 int main(void)
```

```
555 {
556     system1.SetArrivalRateNormal(0.8);
557     system1.SetArrivalRateReduced(0.4);
558     system1.SetServiceRate(1);
559 
560     PRINTTEXT("***␣recursive␣***");
561     system1.CalcStationarySystemStateProbabilities_Recursive();
562     system1.PrintSystemStateProbabilities();
563 
564     PRINTTEXT("***␣SOR␣***");
565     system1.CalcStationarySystemStateProbabilities_SOR();
566     system1.PrintSystemStateProbabilities();
567 
568     system1.matrixSystemState.PrintMatrix(true);
569     system1.matrixSystemState.PrintStatistics();
570 
571     return 0;
572 }
573 
574 #endif
575 
576 
577 //***************************************************************************
578 // T002 - number of customers
579 //***************************************************************************
580 
581 #ifdef TEST_T002
582     #define BLOCK_T002
583 #endif
584 
585 #ifdef BLOCK_T002
586 
587 std::ofstream outfile;
588 
589 
590 //---------------------------------------------------------------------------
591 
592 #define SYSTEMSIZE 40
593 #define THRESHOLD_STOP 32
594 #define THRESHOLD_GO 12
595 #define TIME_START 10
596 #define TIME_STOP 100
597 #define TIME_END 120
598 #define ARRIVAL_RATE_NORMAL_LOW 0.5
599 #define ARRIVAL_RATE_NORMAL_HIGH 6
600 #define ARRIVAL_RATE_REDUCED 0
601 
602 #define SERVICE_RATE 1
603 
604 M_M_1_S_TwoPointA system1(SYSTEMSIZE,THRESHOLD_STOP,THRESHOLD_GO);
605 
606 //---------------------------------------------------------------------------
607 
608 void Callback(REAL fX)
609 {
610     outfile <<
611         std::setw(20) << fX <<
612         std::setw(20) << system1.GetNumberOfCustomersMean() <<
613         std::setw(20) << system1.GetBlockingProbability() <<
614         std::endl;
615 }
616 
617 void Start(char * szFileName)
618 {
619     std::cout << szFileName << std::endl;
620     outfile.open(szFileName,std::ios::out);
621     outfile << "%␣" << szFileName << std::endl;
622     outfile << "%" <<
623         std::setw(19) << "time" <<
624         std::setw(20) << "E[X]" <<
625         std::setw(20) << "p_Blocking" <<
626         std::endl;
627 
```

```
628     system1.SetArrivalRateReduced(ARRIVAL_RATE_REDUCED);
629     system1.SetServiceRate(SERVICE_RATE);
630 
631     system1.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
632 
633     // we start in idle state
634     //system1.SetSystemStateProbability(0,1);
635     //for (unsigned int i=1; i<system1.GetNumberOfSystemStates(); system1.↩
            SetSystemStateProbability(i++,0));
636     system1.CalcStationarySystemStateProbabilities_SOR();
637 
638     system1.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_START, &Callback);
639     system1.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_HIGH);
640     system1.CalcTransientSystemStateProbabilities_RK4Adapt(TIME_START, TIME_STOP, &Callback);
641     system1.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
642     system1.CalcTransientSystemStateProbabilities_RK4Adapt(TIME_STOP, TIME_END, &Callback);
643     outfile.close();
644 
645 }
646 
647 int main()
648 {
649     std::cout << "*␣*␣*␣*␣*␣␣M_M_1_S_TwoPointA_T002␣␣*␣*␣*␣*␣*" << std::endl;
650     Start("results/M_M_1_S_TwoPointA_T002/M_M_1_S_TwoPointA_T002.dat");
651 }
652 
653 #endif
654 
655 #endif
656 
657 
658 //########################################################################
659 //
660 // M/M/1/S-OnePointS
661 //
662 // M/M/1/S queueing system with one-point control of the service rate
663 //
664 //########################################################################
665 
666 #ifdef M_M_1_S_ONEPOINTS
667 
668 //************************************************************************
669 // F001 - flowtime
670 //************************************************************************
671 
672 #ifdef TEST_F001
673     #define BLOCK_F001
674 #endif
675 
676 #ifdef BLOCK_F001
677 
678 std::ofstream outfile;
679 
680 #include "M_M_1_S_OnePointS/M_M_1_S_OnePointS.h"
681 
682 //------------------------------------------------------------------------
683 
684 #define MU_REDUCED 0.1
685 #define MU_NORMAL 1.0
686 #define LAMBDA_NORMAL 0.5
687 #define LAMBDA_REDUCED 0.01
688 
689 #define STARTTIME 4
690 #define STOPTIME 14
691 
692 #define SYSTEMSIZE 4
693 #define THRESHOLD 2
694 
695 M_M_1_S_OnePointS eq1(SYSTEMSIZE,THRESHOLD);
696 
697 //------------------------------------------------------------------------
698 
699 void Callback_Flowtime(REAL fX)
```

```
{
    // to calculate the mean value of the flow time, we have to integrate
    // over the CCDF
    static REAL fXOld;
    static REAL WeightedFlowTimeCCDFOld;
    static REAL fFlowTime;
    if (fX==0)
    {
        fXOld = 0;
        WeightedFlowTimeCCDFOld = 0;
        fFlowTime = 0;
    }
    REAL WeightedFlowTimeCCDF = eq1.CalcWeightedFlowTimeCCDF();
    fFlowTime += (fX-fXOld)*(WeightedFlowTimeCCDF+WeightedFlowTimeCCDFOld)/2;
    WeightedFlowTimeCCDFOld =  WeightedFlowTimeCCDF;
    fXOld = fX;

    outfile <<
        std::setw(20) << fX <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(0) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(4) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(7) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(9) <<
        std::setw(20) << WeightedFlowTimeCCDF <<
        std::setw(20) << fFlowTime <<
/*        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(0) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(1) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(2) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(3) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(4) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(5) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(6) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(7) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(8) <<
        std::setw(20) << eq1.GetFlowStateSojournTimeCCDF(9) <<  */
        std::endl;
}

void Callback_Systemstate(REAL fX)
{
    outfile <<
        std::setw(20) << fX <<
        std::setw(20) << eq1.GetArrivalRate() <<
        std::setw(20) << eq1.GetServiceRateReduced() <<
        std::setw(20) << eq1.GetServiceRateNormal() <<
        std::setw(20) << eq1.GetNumberOfCustomersMean() <<
        std::endl;
}

int main(void)
{
    // *** flowtime ***

    outfile.open("results/M_M_1_S_OnePointA_F001_FlowTime.dat",std::ios::out);
    outfile << "%" << "M_M_1_S_OnePointA_F001_FlowTime" << std::endl;
    outfile << "%" <<
        std::setw(19) << "time" <<
        std::setw(20) << "TF-CDF|1-0" <<
        std::setw(20) << "TF-CDF|2-1" <<
        std::setw(20) << "TF-CDF|3-2" <<
        std::setw(20) << "TF-CDF|4-3" <<
        std::setw(20) << "TF-CDF" <<
        std::setw(20) << "Int(0,t,TF-CDF)" <<
        std::endl;

    eq1.SetArrivalRate(LAMBDA_NORMAL);
    eq1.SetServiceRateNormal(MU_NORMAL);
    eq1.SetServiceRateReduced(MU_REDUCED);

    PRINTVAR(eq1.GetNumberOfSystemStates());
    PRINTVAR(eq1.GetNumberOfFlowStates());

    eq1.CalcCoeffMatrixFlowTime();
```

```
773     eq1.CalcCoeffMatrixSystemState();
774     std::cout << "system␣state␣matrix:␣" << std::endl;
775     eq1.matrixSystemState.PrintMatrix();
776     std::cout << "flow␣state␣matrix:␣" << std::endl;
777     eq1.matrixFlowTime.PrintMatrix();
778
779     eq1.CalcStationarySystemStateProbabilities_SOR();
780
781     eq1.InitCalcFlowStateSojournTimeCCDF();
782     eq1.CalcFlowStateSojournTimeCCDF_RK4Adapt(0, STARTTIME, &Callback_Flowtime);
783     eq1.SetArrivalRate(LAMBDA_REDUCED);
784     eq1.CalcFlowStateSojournTimeCCDF_RK4Adapt(STARTTIME, STOPTIME,  &Callback_Flowtime);
785     eq1.SetArrivalRate(LAMBDA_NORMAL);
786     eq1.CalcFlowStateSojournTimeCCDF_RK4Adapt(STOPTIME, 20, &Callback_Flowtime);
787
788     outfile.close();
789
790     // *** system state probabiliy ***
791
792     outfile.open("results/M_M_1_S_OnePointA_F001_SystemState.dat",std::ios::out);
793     outfile << "%" << "M_M_1_S_OnePointA_F001_SystemState" << std::endl;
794     outfile << "%" <<
795         std::setw(19) << "time" <<
796         std::setw(20) << "lambda" <<
797         std::setw(20) << "mu_red" <<
798         std::setw(20) << "mu_norm" <<
799         std::setw(20) << "EX" <<
800         std::endl;
801
802     eq1.SetArrivalRate(LAMBDA_NORMAL);
803     eq1.SetServiceRateNormal(MU_NORMAL);
804     eq1.SetServiceRateReduced(MU_REDUCED);
805
806     eq1.CalcStationarySystemStateProbabilities_SOR();
807     eq1.CalcTransientSystemStateProbabilities_RK4Adapt(0, STARTTIME, &Callback_Systemstate);
808     eq1.SetArrivalRate(LAMBDA_REDUCED);
809     eq1.CalcTransientSystemStateProbabilities_RK4Adapt(STARTTIME, STOPTIME, &↩
        Callback_Systemstate);
810     eq1.SetArrivalRate(LAMBDA_NORMAL);
811     eq1.CalcTransientSystemStateProbabilities_RK4Adapt(STOPTIME, 20, &Callback_Systemstate);
812
813     outfile.close();
814
815     return 0;
816 }
817
818 #endif  // TEST_F001
819
820 #endif
821
822
823 //########################################################################
824 //
825 // M/M/1/S-TwoPointADelay
826 //
827 // M/M/1/S queueing system with delayed two-point control of the arrival rate
828 //
829 //########################################################################
830
831 #ifdef M_M_1_S_TWOPOINTADELAY
832
833 #include "M_M_1_S_TwoPointADelay/M_M_1_S_TwoPointADelay.h"
834 #include "Matrix/LEQSolverSOR.h"
835
836 //************************************************************************
837 // T001 - number of customers (depending on the two thresholds)
838 //************************************************************************
839
840 #ifdef TEST_T001
841     #define BLOCK_T001_T002
842 #endif
843
844 //************************************************************************
```

```
845 // T002 - number of customers (depending on the message transfer rate)
846 //***************************************************************************
847
848 #ifdef TEST_T002
849     #define BLOCK_T001_T002
850 #endif
851
852 #ifdef BLOCK_T001_T002
853
854 std::ofstream outfile;
855
856 //---------------------------------------------------------------------------
857
858 //#define SYSTEMSIZE 4
859
860 #define SYSTEMSIZE 40
861 #define MAX_MSG_ON_THE_WAY 8        // should be even!!
862
863 #define THRESHOLD_STOP 32
864 #define THRESHOLD_GO 12             // for TEST_T002
865 //#define THRESHOLD_STOP 3
866
867 #define TIME_START 10
868 #define TIME_STOP 100
869 #define TIME_END 120
870
871 #define ARRIVAL_RATE_NORMAL_LOW 0.5
872 #define ARRIVAL_RATE_NORMAL_HIGH 6
873 #define ARRIVAL_RATE_REDUCED 0
874 #define SERVICE_RATE 1
875 #define MESSAGE_RATE 1              // for TEST_T001
876
877 M_M_1_S_TwoPointADelay * system1;
878
879 //---------------------------------------------------------------------------
880
881 void Callback(REAL fX)
882 {
883     outfile <<
884         std::setw(20) << fX <<
885         std::setw(20) << system1->GetNumberOfCustomersMean() <<
886         std::setw(20) << system1->GetBlockingProbability() <<
887         std::endl;
888 }
889
890 void Start(int iThresholdGo,float fControlMessageTransferRate, char * szFileName)
891 {
892     system1 = new M_M_1_S_TwoPointADelay(SYSTEMSIZE,THRESHOLD_STOP,iThresholdGo,
        MAX_MSG_ON_THE_WAY);
893
894     std::cout << szFileName << std::endl;
895     outfile.open(szFileName,std::ios::out);
896     outfile << "% " << szFileName << std::endl;
897     outfile << "%" <<
898         std::setw(19) << "time" <<
899         std::setw(20) << "E[X]" <<
900         std::setw(20) << "p_Blocking" <<
901         std::endl;
902
903     system1->SetArrivalRateReduced(ARRIVAL_RATE_REDUCED);
904     system1->SetServiceRate(SERVICE_RATE);
905     system1->SetControlMessageTransferRate(fControlMessageTransferRate);
906
907     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
908
909     // it is difficult to calculate the stationary state probabilities
910     // for this markov chain with SOR. so we start in idle state and let
911     // runge-kutta run for some time to get the stationary state probabilities
912     system1->SetSystemStateProbability(0,1);
913     for (unsigned int i=1; i<system1->GetNumberOfSystemStates(); system1->
        SetSystemStateProbability(i++,0));
914     system1->CalcTransientSystemStateProbabilities_RK4Adapt(0, 50, NULL);
915
```

```
916     // now we start our test
917     system1->CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_START, &Callback);
918     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_HIGH);
919     system1->CalcTransientSystemStateProbabilities_RK4Adapt(TIME_START, TIME_STOP, &Callback)
            ;
920     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
921     system1->CalcTransientSystemStateProbabilities_RK4Adapt(TIME_STOP, TIME_END, &Callback);
922 
923     outfile.close();
924 
925     //system1->PrintSystemStateProbabilities(true);
926 
927     delete system1;
928 }
929 
930 int main()
931 {
932     #ifdef TEST_T001
933         std::cout << "*␣*␣*␣*␣*␣␣M_M_1_S_TwoPointADelay_T001␣␣*␣*␣*␣*␣*" << std::endl;
934         Start( 4, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_04.dat");
935         Start( 8, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_08.dat");
936         Start(12, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_12.dat");
937         Start(16, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_16.dat");
938         Start(20, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_20.dat");
939         Start(24, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_24.dat");
940         Start(28, MESSAGE_RATE, "results/M_M_1_S_TwoPointADelay_T001/
                M_M_1_S_TwoPointADelay_T001_28.dat");
941     #endif
942     #ifdef TEST_T002
943         std::cout << "*␣*␣*␣*␣*␣␣M_M_1_S_TwoPointADelay_T002␣␣*␣*␣*␣*␣*" << std::endl;
944         Start(THRESHOLD_GO, 0.01, "results/M_M_1_S_TwoPointADelay_T002/
                M_M_1_S_TwoPointADelay_T002_A.dat");
945         Start(THRESHOLD_GO, 0.1, "results/M_M_1_S_TwoPointADelay_T002/
                M_M_1_S_TwoPointADelay_T002_B.dat");
946         Start(THRESHOLD_GO,  1, "results/M_M_1_S_TwoPointADelay_T002/
                M_M_1_S_TwoPointADelay_T002_C.dat");
947         Start(THRESHOLD_GO, 10, "results/M_M_1_S_TwoPointADelay_T002/
                M_M_1_S_TwoPointADelay_T002_D.dat");
948     #endif
949 }
950 
951 #endif
952 
953 
954 //***************************************************************************
955 // T003 - absorbing states
956 //***************************************************************************
957 
958 #ifdef TEST_T003
959     #define BLOCK_T003
960 #endif
961 
962 #ifdef BLOCK_T003
963 
964 std::ofstream outfile;
965 
966 //---------------------------------------------------------------------------
967 
968 #define SYSTEMSIZE 4
969 #define MAX_MSG_ON_THE_WAY_A 4
970 #define MAX_MSG_ON_THE_WAY_B 5
971 #define THRESHOLD_STOP 3
972 #define THRESHOLD_GO 1
973 #define TIME_END 20000000
974 #define ARRIVAL_RATE_NORMAL 0.5
975 #define ARRIVAL_RATE_REDUCED 0.1
976 #define SERVICE_RATE 1
```

```
977 #define MESSAGE_RATE 1
978
979 M_M_1_S_TwoPointADelay systemA(SYSTEMSIZE,THRESHOLD_STOP,THRESHOLD_GO,MAX_MSG_ON_THE_WAY_A);
980 M_M_1_S_TwoPointADelay systemB(SYSTEMSIZE,THRESHOLD_STOP,THRESHOLD_GO,MAX_MSG_ON_THE_WAY_B);
981
982
983 int main()
984 {
985     std::cout << "**********␣max␣#␣of␣messages␣is␣even,␣reduced␣arrival␣rate␣!=␣0␣**********"
          << std::endl;
986     systemA.PrintSystemStates();
987     // set the parameters
988     systemA.SetServiceRate(SERVICE_RATE);
989     systemA.SetControlMessageTransferRate(MESSAGE_RATE);
990     systemA.SetArrivalRateReduced(ARRIVAL_RATE_REDUCED);
991     systemA.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL);
992     // we start in idle state
993     systemA.SetSystemStateProbability(0,1);
994     for (unsigned int i=1; i<systemA.GetNumberOfSystemStates(); systemA.
          SetSystemStateProbability(i++,0));
995     // let the system run for a long time
996     systemA.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_END, NULL);
997     systemA.PrintSystemStateProbabilities();
998
999     std::cout << "**********␣max␣#␣of␣messages␣is␣even,␣reduced␣arrival␣rate␣=␣0␣**********"
          << std::endl;
1000    //systemA.PrintSystemStates();
1001    // set the parameters
1002    systemA.SetServiceRate(SERVICE_RATE);
1003    systemA.SetControlMessageTransferRate(MESSAGE_RATE);
1004    systemA.SetArrivalRateReduced(0);
1005    systemA.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL);
1006    // we start in idle state
1007    systemA.SetSystemStateProbability(0,1);
1008    for (unsigned int i=1; i<systemA.GetNumberOfSystemStates(); systemA.
          SetSystemStateProbability(i++,0));
1009    // let the system run for a long time
1010    systemA.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_END, NULL);
1011    systemA.PrintSystemStateProbabilities();
1012
1013    std::cout << "**********␣max␣#␣of␣messages␣is␣odd,␣reduced␣arrival␣rate␣!=␣0␣**********"
          << std::endl;
1014    systemB.PrintSystemStates();
1015    // set the parameters
1016    systemB.SetServiceRate(SERVICE_RATE);
1017    systemB.SetControlMessageTransferRate(MESSAGE_RATE);
1018    systemB.SetArrivalRateReduced(ARRIVAL_RATE_REDUCED);
1019    systemB.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL);
1020    // we start in idle state
1021    systemB.SetSystemStateProbability(0,1);
1022    for (unsigned int i=1; i<systemB.GetNumberOfSystemStates(); systemB.
          SetSystemStateProbability(i++,0));
1023    // let the system run for a long time
1024    systemB.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_END, NULL);
1025    systemB.PrintSystemStateProbabilities();
1026
1027    std::cout << "**********␣max␣#␣of␣messages␣is␣odd,␣reduced␣arrival␣rate␣=␣0␣**********"
          << std::endl;
1028    //systemB.PrintSystemStates();
1029    // set the parameters
1030    systemB.SetServiceRate(SERVICE_RATE);
1031    systemB.SetControlMessageTransferRate(MESSAGE_RATE);
1032    systemB.SetArrivalRateReduced(0);
1033    systemB.SetArrivalRateNormal(ARRIVAL_RATE_NORMAL);
1034    // we start in idle state
1035    systemB.SetSystemStateProbability(0,1);
1036    for (unsigned int i=1; i<systemB.GetNumberOfSystemStates(); systemB.
          SetSystemStateProbability(i++,0));
1037    // let the system run for a long time
1038    systemB.CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_END, NULL);
1039    systemB.PrintSystemStateProbabilities();
1040
1041 }
```

```
1042 #endif
1043
1044
1045 //*****************************************************************************
1046 // T004 - number of customers (absorbing states)
1047 //*****************************************************************************
1048
1049 #ifdef TEST_T004
1050     #define BLOCK_T004
1051 #endif
1052
1053 #ifdef BLOCK_T004
1054
1055 std::ofstream outfile;
1056
1057 //-----------------------------------------------------------------------------
1058
1059 #define SYSTEMSIZE 4
1060 #define MAX_MSG_ON_THE_WAY 3
1061 #define THRESHOLD_STOP 3
1062 #define THRESHOLD_GO 1
1063
1064 #define TIME_START 10
1065 #define TIME_STOP 1000000
1066 #define TIME_END 1000000
1067
1068 #define ARRIVAL_RATE_NORMAL_LOW 0.5
1069 #define ARRIVAL_RATE_NORMAL_HIGH 6
1070 #define ARRIVAL_RATE_REDUCED 0
1071 #define SERVICE_RATE 1
1072 #define MESSAGE_RATE 10
1073
1074 M_M_1_S_TwoPointADelay * system1;
1075
1076 //-----------------------------------------------------------------------------
1077
1078 void Callback(REAL fX)
1079 {
1080     outfile <<
1081         std::setw(20) << fX <<
1082         std::setw(20) << system1->GetNumberOfCustomersMean() <<
1083         std::setw(20) << system1->GetBlockingProbability() <<
1084         std::endl;
1085 }
1086
1087 void Start(char * szFileName)
1088 {
1089     system1 = new M_M_1_S_TwoPointADelay(SYSTEMSIZE,THRESHOLD_STOP,THRESHOLD_GO,
          MAX_MSG_ON_THE_WAY);
1090
1091     std::cout << szFileName << std::endl;
1092     outfile.open(szFileName,std::ios::out);
1093     outfile << "% " << szFileName << std::endl;
1094     outfile << "%" <<
1095         std::setw(19) << "time" <<
1096         std::setw(20) << "E[X]" <<
1097         std::setw(20) << "p_Blocking" <<
1098         std::endl;
1099
1100     system1->SetArrivalRateReduced(ARRIVAL_RATE_REDUCED);
1101     system1->SetServiceRate(SERVICE_RATE);
1102     system1->SetControlMessageTransferRate(MESSAGE_RATE);
1103
1104     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
1105
1106     // it is difficult to calculate the stationary state probabilities
1107     // for this markov chain with SOR. so we start in idle state and let
1108     // runge-kutta run for some time to get the stationary state probabilities
1109     system1->SetSystemStateProbability(0,1);
1110     for (unsigned int i=1; i<system1->GetNumberOfSystemStates(); system1->
          SetSystemStateProbability(i++,0));
1111     system1->CalcTransientSystemStateProbabilities_RK4Adapt(0, 50, NULL);
1112
```

```
1113     system1->PrintSystemStates();
1114     system1->PrintSystemStateTransitionRates();
1115 
1116     // now we start our test
1117     system1->CalcTransientSystemStateProbabilities_RK4Adapt(0, TIME_START, &Callback);
1118     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_HIGH);
1119     system1->CalcTransientSystemStateProbabilities_RK4Adapt(TIME_START, TIME_STOP, &Callback)
         ;
1120     system1->PrintSystemStateProbabilities();
1121 
1122 
1123     system1->SetArrivalRateNormal(ARRIVAL_RATE_NORMAL_LOW);
1124     system1->CalcTransientSystemStateProbabilities_RK4Adapt(TIME_STOP, TIME_END, &Callback);
1125 
1126     outfile.close();
1127 
1128     //system1->PrintSystemStateProbabilities(true);
1129 
1130     delete system1;
1131 }
1132 
1133 int main()
1134 {
1135     #ifdef TEST_T004
1136         std::cout << "*␣*␣*␣*␣*␣␣M_M_1_S_TwoPointADelay_T001␣␣*␣*␣*␣*␣*" << std::endl;
1137         Start("results/M_M_1_S_TwoPointADelay_T004/M_M_1_S_TwoPointADelay_T004.dat");
1138     #endif
1139 }
1140 
1141 #endif
1142 
1143 #endif
```

B. MATLAB-Programme

Listing B.1: MM1S_stationaer.m

```
function MM1S_stationaer()

% M/M/1/S-Warteschlangensystem - stationäre Wahrscheinlichkeiten

% Zu zeichnende Ankunftsraten
lambdas = linspace(0,2,100);

% Berechnung starten
EX = zeros(1,length(lambdas));
pB = zeros(1,length(lambdas));
for k=1:length(lambdas)

    lambda=lambdas(k);
    mu = 1;

    % Matrix der Übergangsraten
    Q = [...
        -lambda lambda 0 0 0 0;...
        mu -lambda-mu lambda 0 0 0; ...
        0 mu -lambda-mu lambda 0 0; ...
        0 0 mu -lambda-mu lambda 0; ...
        0 0 0 mu -lambda-mu lambda; ...
        0 0 0 0 mu -mu ];

    Q1=Q;
    b1 = zeros(1,length(Q1));

    % Die Summe der Wahrscheinlichkeiten ist 1
    Q1(:,end) = ones(length(Q1),1);
    b1(1,end)=1;

    % Berechnen der Zustandswahrscheinlichkeiten
    StatProb = b1/Q1;

    % Berechnen der interessierenden Werte
    EX(k) = sum(StatProb.*(0:length(StatProb)-1));
    pB(k) = StatProb(end);
end

% Bild Anzahl der Anforderungen im System
figure();
plot(lambdas,EX,'b-');
xlabel('\lambda');
ylabel('E[X]');

% Bild Blockierwahrscheinlichkeit
figure();
semilogy(lambdas,pB,'b-');
axis([0 2 1e-6 1])
xlabel('\lambda');
ylabel('p_{blocking}');

end
```

Listing B.2: MM1S_transient.m

```
function MM1S_transient()

% M/M/1/s-Warteschlangensystem - transiente Zustandswahrscheinlichkeiten
```

```
4
5  N = 6;
6
7  lambdas = [0.2];
8  tout = zeros(1,length(lambdas));
9  yout = zeros(1,length(lambdas));
10
11 function [tout, EX, pB] = CalcTransientStateProb(lambda,mu)
12
13     Q = [...
14         -lambda lambda 0 0 0 0;...
15         mu -lambda-mu lambda 0 0 0; ...
16         0 mu -lambda-mu lambda 0 0; ...
17         0 0 mu -lambda-mu lambda 0; ...
18         0 0 0 mu -lambda-mu lambda; ...
19         0 0 0 0 mu -mu ];
20
21     function [dydt] = deriv(t,y)
22         dydt = (y' * Q)';
23     end
24
25     [tout,yout]=ode45(@deriv,[0 25],[1 0 0 0 0 0]);
26     EX = zeros(1,length(tout));
27     for k=1:length(tout)
28         EX(k)=sum(yout(k,:).*(0:5));
29     end
30     pB = yout(:,6);
31
32 end
33
34 [tout1, EX1, pB1] = CalcTransientStateProb(0.2,1);
35 [tout2, EX2, pB2] = CalcTransientStateProb(0.5,1);
36 [tout3, EX3, pB3] = CalcTransientStateProb(0.8,1);
37 [tout4, EX4, pB4] = CalcTransientStateProb(0.95,1);
38
39 % Bild Anzahl der Anforderungen im System
40 figure();
41 plot(tout1,EX1,'b-');
42 hold on
43 plot(tout2,EX2,'b-');
44 plot(tout3,EX3,'b-');
45 plot(tout4,EX4,'b-');
46 xlabel('t');
47 ylabel('E[X]');
48
49 text1 = text('FontSize',9,'Position',[16.62 0.1645],'String','\lambda=0.2');
50 text2 = text('FontSize',9,'Position',[16.73 0.8151],'String','\lambda=0.5');
51 text3 = text('FontSize',9,'Position',[16.68 1.765],'String','\lambda=0.8');
52 text4 = text('FontSize',9,'Position',[16.5 2.248],'String','\lambda=0.95');
53
54 end
```

Listing B.3: MM1S_OnePointS.m

```
1  function MM1S_OnePointS()
2
3  lambda = 0.8;
4  mu1 = 1;
5  mu2 = 2;
6
7  % Berechnen der stationären Wahrscheinlichkeiten
8
9  QS = [...
10 -lambda     lambda      0           0           0       ;...
11 mu1         -lambda-mu1 lambda      0           0       ;...
12 0           mu1         -lambda-mu1 lambda      0       ; ...
13 0           0           mu2         -lambda-mu2 lambda  ; ...
14 0           0           0           mu2         -mu2    ; ...
15 ];
16
17 QS1 = QS;
18 QS1(:,end)=ones(length(QS),1);
19
```

```
b1 = zeros(1,length(QS)); b1(end)=1

StatProb = b1/QS1


function [tout, fout] = CalcFlowTime(lambda,mu1,mu2)

    QF = [...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       0           0       0   ;...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       0           0       0   ;...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       0           0       0   ;...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       0           0       0   ;...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       0           0       0   ;...
        mu1 0   0   0   -mu1-lambda lambda      0           0       0           0 ...
                    0       0           0       0   ;...
        0   mu1 0   0   0   0           -mu1-lambda lambda      0       0           0 ...
                    0       0           0       0   ;...
        0   0   mu2 0   0   0           0           -mu2-lambda lambda  0           0 ...
                    0       0           0       0   ;...
        0   0   0   mu2 0   0           0           0           -mu2    0           0 ...
                    0       0           0       0   ;...
        0   0   0   0   0   mu1         0           0           0       -mu1-lambda lambda ...
                0       0           0       0   ;...
        0   0   0   0   0   0           mu2         0           0       0           -mu2-...
            lambda lambda  0           0       0   ;...
        0   0   0   0   0   0           0           mu2         0       0           0 ...
                    -mu2    0           0       0   ;...
        0   0   0   0   0   0           0           0           0       mu2         0 ...
                    0       -mu2-lambda lambda  0   ;...
        0   0   0   0   0   0           0           0           0       0           mu2 ...
                    0       0           -mu2    0   ;...
        0   0   0   0   0   0           0           0           0       0           0 ...
                    0       mu2         0       -mu2;...
        ]

    function [dfdt] = deriv(t,f)
        dfdt = QF * f;
    end

    [tout,fout]=ode45(@deriv,[0 10],[0 0 0 0 0 1 1 1 1 1 1 1 1 1 1]);
end

[tout1, fout1] = CalcFlowTime(lambda,mu1,mu2);

f1 = ...
    fout1(:,6) *StatProb(1)+...
    fout1(:,10)*StatProb(2)+...
    fout1(:,13)*StatProb(3)+...
    fout1(:,15)*StatProb(4);
f1 = f1 / (1-StatProb(5));  % Es werden nur Anforderungen betrachtet, die vom System ...
    angenommen werden

%figure();
%plot(tout1,fout1(:,6),'b-','LineWidth',3);
%hold on;
%plot(tout1,fout1(:,10),'b-','LineWidth',3);
%plot(tout1,fout1(:,13),'b-','LineWidth',3);
%plot(tout1,fout1(:,15),'b-','LineWidth',3);
%axis([0  7 0 1.1]);

figure();
plot(tout1,fout1(:,6),'b--');
hold on;
plot(tout1,fout1(:,10),'b--');
plot(tout1,fout1(:,13),'b--');
plot(tout1,fout1(:,15),'b--');
plot(tout1,f1,'b-');  % ,'LineWidth',2
axis([0  7 0 1.1]);
```

```
77 xlabel('t');
78 ylabel('W\{T_F>t\}');
79 text1 = text('FontSize',9,'Position',[2.191 0.566],'String','f_{14}');
80 text2 = text('FontSize',9,'Position',[1.704 0.566],'String','f_{12}');
81 text3 = text('FontSize',9,'Position',[1.316 0.566],'String','f_{9}');
82 text4 = text('FontSize',9,'Position',[0.6386 0.566],'String','f_{5}');
83
84
85
86 % Auswirkung von nachkommenden Anforderungen
87 [tout2, fout2] = CalcFlowTime(0,mu1,mu2);
88 [tout3, fout3] = CalcFlowTime(5*lambda,mu1,mu2);
89
90 f2 = ...
91     fout2(:,6) *StatProb(1)+...
92     fout2(:,10)*StatProb(2)+...
93     fout2(:,13)*StatProb(3)+...
94     fout2(:,15)*StatProb(4);
95 f2 = f2 / (1-StatProb(5));  % Es werden nur Anforderungen betrachtet, die vom System
       angenommen werden
96
97 f3 = ...
98     fout3(:,6) *StatProb(1)+...
99     fout3(:,10)*StatProb(2)+...
100    fout3(:,13)*StatProb(3)+...
101    fout3(:,15)*StatProb(4);
102 f3 = f3 / (1-StatProb(5));  % Es werden nur Anforderungen betrachtet, die vom System
       angenommen werden
103
104 figure();
105 plot(tout1,f1,'b-');
106 hold on;
107 plot(tout2,f2,'b-');
108 plot(tout3,f3,'b-');
109 axis([0  7 0 1.1]);
110 xlabel('t');
111 ylabel('W\{T_F>t\}');
112 text1 = text('FontSize',9,'Position',[2.741 0.151],'String','\lambda^* = \lambda');
113 text2 = text('FontSize',9,'Position',[3.504 0.151],'String','\lambda^* = 0');
114 text3 = text('FontSize',9,'Position',[2.106 0.151],'String','\lambda^* = 5\lambda');
115
116
117
118
119 end
```

Listing B.4: MM1S_Ausfall.m

```
1  function MM1S_Ausfall()
2
3  lambda1 = 0.1;
4  lambda2 = 0.2;
5  lambda3 = 0.4;
6
7  mu = 1;
8
9  function [tout, fout] = CalcFlowTime(lambda,mu)
10
11     QF = [...
12         -lambda lambda      0           0           0           0           0       ;...
13         mu      -lambda-mu  lambda      0           0           0           0       ;...
14         0       mu          -lambda-mu  lambda      0           0           0       ;...
15         0       0           mu          -lambda-mu  lambda      0           0       ;...
16         0       0           0           mu          -lambda-mu  lambda      0       ;...
17         0       0           0           0           mu          -lambda-mu  lambda  ;...
18         0       0           0           0           0           0           0       ;...
19         ]
20
21     function [dfdt] = deriv(t,f)
22         dfdt = QF * f;
23     end
24
25     [tout,fout]=ode45(@deriv,[0 50000],[1 1 1 1 1 1 0]);
```

```
26 end
27
28 [tout1, fout1] = CalcFlowTime(lambda1,mu);
29 [tout2, fout2] = CalcFlowTime(lambda2,mu);
30 [tout3, fout3] = CalcFlowTime(lambda3,mu);
31
32
33 figure();
34 semilogx(tout1,fout1(:,1));
35 hold on;
36 semilogx(tout2,fout2(:,1));
37 semilogx(tout3,fout3(:,1));
38
39 xlabel('t');
40 ylabel('W\{T>t\}');
41 axis([0 50000 0.9 1]);
42
43 text1 = text('FontSize',9,'Position',[5 0.98],'String','\lambda=0.4');
44 text2 = text('FontSize',9,'Position',[200 0.98],'String','0.2');
45 text3 = text('FontSize',9,'Position',[12000 0.98],'String','0.1');
46
47 end
```

C. Evolutionärer Algorithmus zum Finden einer Cox-Verteilung

Listing C.1: cox.m

```
function cox(MaxIterations, F)
%COX Annäherung einer gegebenen Wahrscheinlichkeitsverteilung durch eine Cox-Verteilung
%   COX(MaxIterations, F)
%   MaxIterations ... Maximale Anzahl von Iterationen
%   F ... komplementäre Verteilungsfunktion der gewünschten Verteilung [t1 f1; t2 f2; t3 f3; ...]
%   Zur Annäherung wird ein evolutionärer Algorithmus verwendet.
%   Markus Sommereder, 2006-03-10

    %--------------------------------------------------------------------------

    NUMBER_OF_SERVERS =3; % Anzahl der Stufen der Cox-Verteilung

    POPULATION_SIZE = 6;    % Größe der Population
    MULTI = 4;              % Vervielfachung

    MAX_ITERATIONS = 100;   % Default-Wert für die max. Anzahl an Iterationen
    EPS = 1e-6;             % gewünschte Genauigkeit der Cox-Verteilung
                            % (Mittelwert der Abstandsquadrate)

    NUMBER_OF_PARENTS = 2;  % Anzahl der Eltern [1,POPULATION_SIZE]
    GROUP_SIZE = 0.1;       % Gruppengröße für Selektion / Selektionsdruck (0,1]
                            % Default: 0.1
    MUTATION = 0.4;         % Anzahl von Mutationen [0,1]. Default: 0.4

    VISUALIZATION = 1;      % 1: Evolution zeichnen, 0: nicht zeichnen

    %--------------------------------------------------------------------------

    if nargin < 1, MaxIterations = MAX_ITERATIONS; end

    if nargin < 2
        F = [0 1; 1 0.95; 2 0.8; 3 0.6; 4 0.4; 5 0.25; 6 0.15; 7 0.09; 8 0.04; ...
             9 0.02; 10 0.01];
    end

    % Zufallszahlengenerator initialisieren
    rand('state',sum(100*clock))

    VectorSize = NUMBER_OF_SERVERS*2-1;
    PopulationSizeNew = POPULATION_SIZE*MULTI;
    Visualization = VISUALIZATION && (POPULATION_SIZE <= 6) && (MULTI <= 5);

    % Initialisierung der Population
    Population = rand(VectorSize,POPULATION_SIZE);
    PopulationNew = zeros(VectorSize,PopulationSizeNew);

    % Zeichnung der aktuellen Population vorbereiten
    if Visualization
        figure;
        h = zeros(PopulationSizeNew,1);
        for i=1:POPULATION_SIZE
            for j=1:MULTI
                h((i-1)*MULTI+j) = subplot(MULTI,POPULATION_SIZE,(i-1)*MULTI+j);
            end
        end
```

```
    end

    % Evolution starten
    BestFitness = inf;
    BestFitnessHistory = zeros(1,MAX_ITERATIONS);
    IterationCount = 0;

    while (BestFitness > EPS) && (IterationCount < MaxIterations)

        IterationCount = IterationCount + 1;
        disp(['Anzahl Iterationen: ' num2str(IterationCount)]);

        % Nachkommen erzeugen

        % Die alten Polulation wird übernommen
        PopulationNew(:,1:POPULATION_SIZE)=Population;

        % Der Rest besteht aus Nachkommen
        for i=POPULATION_SIZE+1:PopulationSizeNew
            % Eltern auswählen
            Parents=ceil(POPULATION_SIZE*rand(1,NUMBER_OF_PARENTS));
            % Merkmale auswählen
            for j=1:VectorSize
                PopulationNew(j,i)=Population(j,Parents(ceil(NUMBER_OF_PARENTS*rand)));
            end
        end


        % Mutation
        for i=1:ceil(VectorSize*PopulationSizeNew*MUTATION);
            a=ceil(VectorSize*rand);
            b=ceil(PopulationSizeNew*rand);
            r = rand-0.5;
            if mod(a,2)==0      % Wahrscheinlichkeit
                if r > 0
                    m = min([0.1 1-PopulationNew(a,b)]);
                else
                    m = min([0.1 PopulationNew(a,b)]);
                end
                PopulationNew(a,b) = PopulationNew(a,b)+r*m;
            else                % Bedienrate
                PopulationNew(a,b) = PopulationNew(a,b)*(1+(rand-0.5)/10);
            end
        end

        % Fitness berechnen
        fitness = zeros(1,POPULATION_SIZE*MULTI);
        for k=1:POPULATION_SIZE*MULTI
            A = CreateMatrixForCox(PopulationNew(:,k));
            [fitness(k), tout, yout] = CalculateFlowTime(A,F);
            if Visualization
                if fitness(k) < BestFitness
                    plot(h(k),tout,yout,'r-',F(:,1),F(:,2),'.');
                else
                    plot(h(k),tout,yout,'b-',F(:,1),F(:,2),'.');
                end
                title(h(k),num2str(fitness(k)));
            end
        end

        % Selektion

        % Population nach der Fitness sortieren, beste Fitness merken
        [f,index]=sort(fitness);
        BestFitness=f(1);
        BestFitnessHistory(IterationCount)=BestFitness;

        A = 1; % Das beste Individuum wird immer übernommen
        while length(A) < POPULATION_SIZE
            GroupSize = ceil(GROUP_SIZE*PopulationSizeNew);
            Group = sort(ceil(PopulationSizeNew*rand(1,GroupSize)));
            for j=1:GroupSize
                if max(ismember(A,Group(j)))==0
```

```
                    A = union(A,Group(j));
                    break;
                end
            end
        end
        Population=PopulationNew(:,index(A));

        % Zeichnung auffrischen
        drawnow;
    end

    % Zeichnung Fitness-Funktion
    figure;
    plot(1:IterationCount,BestFitnessHistory(1:IterationCount),'.');
    title('Verlauf der Fitness-Funktion','FontSize',12);

    % Zeichnung Annäherung
    figure;
    k=index(1);
    A = CreateMatrixForCox(PopulationNew(:,k));
    [fitness(k), tout, yout] = CalculateFlowTime(A,F);
    plot(tout,yout,'b-',F(:,1),F(:,2),'.');
    xlabel('t');
    ylabel('W \{T_F \geq t\}');
    title('Beste Annäherung','FontSize',12);

    % Werte für die beste Annäherung ausgeben
    for i=1:NUMBER_OF_SERVERS-1;
        disp(['Bedienrate ' num2str(i) ' = ' num2str(PopulationNew(2*i-1,k))]);
        disp(['Übergangswahrscheinlichkeit ' num2str(i) '-' num2str(i+1) ...
                ' = ' num2str(PopulationNew(2*i,k))]);
    end
    disp(['Bedienrate ' num2str(NUMBER_OF_SERVERS) ' = ' ...
                    num2str(PopulationNew(2*NUMBER_OF_SERVERS-1,k))]);
    disp(['Fitness = ' num2str(BestFitness)]);

    %-------------------------------------------------------------------------

    function [Q] = CreateMatrixForCox(values)
        % Q: Matrix der Übergangsraten
        % values: Werte für die Cox-Verteilung:
        %         [mu(1) p(1,2) mu(2) p(2,3) ... mu(n-1) p(n-1,n) mu(n)]

        Q = zeros(NUMBER_OF_SERVERS);
        for i=1:NUMBER_OF_SERVERS
            Q(i,i)=-values(2*i-1);
            if i<NUMBER_OF_SERVERS
                Q(i,i+1)=values(2*i-1)*values(2*i);
            end
        end
    end

    %-------------------------------------------------------------------------

    function [d, t, y1] = CalculateFlowTime(Q, F)
        % Q: Matrix
        % F: Gewünschte Durchflusszeitverteilung
        % d: Distanz der berechneten zur gewünschten
        %    Durchflusszeitverteilung (Durchschnitt der Abstandsquadrate)
        % tout, yout: Werte der berechneten Durchflusszeitverteilung

        STEPSIZE = 0.2;
        TSTOP = max(F(:,1));

        [m n]=size(Q);
        if m ~= n, error('Matrix Q ist nicht quadratisch'); end
        y=ones(m,1);

        function [dydx] = deriv(x,y)
            dydx = zeros(length(y),1);
            dydx = Q * y;
        end

```

```
202         [t, y] = ode45(@deriv,[0,TSTOP],y,STEPSIZE);
203         y1 = y(:,1);
204 
205         % Fehler bestimmen
206         d = 0;
207         for i=1:length(F)
208             d = d + (interp1(t,y1,F(i,1)) - F(i,2))^2;
209         end
210         d = d / length(F);
211     end
212 
213 end
214 
215 %*************************************************************************
```

Literaturverzeichnis

[Stewart 1994] William J. Stewart: *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[van As 1984] Harmen R. van As: *Modellierung und Analyse von Überlast-Abwehrmechanismen in Paketvermittlungsnetzen.* Dissertation, Universität Siegen, 1984.

[Kühn 1972] Paul Kühn: *Über die Berechnung der Wartezeiten in Vermittlungs- und Rechnersystemen.* Dissertation, Universität Stuttgart, 1972.

[Viertl 2003] Reinhard Viertl: *Einführung in die Stochastik. Mit Elementen der Bayes-Statistik und der Analyse unscharfer Information.* Springer, 2003.

[Allen 1990] Arnold O. Allen: *Probability, Statistics and Queueing Theory.* Academic Press, 1990.

[Hager/Mathar/Mattfeldt 1995] Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: *Intelligent Cruise Control and Reliable Communication of Mobile Stations.* IEEE Transaction on Vehicular Technology, Vol. 44 No. 3, 1995. (S. 443–448)

[Hazeghi 2006] Kasra Hazeghi: *Warteschlangen-Modelle.* ETH Zürich, 2006.

[Dietrich/Stahl 1973] Günter Dietrich, Henry Stahl: *Matrizen und Determinanten und ihre Anwendung in Technik und Ökonomie.* Fachbuchverlag Leipzig, 1973.

[Meister 2005] Andreas Meister: *Numerik linearer Gleichungssysteme. Eine Einführung in moderne Verfahren.* Vieweg Verlag, Wiesbaden, 2005.

[Knorrenschild 2005] Michael Knorrenschild: *Numerische Mathematik. Eine beispielorientierte Einführung.* Fachbuchverlag Leipzig, 2005.

[Herzog/Woo/Chandy 1975] U. Herzog, L. Woo, K. M. Chandy: *Solution of Queueing Problems by a Recursive Technique.* IBM Journal of Research and Development 19, 1975. (S. 295–300)

[Press et al. 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[Fehlberg 1969] Erwin Fehlberg: *Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems.* NASA Technical Report 315, 1969.