Eigenanalysis Solution for Quasi Birth and Death Process.

by

Nikhil Jain

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Industrial and Systems Engineering

APPROVED

Prof. John N. Daigle, Chairman:

Prof. Joel A. Nachlas:

Prof. Hanif D. Sherali:

April, 1991

Blacksburg, Virginia
Eigenanalysis solution for Quasi Birth and Death Process.

by

Nikhil Jain

Committee Chairman: Professor John N. Daigle
Industrial and Systems Engineering

Abstract

The behavior of many systems of practical interest in communications and other areas is well modeled by a single server exponential queueing system in which the arrival and service rates are dependent upon the state of a Markov chain, the dynamics of which are independent of the queue length. Formal solution to such models based on Neuts's matrix geometric approach have appeared frequently in the literature. A major problem in using the matrix geometric approach is the computation of the rate matrix, which requires the solution of a matrix polynomial. In particular, computational times appear to be unpredictable and excessive for many problems of practical interest.

Alternative techniques which employ eigenanalysis have been developed. These techniques are polynomially bounded and yield results very quickly compared to iterative routines. On the other hand, the class of systems to which the eigenanalysis based techniques apply have been somewhat restricted. In this thesis, we modify the eigenanalysis approach initially presented in order to remove some of these restrictions.
Acknowledgements

I would like to express my deepest appreciation to Professor John N. Daigle, my advisor at Virginia Polytechnic Institute and State University, for his guidance and encouragement. I would also like to thank my committee for their input. Finally, I thank my family for their love and support.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>iii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Determination of the Rate Matrix</td>
<td>10</td>
</tr>
<tr>
<td>3. Eigenanalysis-based Determination of Rate Matrix for General Systems</td>
<td>20</td>
</tr>
<tr>
<td>4. Example: Cellular Telephone Access System</td>
<td>25</td>
</tr>
<tr>
<td>5. Conclusions</td>
<td>39</td>
</tr>
<tr>
<td>Bibliography</td>
<td>41</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
</tr>
<tr>
<td>A. Matrix Geometric Routines</td>
<td>45</td>
</tr>
<tr>
<td>B. Modified Eigenanalysis Routines</td>
<td>57</td>
</tr>
<tr>
<td>Vita</td>
<td>75</td>
</tr>
</tbody>
</table>
1. Overview

The queueing behavior of numerous communication and manufacturing systems is well modeled by a quasi birth and death model. That is, the system is well modeled by a continuous time Markov chain for which the infinitesimal generator has the block tri-diagonal form

\[
\tilde{Q} = \begin{pmatrix}
B_0 & A_0 & 0 & \ldots & \ldots & \\
B_1 & A_1 & A_0 & 0 & \ldots & \\
0 & A_2 & A_1 & A_0 & 0 & \\
0 & 0 & A_2 & A_1 & A_0 & \\
\ldots & \ldots & \ldots & \ldots & \ldots & \\
\end{pmatrix},
\]

where \(A_i\) and \(B_i\) are \((K + 1)\)-square real matrices. In this case the Markov chain reduces to the form \(\{ (\bar{n}_t, \bar{\phi}_t), \ t \geq 0 \}\), where \(\bar{n}_t\) and \(\bar{\phi}_t\) are defined as the number present in the system and the phase of the system, respectively, at time \(t\). This Markov chain is a special case of a Markov chain of the G/M/1 type as defined in Neuts [5].

In general, the state of the Markov chain at time \(t\) is given by the ordered pair \((n, j)\), \(n \geq 0\) and \(0 \leq j \leq K\). For notational convenience, it is customary to partition the equilibrium probabilities for the Markov Chain in vector form as follows:

\[
P_n = \lim_{{t \to \infty}} \{ P\{ \bar{n}_t = n, \bar{\phi}_t = 0 \}, P\{ \bar{n}_t = n, \bar{\phi}_t = 1 \}, \ldots, P\{ \bar{n}_t = n, \bar{\phi}_t = K \} \},
\]

and refer to \(P_n\) as the level \(n\) probability vector. The stationary probability vector for the system is then denoted by the infinite vector

\[
P = [P_0, P_1, P_2, \ldots]
\]

The equilibrium probabilities for the Markov chain can then be written in the form [5]

\[
P_n = P_0 R^n,
\]
where $R$ is the matrix of constants whose spectral radius is strictly less than unity. It follows from the above equation that the phase-occupancy masses are geometrically decreasing, but the geometric parameter is a matrix, denoted by $R$.

The matrix $R$ is called the rate matrix for the system. Its dimension is $(K + 1) \times (K + 1)$, and the element $r_{i,m}$ of $R$ represents the expected number of visits to the state $(i + 1, m)$ starting from state $(i, l)$ before returning to level $i$. The matrix $R$ is a fundamental descriptor for Markov chains of the G/M/1 type; a knowledge of $R$ allows one to compute not only occupancy distributions, but other quantities of interest such as waiting times[1].

Historically, the rate matrix has been computed using iterative routines. A detailed overview
of available techniques is provided in Gün[27]; some important ones are listed below. Successive substitutions (SS), has an R-linear convergence rate and can be extremely slow for traffic intensities close to unity. Modified SS is about twenty percent faster than SS. The Newton-Kantorovich (NK) method has a faster rate of convergence than either of the above methods, but the computations involved at each iteration are considerable. On the whole, the NK method performs worse than the SS methods. The modified NK scheme out-performs the SS method and results in a fifty to seventy percent reduction in computational times. The matrix $R$ can also be calculated by solving a quadratic system of equations, by spectral decomposition or by nonlinear optimization techniques; see [28] and [29]. Neuts has alluded to these possibilities in [5].

The SS technique may be preferred in some cases due to the following reasons. First, the results for the QBD process can be put into a concise and explicit form. Second, it is possible to easily implement algorithms that give numerically stable results. However, the complexity of the SS technique is not polynomially bounded.

Recently, an alternative technique that employs eigenvalues and eigenvectors has been developed. This technique is polynomially bounded and yields results very quickly compared to iterative routines[1]. Henceforth, we will refer to this technique as the eigenanalysis technique. In this thesis we focus on the eigenanalysis technique of calculating the rate matrix. We develop a modification to the initial version of the technique. The modification will open its application to a wider class of systems.

In the remainder of this chapter, we review three examples drawn from the literature that illustrate the applicability of the QBD model to analyzing engineering problems. Then, we review other related work from the literature, and finally provide a brief overview of the remainder of the thesis.

2. Applications

In this section, we describe three settings in which systems have been modeled as a QBD process to obtain useful engineering results. Two of these applications are from communications and one is
2.1 A Model of a Production line with Intermediate Buffer and Station Breakdown

In [23], a production line with serial arrangement of work stations is analyzed. Items pass through successive stations with specific operations being performed at each station. Please refer to figure 1.2.

A cause of production line inefficiency is station breakdowns. When a station breaks down, its successor may be forced down or starved since its source of work is not operational. Similarly, its predecessor may be forced down or blocked since its destination may be unable to remove the processed items from their allocated storage areas. Interstation storage buffers are used between stations to reduce the interference between them. However, finite capacity buffers cannot entirely guarantee independence among the various stations. Blocking occurs when a station output buffer becomes full even if the successor station is operational, and starving occurs when a station empties its input buffer even though its predecessor is still operational.

In [23], a production line with two unreliable stations and a finite intermediate buffer is modeled using Markov chains. The number of elements in the buffer is analogous to the number in the queue. The status of the two stations on either side of the buffer determines the different states of the phase process. For example, one state corresponds to the case in which both machines are broken, another
corresponds the case in which one machine is working and the other broken down.

The number of possible states depends on the real system that is being modeled. In [23] systems with a maximum of twenty phases are considered. The buffer occupancy is determined for a specific set of parameters, and the results are used to draw conclusions about the effect of buffer size on production rate.

2.2 Queueing-Blocking System with Two Arrival Streams and Guard Channel

In [15], the operation of a cellular telephone system is analyzed. In a cellular telephone system, mobile subscribers are provided with telephone service within a geographical area. The geographical area is divided into cells. Each cell has a communication equipment called the cell service unit, which communicates with the mobile unit using one of the numerous frequency channels allocated to it. When a mobile unit moves from an area serviced by one cell unit to another, the call is handed over to the cell service unit of the new cell. The new service unit then allocates a frequency channel from among its free channels. If the new service unit does not have a free frequency channel then the call is terminated. It is desirable to avoid call termination of this nature whenever possible.

Numerous protocols have been designed to limit the probability of interrupting an ongoing telephone call. One such protocol, discussed below, is examined in [15]. The cell service unit handles two type of calls: calls that originate in the cell, and those that are transferred from other cells. The calls that originate within the cell are queued if more than a certain number of channels, denoted by $g$, are occupied by ongoing calls. These queued calls are not established until there are $g$ channels free. In order words, the service unit must have a certain number of channels free before it will complete a new call originating from its own cell.

In order to design the cell service unit, two important engineering specifications have to be determined: the total number of frequency channels that would be required at each unit, and the number of channels reserved ($g$) exclusively for calls that are transferred from other cells. The problem can be modeled using a QBD process. The state of the system is represented by the ordered pair $(i, j)$, where $i$ is the number of call completion requests in the queue, and $j$ is the
Figure 1.3 The state diagram for the cellular telephone access system.
number of calls in progress. The state of the phase process is defined as the number of channels of
the service unit that are being used and is denoted by \( j \). The model is depicted in figure 1.3. In the
system calls arrive at rate \( \lambda \), are transferred into the cell from other adjoining cells at rate \( \gamma \), and
are completed at rate \( \mu \). In the paper, the effect of changing the number of guard channels on the
probability of blocking is studied. Systems having as many as 100 channels are analyzed.

2.3 Queueing Analysis of a Packet Voice Communication System

In packet switched networks, data and voice share the same channel. This ability to share the
channel has made packet switched technology very attractive. To transmit voice, a voice source
is first digitized. The resulting bits are then grouped together as packets. Finally, the packets are
presented to the network for transmission. In order for the voice to be transmitted across the channel
effectively, the packets must arrive at the destination within a certain time limit. This places a very
tight restriction on the amount of delay that can be allowed for the voice packets.

In normal conversation, each party alternates between the "active" and "inactive" modes. When
in active mode, a party is actually talking, while in inactive mode the party is silent. When the
party is active, voice packets are generated and during the inactive mode no packets are generated.
This behavior is shown in figure 1.4.

In a system that transmits voice and data on the same link, packets from the voice and data
sources have to be accepted and presented to the link in an orderly fashion. One scheme to do this
is statistical multiplexing. Packets from all the voice and data sources are queued together in a
common queue. The link then transmits the packets on a first-come-first-served basis. A system of
this type is depicted in figure 1.5.

In the model described in [22], each voice source generates a packet with the deterministic
intergeneration time of \( 1/V \) seconds while active. The packets arrive instantaneously at the queue,
and each packet requires a deterministic service time of \( 1/VC \) seconds. Thus, \( C \) active sources
would just saturate the server. However the total number of voice sources that may be multiplexed
onto the link, \( N \), may be larger than \( C \). Hence, a backlog of packets is created during periods when
Figure 1.4 Typical behavior of voice sources.

more than $C$ voice sources are active, and the backlog is reduced in periods when less than $C$ voice sources are active.

Figure 1.5 Packet voice statistical multiplexing system

The queueing process for the statistical multiplexer is modeled as a semi Markov process (SMP). The matrix geometric method is used to solve for the rate matrix. The $R$ matrix is then used to
study the behavior of queues in the packet voice statistical multiplexer. In the paper systems with twenty voice sources are analyzed.

3. Other application of the QBD model

Other applications of the QBD process in manufacturing and computer communication systems are now indicated. In [11], a continuous time Markov chain is used to find the delay characteristics for the Carrier Sense Multiple Access/Collision Detect (CSMA/CD) protocol. In [14], performance characteristics of a distributed computing system are studied using a QBD process with 4 phases. In [16], a QBD process is used to model a finite population CSMA/CD network and an efficient recursive algorithm to determine its performance is presented. In manufacturing, [18] presents an analysis of the (R,r) continuous review inventory policy, and in [19], a wait repair problem is discussed.

4. Thesis organization

This thesis comprises five chapters. Chapter 2 describes the matrix geometric method and the eigenanalysis method of solving for the stationary distribution of a QBD process. Chapter 3 presents original work, which is an extension of the methodology presented in Chapter 2. In Chapter 4, we solve a problem from the literature as an example application. Chapter five presents conclusions. In the appendix we present a listing of the computer program that implemented the algorithm developed in Chapters 2 and Chapter 3.
Chapter 2

Determination of the Rate Matrix

1. Overview

We indicated in Chapter 1 that the rate matrix of a quasi birth and death (QBD) process is an important and useful characteristic upon which occupancy and waiting time analysis can be based. This chapter describes some of the important methods available for determining the rate matrix. We first introduce terminology and set up an efficient notation through with which to express the balance equations of the QBD process. Next we describe the matrix geometric solution technique. We then present a method of determining the rate matrix based on eigenanalysis.

2. Preliminaries

In this section, we present the balance equations for the QBD process discussed in Chapter 1. Such a QBD process can be represented by the state diagram shown in figure 2.1.

If the states included in the rectangle A of figure 2.1 are considered in isolation they represent the states of a continuous time, discrete state Markov chain, henceforth referred to as the phase process of the QBD. The phase process governs the arrival and the service rates of the system. That is, if the phase process is in state $k$ then the interarrival and the service times are exponentially distributed with parameters $\lambda_k$ and $\mu_k$, respectively.

In the case considered in this chapter, the phase process is a birth and death process. Thus if the phase process is in state $k$, then times between transition to phase $k - 1$ and $k + 1$ are distributed exponentially with parameters $\delta_k$ and $\beta_k$, respectively. The transition rates to other states from phase $k$ are zero.

We begin deriving the balance equations by first writing the balance equation for the interior of the chain. The interior states are states for which $i > 0$ and $0 < j < K$. Figure 2.2 is a partial state diagram representing an interior state $(i, j)$. Let $P_{i,j}$ for $i > 0$ and $0 < j < K$, be the stationary
Figure 2.1 Quasi Birth Death Process.

The probability that the chain is in state \((i,j)\). The balance equations for interior states have the form

\[
(\lambda_j + \mu_j + \beta_j + \delta_j)P_{i,j} = \lambda_j P_{i-1,j} + \beta_{j-1} P_{i,j-1} + \mu_j P_{i+1,j} + \delta_{j+1} P_{i,j+1},
\]

as can be seen from figure 2.2.

We will now consider the state corresponding to the left upper boundary of the system, namely the state \((0,0)\). From this state there are transitions to states \((0,1)\) and \((1,0)\) only. This implies that \(\mu_0\), \(\delta_0\) and \(\beta_1\) are each zero. The states \((-1,0)\) and \((0,-1)\) are not defined therefore \(P_{-1,0}\) and \(P_{0,-1}\) are zero. Substituting a zero for \(i\) and \(j\) and equating \(\mu_0\), \(\delta_0\), \(\beta_1\), \(P_{-1,0}\) and, \(P_{0,-1}\) to zero in
Figure 2.2  Partial State diagram of the QBD process.

equation (2.1) leads to

\[(\lambda_0 + \beta_0) P_{0,0} = \delta_1 P_{0,1} + \mu_0 P_{1,0}.\]  \hspace{1cm} (2.2)

Now we consider the states on the left boundary of the state diagram; that is, the states of the form \((0, j)\) for \(0 < j < K\). The system transitions from \((0, j)\) into states \((0, j + 1)\) and \((0, j - 1)\), and vice versa. In this case \(\mu_j\) is zero. The states \((-1, j)\) for \(0 < j < K\) are undefined, therefore \(P_{-1,j}\) is zero. The balance equations for this set of states can be written by substituting 0 for \(i\) in equation (2.1) and zero for \(\mu_0\) and \(P_{-1,j}\) in the result. This leads to

\[(\lambda_j + \beta_j + \delta_j) P_{0,j} = \beta_{j-1} P_{0,j-1} + \delta_{j+1} P_{0,j+1} + \mu_j P_{1,j}.\]  \hspace{1cm} (2.3)

We now consider the states that are along the lower boundary of the state diagram, states of the form \((i, K)\) for \(i > 0\). In this case the chain can transition into states \((i + 1, K)\), \((i - 1, K)\) and \((i, K - 1)\) only. In this situation the values of \(\delta_{K+1}\) and \(\beta_K\) are zero. Substituting these values in equation (2.1) leads to

\[(\lambda_K + \delta_K + \mu_K) P_{i,K} = \beta_{K-1} P_{i,K-1} + \mu_K P_{i+1,K} + \lambda_K P_{i-1,K}.\]  \hspace{1cm} (2.4)
The equations for the states \((i,0)\) for \(i > 0\), that is, the left boundary, and the state \((0,K)\) can be written as
\[
(\lambda_0 + \mu_0 + \beta_0)P_{i,0} = \lambda_0 P_{i-1,0} + \mu_0 P_{i+1,0} + \delta_1 P_{i,1}, 
\]
(2.5)
and
\[
(\lambda_K + \delta_K)P_{0,K} = \beta_{K-1} P_{1,K-1} + \mu_K P_{1+1,K}, 
\]
(2.6)
respectively. Rewriting equations (2.1) through (2.6) in matrix form leads to an intuitively satisfying interpretation. Towards this end, we define \(M\), \(\Lambda\) and \(Q\), each \((K+1)\)-square matrices, as follows:

\[
\Lambda = \text{diag} (\lambda_0, \lambda_1, \ldots, \lambda_k)
\]
\[
M = \text{diag} (\mu_0, \mu_1, \ldots, \mu_k)
\]
\[
Q = \begin{pmatrix}
-\beta_0 & \beta_0 & 0 & \cdots & \cdots & \cdots \\
\delta_1 & -(\delta_1 + \beta_1) & \beta_1 & 0 & \cdots & \cdots \\
0 & \delta_2 & -(\delta_2 + \beta_2) & \beta_2 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
0 & 0 & 0 & \cdots & \delta_k & -\delta_k
\end{pmatrix}
\]
Let the vector \(P_n\) denote the joint occupancy distribution at level \(n\) for \(n \geq 0\) and be defined as follows:
\[
P_n = [P_{n0} \ P_{n1} \ \cdots \ P_{n,K}].
\]
Also, define the row vector
\[
P = [P_1 \ P_2 \ P_3 \ \cdots].
\]
The equations (2.2), (2.3) and (2.6) correspond to the zero level. These equations can be written in the compact form
\[
P_0 (\Lambda - Q) - P_1 M = 0.
\]
(2.7)
The equations (2.1), (2.4) and (2.5) represent equations for levels greater than zero. These equations can be written in the compact form
\[
P_{n-1}\Lambda - P_n (\Lambda - Q + M) + P_{n+1} M = 0.
\]
(2.8)
Thus all the balance equations (2.1), (2.2), (2.3), (2.4), (2.5) and (2.6) can be represented by matrix equations (2.8) and (2.7).

We note in passing that the matrix $Q$ represents the generator of the phase process. If the phase process is changed from a birth death process to a more generic continuous time Markov chain all equations will remain unchanged; only the matrix $Q$ has to be redefined. In Chapter 3, we will represent the solution for the QBD process having a more general $Q$.

3. Matrix Geometric Solution

In 1981 Neuts[5] has shown that a QBD process on the state space $E = \{(i, j), i \geq 0, 1 \leq j \leq m\}$ with a infinitesimal generator $\tilde{Q}$, given by

$$\tilde{Q} = \begin{pmatrix}
B_0 & A_0 & 0 & \cdots & \cdots & \\
B_1 & A_1 & A_0 & 0 & \cdots & \\
0 & A_2 & A_1 & A_0 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots
\end{pmatrix},$$

where $B_0 e + A_0 e = B_1 e + A_1 e + A_0 e = (A_0 + A_1 + A_2)e = 0$, is positive recurrent if and only if (Theorem 3.1.1 Neuts 1981 [5]) the minimal nonnegative solution $R$ to the matrix equation

$$R^2 A_2 + RA_1 + A_0 = 0,$$

has all its eigenvalues inside the unit disc. The stationary probability vector $P = [P_0 \quad P_1 \quad P_2 \quad \ldots]$ of $\tilde{Q}$ is given by

$$P_n = P_0 R^n = P_{n-1} R \quad \text{for} \quad n \geq 0. \quad (2.9)$$

In terms of the balance equations derived in the previous section and the matrices defined therein, the matrices $A_2$, $A_1$, $A_0$, $B_0$ and $B_1$ can be written as $A_1 = -(\Lambda - Q + M)$, $A_2 = \Lambda$, $A_0 = M$, $B_0 = -M$ and $B_1 = (\Lambda - Q)$.

If we substitute for $P_n$ from equation (2.9) into equation (2.8) we obtain

$$P_{n-1} \Lambda - P_{n-1} R(\Lambda - Q + M) + P_{n-1} R^2 M = 0, \quad \text{for} \quad n \geq 1.$$

Factoring $P_{n-1}$, we find

$$P_{n-1}[\Lambda - R(\Lambda - Q + M) + R^2 M] = 0, \quad \text{for} \quad n \geq 1. \quad (2.10)$$
A sufficient condition for equation (2.10) to hold is

\[ \Lambda - R(\Lambda - Q + M) + R^2 M = 0. \tag{2.11} \]

The matrix \( R \) can be determined by successive substitution. This can be done by first rewriting (2.11) as

\[ R = (\Lambda + R^2 M)(\Lambda - Q + M)^{-1}, \]

and then using this expression as the basis for the recursion

\[ R_j = \Lambda(\Lambda - Q + M)^{-1} + R_{j-1}^2 M(\Lambda - Q + M)^{-1}. \tag{2.12} \]

The recursion is initialized by using \( R_0 = 0 \). Successive approximations to \( R \) are then computed.

Neuts[5] has shown that the sequence \( \{R_0, R_1, R_2, R_3, \ldots \} \) is monotonically increasing and that it converges to the minimal nonnegative solution to (2.11), which uniquely provides the rate matrix \( R \). The convergence rate of the above sequence is R-linear[27].

4. Eigenanalysis Approach

The eigenanalysis technique of solving equation (2.11) for \( R \) was examined by Daigle and Lucantoni in 1989[1]. The computational effort in this technique is polynomially bounded and is numerically stable [1]. This technique uses efficient commercially available algorithms to determine eigenvectors and eigenvalues of a \( 2(K + 1) \)-square matrix and uses these eigenvalues and eigenvectors, in turn, to calculate \( R \).

Define the vector \( V_v \) to be the left eigenvector of \( R \) corresponding to its eigenvalue \( v \). That is, let

\[ V_v R = V_v v. \tag{2.13} \]

Since the right side of equation (2.11) is zero, multiplying the left and the right side of (2.11) by \( V_v \) yields

\[ V_v [\Lambda - R(\Lambda - Q + M) + R^2 M] = 0. \tag{2.14} \]
This equation can be written as

\[ [V_v A - V_v R(A - Q + M) + V_v R^2 M] = 0 \]

and simplified using (2.13) to give

\[ V_v [\Lambda - v(\Lambda - Q + M) + v^2 M] = 0. \]

Let \( v = 1/\sigma \) for \( \sigma \neq 0 \). Then, the above equation becomes

\[ V_v [\sigma^2 \Lambda - \sigma(\Lambda - Q + M) + M] = 0. \]  \hspace{1cm} (2.15)

If we define

\[ A(z) = \Lambda z^2 - (\Lambda - Q + M)z + M, \]

then equation (2.15) can be written as

\[ V_v A(\sigma) = 0. \]

Upon taking transpose of equation (2.15) we obtain

\[ [\sigma^2 \Lambda - \sigma(\Lambda - Q + M) + M]^T V_v^T = 0, \]

or equivalently,

\[ [\sigma^2 \Lambda - \sigma(\Lambda - Q^T + M) + M] V_v^T = 0. \]  \hspace{1cm} (2.16)

Equation (2.16) is a \((K + 1)\)-square matrix of quadratic equations, called a second order \( \lambda \)-matrix. It can be converted into a \(2(K + 1)\)-square matrix of linear equations in \( \sigma \). To this end we let \( Y = \sigma V_v^T \)
and substitute this expression into equation (2.16). This substitution yields the following system of equations, which is linear in \( \sigma \):

\[ \sigma \Lambda Y - [\sigma(\Lambda - Q^T + M) + M] V_v^T = 0 \]

and

\[ \sigma V_v^T = Y. \]
The above set of matrix equations can be written as a single matrix equation as follows:

\[
\begin{bmatrix}
M & 0 & I \\
0 & I & \\
I & 0 & \\
\end{bmatrix} - \sigma \begin{bmatrix}
(A - Q^T + M) & -A \\
I & 0 \\
\end{bmatrix} \begin{bmatrix}
V_u^T \\
Y \\
\end{bmatrix} = 0.
\]  
(2.17)

The equation (2.17) is analogous to

\[
[B - \sigma A][X] = 0,
\]  
(2.18)

where \(A, B\) and \(X\) are given by:

\[
A = \begin{bmatrix}
A - Q^T + M & -A \\
I & 0 \\
\end{bmatrix},
\]

\[
B = \begin{bmatrix}
M & 0 \\
0 & I \\
\end{bmatrix},
\]

and

\[
X = \begin{bmatrix}
V_u^T \\
Y \\
\end{bmatrix}.
\]

Define \(X = B^{-1}\tilde{X}\). Then, upon substitution for \(X\) in equation (2.18) we find:

\[
[I - \sigma AB^{-1}][\tilde{X}] = 0,
\]

where

\[
AB^{-1} = \begin{bmatrix}
[A - Q^T + M]M^{-1} & -A \\
M^{-1} & 0 \\
\end{bmatrix}.
\]  
(2.19)

In equation (2.18), \(1/\sigma\) is an eigenvalue of \(AB^{-1}\) and \(\tilde{X}\) is the right eigenvector of \(AB^{-1}\) corresponding to the eigenvalue \(1/\sigma\). The vector \(X\) can then be calculated using \(X = B^{-1}\tilde{X}\). Define \(N\) to be the diagonal matrix of eigenvalues. That is, let

\[
N = \begin{bmatrix}
v_0 & 0 & 0 & 0 & 0 \\
0 & v_1 & 0 & 0 & 0 \\
0 & 0 & v_2 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & v_K
\end{bmatrix},
\]

where \(v_i = 1/\sigma_i\) for \(i = 0, 1, 2, \ldots, K\). Let \(V\) be a matrix defined as

\[
V = [V_{v0}, V_{v1}, \ldots, V_{vK}],
\]

17
where $V_{v_k}$ is a part of the $X$ vector that corresponds to eigenvalue $v_k$. Then,

$$R = V^{-1}NV.$$  

So far, we have determined the matrix $R$. However, in order to specify $P$ completely we have to determine the value of the vector $P_0$. Assuming that we have the value of $R$, $P_0$ can be calculated by using the fact that the generating function for the sequence $\{P_0, P_1, \ldots\}$ is given by

$$G(z) = \sum_{n=0}^{\infty} z^n P_n = \sum_{n=0}^{\infty} z^n P_0 R^n,$$

or

$$G(z) = P_0 (I - zR)^{-1}. \quad (2.20)$$

Taking limits of both sides of equation (2.20) as $z \to 1$ yields

$$G(1) = P_0 (I - R)^{-1}. \quad (2.21)$$

Solving (2.21) for $P_0$, we obtain

$$P_0 = G(1)(I - R).$$

In order to determine $P_0$ we need to calculate $G(1)$. We use equation (2.8):

$$P_{n-1} \Lambda - P_n (\Lambda - Q + M) + P_{n+1} M = 0.$$  

Multiplying both sides of (2.8) by $z^{n-1}$ and summing over $n$ gives:

$$\sum_{n=1}^{\infty} P_{n-1} z^{n-1} \Lambda - \sum_{n=1}^{\infty} P_n z^{n-1} (\Lambda - Q + M) + \sum_{n=1}^{\infty} P_{n+1} z^{n-1} M = 0,$$

which simplifies to

$$G(z) \Lambda - \frac{1}{z}(G(z) - P_0)(\Lambda - Q + M) + \frac{1}{z^2}(G(z) - P_0 - P_1 z)M = 0,$$

or

$$G(z)[\Lambda z^2 - (\Lambda - Q + M)z + M] = (1 - z)P_0 M.$$
Taking limits as $z \to 1$, we find

$$G(1)Q = 0,$$

which implies that $G(1)$ is the left eigenvector of $Q$ corresponding to its zero eigenvalue. The other parameters of interest of the system can be determined from $R$ and $P_0$. For example the expected occupancy is given by[1]:

$$E[\bar{n}] = G(1)(I - R)^{-1}Re,$$

where $\bar{n}$ is the number in the system.

The eigenanalysis method of determining $R$ is desirable over the iterative method SS, due to its computational advantages. The method relies on the determination of the eigenvalues and eigenvectors of the matrix $AB^{-1}$, which was previously defined. In order to compute the matrix $AB^{-1}$, the matrix $M$ must be nonsingular. The matrix $M$ is a diagonal matrix and is nonsingular only if all its diagonal elements are positive. In terms of the system being modeled, this means that all service rates must be positive. Some of the systems, like the cellular telephone access system discussed in Chapter 1, have singular $M$. The eigenanalysis method as outlined in this chapter cannot be applied directly to analyze such systems. In Chapter 3 we will modify the eigenanalysis methodology described here to relax this restriction on $M$. 
Chapter 3

Eigenanalysis-based Determination of Rate Matrix for General Systems

1. Introduction

In the previous chapter methods to evaluate the rate matrix $R$ were discussed. Though Neuts[5] iterative technique can be used for general $M$, $A$, and $Q$ matrices, the number of iterations required to solve a system using his iterative technique is not polynomially bounded. On the other hand, the complexity of the eigenanalysis approach is $O(N^3)$, where $N$ is the dimension of the system. The eigenanalysis approach would therefore be preferred for systems having large dimensionality, especially if the traffic intensity is high[1].

In Chapter 2, we presented the eigenanalysis method for systems with nonsingular $M$ matrix. In this chapter we relax the requirement of nonsingular $M$. Such systems can be solved using alternate techniques, but the modifications presented in this chapter decrease the dimensionality of the problem and hence the resulting method is computationally efficient. The theory presented here constitutes original work to the best of the author’s knowledge.

2. Generalization of Eigenanalysis Approach

In this section we modify the eigenanalysis approach so that it can be used in the determination of the rate matrix for general $Q$ and singular $M$ matrix. As pointed out in Chapter 2, determining the rate matrix $R$ and $G(1)$, completely solves a quasi birth-death process.

As a point of departure, consider the matrix equation (2.15) with all the matrices defined as in Chapter 2:

$$V_0[\sigma^2A - \sigma(A - Q + M) + M] = 0. \quad (3.1)$$

In the model under consideration the matrix $M$ is diagonal. The $j^{th}$ element on the diagonal represents the service rate for the phase $j$. Thus, if the system has a zero service rate in say, phase $j$, then $m_{jj}$, the corresponding diagonal entry in the $M$ matrix, is zero. This makes the determinant
of \( M \) zero and hence, \( M \) is singular. The method outlined in Chapter 2 cannot be used directly if \( M \) is singular. We now present a technique that could be used for singular \( M \).

The matrix \( M \) with some of its diagonal elements zero can be expressed as

\[
T^T \hat{M} T,
\]

where

\[
\hat{M} = \begin{pmatrix} M_a & 0 \\ 0 & 0 \end{pmatrix},
\]

\( M_a \) is a nonsingular diagonal matrix, and \( T \) is a column translation matrix. That is, \( T \) is obtained from an identity matrix by rearranging its columns. Replacing \( M \) by \( T^T \hat{M} T \) in (3.1), we obtain

\[
V_\sigma [\Lambda \sigma^2 - (\Lambda - Q + T^T \hat{M} T) \sigma + T^T \hat{M} T] = 0.
\]

Because \( T^T T = I \), this expression can be rewritten as

\[
V_\sigma T^T [T \Lambda T^T \sigma^2 - (T(\Lambda - Q) T^T + \hat{M}) \sigma + \hat{M} T] = 0. \tag{3.2}
\]

We define \( \hat{\Lambda}, \tilde{V}_\sigma, \) and \( \hat{Q} \) as follows:

\[
\hat{\Lambda} = T \Lambda T^T = \begin{pmatrix} A_a & 0 \\ 0 & A_b \end{pmatrix},
\]

\[
\hat{Q} = T \hat{Q} T^T = \begin{pmatrix} Q_{aa} & Q_{ab} \\ Q_{ba} & Q_{bb} \end{pmatrix},
\]

and

\[
\tilde{V}_\sigma = V_\sigma T^T.
\]

Equation (3.2) can now be written as

\[
\tilde{V}_\sigma [\hat{\Lambda} \sigma^2 - (\hat{\Lambda} - \hat{Q} + \hat{M}) \sigma + \hat{M}] = 0. \tag{3.3}
\]

Taking the transpose of both sides of equation (3.3), we obtain

\[
[\hat{\Lambda} \sigma^2 - (\hat{\Lambda} - \hat{Q}^T + \hat{M}) \sigma + \hat{M}] \tilde{V}_\sigma^T = 0, \tag{3.4}
\]
where $\tilde{Q}^T$ is given by

$$
\begin{pmatrix}
Q_{aa} & Q_{ab} \\
Q_{ba} & Q_{bb}
\end{pmatrix}^T =
\begin{pmatrix}
Q_{aa}^T & Q_{ba}^T \\
Q_{ab}^T & Q_{bb}^T
\end{pmatrix}.
$$

Since $\tilde{V}_c$ is a row vector, it can be written as $\tilde{V}_c = [X_1 \quad X_2]$, where $X_1$ and $X_2$ are row vectors.

The dimension of $X_1$ is equal to the dimension of the matrix $M_a$. In expanded form (3.4) can then be written as

$$
\begin{bmatrix}
\begin{pmatrix}
\Lambda_a & 0 \\
0 & \Lambda_b
\end{pmatrix} \sigma^2 - \begin{pmatrix}
\Lambda_a & 0 \\
0 & \Lambda_b
\end{pmatrix} - \begin{pmatrix}
Q_{aa}^T & Q_{ba}^T \\
Q_{ab}^T & Q_{bb}^T
\end{pmatrix} + \begin{pmatrix}
M_a & 0 \\
0 & 0
\end{pmatrix} \sigma + \begin{pmatrix}
M_a & 0 \\
0 & 0
\end{pmatrix}
\end{bmatrix} \begin{bmatrix}
X_1 \\
X_2
\end{bmatrix} = 0. \quad (3.5)
$$

The above expression is a $(K + 1)$-square matrix of quadratic equations, called a second order $\lambda$-matrix. Upon partitioning (3.5), we find:

$$
[A_a \sigma^2 - (\Lambda_a - Q_{aa}^T + M_a) \sigma + M_a]X_1 + Q_{ab}^T \sigma X_2 = 0 \quad (3.6)
$$

and

$$
Q_{ab}^T \sigma X_1 + [A_b \sigma^2 - (\Lambda_b - Q_{bb}^T) \sigma] X_2 = 0. \quad (3.7)
$$

We are now going to convert the second order system of equations (3.6) and (3.7) into a first order system. We begin this process by defining the vector $Y_1$ as $Y_1 = \sigma X_1$. If $Y_1$ is substituted for $\sigma X_1$ in equations (3.6) and (3.7), we obtain the following system of equations:

$$
\sigma \Lambda_a Y_1 - (\Lambda_a - Q_{aa}^T + M_a) \sigma X_1 + M_a X_1 + Q_{ba}^T \sigma X_2 = 0, \quad (3.8)
$$

and

$$
\sigma Q_{ab}^T X_1 + [\Lambda_b \sigma^2 - (\Lambda_b - Q_{bb}^T) \sigma] X_2 = 0. \quad (3.9)
$$

For $\sigma \neq 0$, we may divide (3.9) by $\sigma$ to obtain

$$
Q_{ab}^T X_1 - (\Lambda_b - Q_{bb}^T) X_2 + \Lambda_b \sigma X_2 = 0. \quad (3.10)
$$

Equation (3.8) and (3.10) can then be written as

$$
\begin{bmatrix}
\begin{pmatrix}
M_a & 0 \\
Q_{ab}^T & -(M_a - Q_{aa}^T) \\
0 & 0
\end{pmatrix} & \begin{pmatrix}
0 \\
0 & I \\
0 & 0
\end{pmatrix}
\end{bmatrix} \begin{pmatrix}
X_1 \\
X_2
\end{pmatrix} = 0. \quad (3.11)
$$
Equation (3.11) is analogous to

\[
[B - \sigma A][X] = 0,
\]

(3.12)

where \(A\), \(B\) and \(X\) are defined as follows:

\[
A = \begin{pmatrix}
(\Lambda_a - Q_{aa}^T + M_a) & -Q_{ab}^T & -\Lambda_a \\
0 & -\Lambda_b & 0 \\
I & 0 & 0
\end{pmatrix},
\]

\[
B = \begin{pmatrix}
M_a & 0 & 0 \\
Q_{ab}^T & -(\Lambda_b - Q_{bb}^T) & 0 \\
0 & 0 & I
\end{pmatrix},
\]

and

\[
X = \begin{pmatrix}
X_1 \\
X_2 \\
Y_1
\end{pmatrix}.
\]

If \(M\) has \(N\) of its diagonal elements zero, then \(A\) and \(B\) are square matrices of dimension \((K+N+1)\) and \(X\) is a vector of length \((K+N+1)\). It should be noted that if the phase process is irreducible than the matrix \(Q_{bb}^T\) is invertible. The diagonal elements of the matrix \(Q_{bb}^T\) are negative, therefore the operation \(-(\Lambda_b - Q_{bb}^T)\) only increases its absolute value. It can be shown trivially that if the matrix \(Q_{bb}\) is invertible so is \(-(\Lambda_b - Q_{bb}^T)\) because it is a submatrix of an irreducible Markov chain.

In models of systems of practical interest the matrix \(M_a\) has only positive elements on its diagonal. Thus for QBD processes with irreducible phase processes the matrix \(B\) is invertible. We can therefore define \(X = B^{-1}\bar{X}\). Then, equation (3.12) can then be rewritten as

\[
[I - \sigma AB^{-1}][\bar{X}] = 0.
\]

(3.13)

In equation (3.13), \(1/\sigma\) is an eigenvalue of the matrix \(AB^{-1}\) and \(\bar{X}\) is its corresponding right eigenvector. The vector \(X\) is obtained by using \(X = B^{-1}\bar{X}\). Let \(\nu_k = 1/\sigma_k\) be one of the eigenvalues of \(AB^{-1}\) and \(V_{\nu_k}\) be the corresponding \(X\) vector. We can index the eigenvalues of \(AB^{-1}\) such that
\(|v_0| \leq |v_1| \leq |v_2| \ldots\), and define a \((K + 1)\) diagonal matrix \(N\) such that

\[
N = \begin{bmatrix}
v_0 & 0 & 0 & 0 \\
0 & v_1 & 0 & 0 \\
0 & 0 & v_2 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & v_K
\end{bmatrix}.
\]

We also define the matrix

\[
V = (V_{v_0} \ V_{v_1} \ \ldots \ \ V_{v_K}).
\]

The rate matrix \(R\) then is equal to

\[
R = V^{-1}NV.
\]

This concludes the description of our computational method for the special case in which the service rate matrix is singular.
Chapter 4

Example:
Cellular Telephone Access System

In this chapter, we present an example to illustrate the application of the eigenanalysis method. We will solve the problem presented in Guérin[15], which is described in detail in Chapter 1, Section 2.2.

Our example has several nice features. First, it is interesting from a practical point of view. Second, it is an extreme example in that $M$ has rank 1. Third, the generator for the Markov chain for the system is of the $G/M/1$ form with complex boundary conditions, so that alternate solution via Neuts’s iterative technique is possible. Fourth, the infinitesimal generator for the Markov chain for the states above level zero has the same form as the $M/Pi/1$ queuing system so that the rate matrix can be calculated by inverting a matrix, thereby giving an additional check on the validity of the rate matrix computation. Finally, the queuing process can be modeled by first modeling a modified system as an $M/G/1$ queuing system with exceptional first service and then scaling the results to obtain ergodic results for the real system.

This chapter comprises of three sections. In Section 1, we sketch the eigenanalysis approach to solve the problem. In Section 2, we present alternate methods to solve the problem in [15]. In Section 3, we comment on the results.

1. Analysis of the Cellular Telephone Access System

Guérin[15] describes a cellular telephone access system, the dynamics of which are characterized by the state diagram of figure 4.1. Inspection of the state diagram of figure 4.1 shows that the generator for the continuous time Markov chain shown in the state diagram has the form

$$Q = \begin{pmatrix}
B_{00} & B_{01} & \ldots & \ldots & \ldots & \ldots \\
B_{10} & B_{11} & A_0 & 0 & \ldots & \ldots \\
0 & B_{21} & A_1 & A_0 & 0 & \ldots \\
0 & 0 & A_2 & A_1 & A_0 & 0 & \ldots \\
0 & 0 & 0 & A_2 & A_1 & A_0 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}$$
Figure 4.1 State diagram for the Cellular telephone Access System.
The matrices $B_{00}$ and $B_{01}$ are of dimensions $(n - g + 1) \times (n - g + 1)$, and the matrix $B_{10}$ has dimension $(g + 1) \times (n - g + 1)$, and $B_{11}$ and $B_{21}$ are $(g + 1)$-square matrices. The generator has the form of equation (1.5.1) of Neuts[5], the form for a Markov chain of the G/M/1 type with complex boundary behavior.

From an analysis point of view, the system can be partitioned into a set of states at level zero, henceforth referred to as $Z$, and the set of states having positive level, where the level is given by the first index of the state descriptor.

As mentioned in Chapter 2, in order to completely solve the system we must find the matrix $R$ and the probability masses of the states at the zero level. The zero level probability vector is denoted by $P_0$, and is a $(n + 1)$-row vector. The solution technique can thus be broadly divided into two parts: determination of the rate matrix $R$ and determination of the vector $P_0$. The process of determining the rate matrix begins by writing the matrices $M, Q,$ and $\Lambda$ as defined in Chapter 2. In the model under consideration, if the zero level states and all associated flows with these states are ignored, then the remaining diagram resembles figure 2.1 with $(g + 1)$ phases. The matrices $M, Q$ and $\Lambda$ can be written as described in Chapter 2. After the above three matrices are defined, the eigenanalysis method described in Chapter 3 can be used to calculate the rate matrix $R$.

The determination of the probability masses of the states on the zero level is described in Section 1.5 of Neuts[5]. Corresponding to the matrix (1.5.2) in Neuts[5] we define the stochastic matrix

\[
B(R) = \begin{bmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11} + RM
\end{bmatrix}.
\]

The vector $P_0$ can be written as $[P_{0A} \ P_{0B}]$, where $P_{0A}$ and $P_{0B}$ correspond to states $(0, i)$ for $0 \geq i < n - g$ and $(0, i)$ for $n - g \geq i \geq n$, respectively. Thus, $P_{0A}$ is an $(n - g)$-row vector and $P_{0B}$ is an $(g + 1)$-row vector. The vector $P_0$ is calculated by scaling the left eigenvector of $B(R)$ corresponding to its zero eigenvalue; that is, define

\[
x_0 B(R) = 0,
\]
then because $B(R)$ is stochastic, all elements of $x_0$ have like sign so that we may choose

$$x_0 e = 1.$$ 

Let $x_0 = [x_{0A} \ x_{0B}]$, where the dimensions of $x_{0A}$ and $x_{0B}$ are the same as $P_{0A}$ and $P_{0B}$, respectively. Then, $P_0$ is proportional to $x_0$ and the constant of proportionality may be computed by recognizing that

$$P_{0A} + P_{0B} + \sum_{k=1}^{\infty} P_k = 1. \quad (4.1)$$

Neuts[5] has shown in Section 1.7, using Theorem 1.7.1, that for $k \geq 1$,

$$P_i = P_{0B} R^i, \quad (4.2)$$

where $P_i$ is a vector of probabilities corresponding to level $i$ for $i \geq 1$. Substituting equation (4.2) into (4.1) leads to

$$P_{0A} + \sum_{i=0}^{\infty} P_{0B} R^i = 1,$$

so that

$$k[x_{0A} + x_{0B}(I - R)^{-1} e] = 1,$$

where $k$ is the required constant of proportionality.

It is worth noting that equation (4.2) corresponds to equation (2.9) of Chapter 2. If the vector $P_{0B}$ and the rate matrix $R$ are known, the probability mass of all the states in figure 4.1 can be determined.

The expected number in the queue may be determined by summing tail probabilities. Let $\bar{n}$ represent the number in the queue. The tail probabilities are given by

$$P(\bar{n} > n) = \sum_{i=n+1}^{\infty} P_{0B} R^i e,$$

which gives

$$P(\bar{n} > n) = P_{0B} R^{n+1} (I - R)^{-1} e.$$
Figure 4.2 Expected queue length for $n=44$.

Then,

$$E[\bar{n}] = \sum_{n=0}^{\infty} P(\bar{n} > n) = \sum_{n=0}^{\infty} P_{0n} R^{n+1} (1 - R)^{-1} e,$$

which simplifies to

$$E[\bar{n}] = P_{0B} R (I - R)^{-2} e.$$

Calculations to reproduce figures 10 and 11 of [15] were made and the results are shown in figures 4.2 and 4.3.

2.1 M/G/1 Approach to Analyze the Cellular Telephone Access System

The process that counts the successive entries into level zero from the positive occupancy levels is a delayed alternating renewal process. That is, the process alternates between periods during which the level is zero and periods during which the level is positive. During periods of positive occupancy, the system behavior is identical to that of M/G/1 system with exceptional first service. Our approach is to first analyze the system as an M/G/1 system with exceptional service [25] and then to scale the resulting probabilities to obtain the ergodic probabilities for our system. The
Figure 4.3 Expected queue length for n=100.

periods during which the level is zero will be referred to as idle period, and the periods with positive occupancy will be referred to as the busy period.

The length of the idle period can be modeled as the time to absorption of the continuous time Markov chain shown in figure 4.4. Each entry to level zero from states with positive occupancy occurs at state $(0, n-g)$. The system transitions among the level zero states until it enters one of the states $(1, n-g)$ to $(1, n)$. Denote the vector of absorption probabilities by $\alpha = [\alpha_0 \alpha_1 \ldots \alpha_g]$, where $\alpha_i$ is the probability of leaving the zero level by entering state $(1, i+n-g)$.

Let $P(t)$ be the vector of probability masses for the states shown in figure 4.4 at a time $t$. Then $P(t) = [P_T(t) \ P_A(t)]$, where $P_T(t)$ represents the masses for the transient (zero level) states, and $P_A(t)$ represents the masses for the absorbing states $(1, n-g)$ to $(1, i)$ for $n-g < i \leq n$. We then have the differential equation

$$\frac{d}{dt} \begin{bmatrix} P_T(t) \\ P_A(t) \end{bmatrix} = \begin{bmatrix} T & T^0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} P_T(t) \\ P_A(t) \end{bmatrix},$$

(4.3)
Figure 4.4 The dynamics of the idle period.
where \[
\begin{bmatrix}
T & T^0 \\
0 & 0 
\end{bmatrix}
\] is the generator for the chain. The matrix \(T\) has dimension \((n+1) \times (n+1)\) and \(T^0\) is a \((n+1) \times (g+1)\) matrix, and \(Te + T^0 = 0\). Since \(P(t)\) is a vector of probability masses we also have

\[P_T(t)e + P_A(t)e = 1, \quad \text{for } t \geq 0. \quad (4.4)\]

The system of equations (4.3) and (4.4) can be solved to yield

\[P_T(t) = P_{T_0} e^{Tt},\]

where \(P_{T_0}\) is the vector of probability masses. Since all entries to zero level states occur in phase \((n - g)\), \(P_{T_0}\) is a vector of zeros except for a one in the \((n - g + 1)\)th position. \(P_A(t)\) is given by

\[P_A(t) = P_{T_0}(e^{Tt} - I)T^{-1}T^0.\]

Since the zero level states are transient, \(\lim_{t \to \infty} e^{Tt} = 0\). Therefore

\[\lim_{t \to \infty} P_A(t) = P_A(\infty) = -P_{T_0}(T^{-1}T^0).\]

From the definition of \(\alpha\), we find \(\alpha = P_A(\infty)\).

The level decreases will be referred to as service completion. We now characterize the exceptional and ordinary service times and give formulas for their moments. The service times are of phase type. That is, the service times are distributed as the time to absorption in the state \((i - 1, n - g)\) of the Markov chain whose state transition diagram is shown in figure 4.5 when the initial state is one of the states \((i, n - g)\) to \((i, j)\) for \(n - g < j \leq n\). In case of the exceptional first service, the initial state is one of the states \((1, n - g)\) to \((1, j)\) for \(n - g < j \leq n\) with the specific phase selected according to the probability vector \(\alpha\), which is defined above. All other services start in phase \((n - g)\). This is because, except for the final service of the busy period, all services end in the phase \((n - g)\) and a new service begins immediately. The generator of the chain described in figure 4.5 can be written as

\[
\begin{pmatrix}
N & N^0 \\
0 & 0 
\end{pmatrix},
\]

32
Figure 4.5 The state diagram for the service times.

where $N$ is a $(g + 1) \times (g + 1)$ matrix, and $N^0$ is a $(n + 1)$ column vector, such that

$$N^0 = \begin{bmatrix} n(\mu - g) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$ 

The noncentral moments for $i \geq 0$, are given by (2.2.7) of Neuts[5]

$$\bar{x}^i = (-1)^i i! (\beta N^{-1} e),$$

where $\beta$ is the initial probability vector. That is, $\beta$ defines the initial state probabilities for the chain.

The difference between the distribution of the exceptional first service and the ordinary service times
results from their initial state probabilities. For exceptional first service $\beta = \alpha$, for other services $\beta$ is a vector of length $(g + 1)$ with a 1 in the first position and zeros elsewhere.

Figure 4.6 The M/G/1 queuing system.

To obtain the probability masses for the system of figure 4.1, we first obtain the probabilities of the M/G/1 queuing system with exceptional first service as shown in figure 4.3. The level zero states are replaced by a state 0. The rate at which the process goes from state 0 to the level 1 states is given by the vector $\lambda \alpha$, and the time the process remains in state 0 before making a transition is exponentially distributed with parameter $\lambda$. The remainder of the state diagram is the same as in
We can now use the Pollaczek-Khintchine (PK) equation for the M/G/1 queuing system with exceptional first service to determine the probability generating function for the number in the queue.

The queue length probabilities of the actual system are different from that of the modified system by a scaling factor $\gamma$. We now present a simple argument to explain why this is so and also derive the factor $\gamma$. To aid in describing the theory that leads to the scaling factor, we will refer to all variables for the modified state diagram of figure 4.6 with a subscript $M$ and call it the process $M$. We will refer to the quantities in the actual state diagram of figure 4.1 by a subscript $A$ and call it the process $A$. The process $M$ and $A$ are alternating renewal processes. Let the length of the busy period be given by $\tilde{y}_i$ for $i = A, M$ and the length of the idle period be given by $\tilde{x}_i$ for $i = A, M$.

Consider a state $(i, j)$ for $i > 0$ and $j \geq (n - 2)$. The probability of being in state $(i, j)$ in system $A$ is given by $P_{ij_A} = \frac{E[\tilde{s}_{ij_A}]}{E[\tilde{x}_A] + E[\tilde{y}_A]}$, where the random variable $\tilde{s}_{ij_A}$ represents the amount of time spent in state $(i, j)$ in a typical cycle. Similarly, the probability of being in state $(i, j)$ for system $M$ is defined as $P_{ij_M} = \frac{E[\tilde{s}_{ij_M}]}{E[\tilde{x}_M] + E[\tilde{y}_M]}$. Taking ratios of the probabilities we obtain

$$\frac{P_{ij_A}}{P_{ij_M}} = \frac{E[\tilde{s}_{ij_A}]}{E[\tilde{x}_A] + E[\tilde{y}_A]} \cdot \frac{E[\tilde{x}_M] + E[\tilde{y}_M]}{E[\tilde{s}_{ij_M}]}.$$ 

Since the expected amount of time spent in any state of the system $A$ is the same as in $M$ for all states above level zero it follows that

$$E[\tilde{s}_{ij_A}] = E[\tilde{s}_{ij_M}].$$

Therefore, $P_{ij_A}$ and $P_{ij_M}$ are related as follows:

$$P_{ij_A} = P_{ij_M} \frac{E[\tilde{x}_M] + E[\tilde{y}_M]}{E[\tilde{x}_A] + E[\tilde{y}_A]} = \gamma P_{ij_M}, \quad (4.4)$$

where $\gamma$ is a constant and is equal to $\frac{E[\tilde{x}_M] + E[\tilde{y}_M]}{E[\tilde{x}_A] + E[\tilde{y}_A]}$. Since the probabilities scale, so do all the moments.

Therefore,

$$E[\tilde{q}_A] = \gamma E[\tilde{q}_M], \quad (4.5)$$

35
where $\tilde{q}_A$ and $\tilde{q}_M$ are the queue lengths in system A and system M respectively. Equation (4.5) suggests that in order to find the expected queue length of the actual system (system A) we find the expected queue of the M/G/1 system (system M) and scale it by a factor $\gamma$.

We now concern ourselves with calculating the factor $\gamma$, which is defined above as

$$
\gamma = \frac{E[\tilde{z}_A] + E[\tilde{y}_M]}{E[\tilde{z}_A] + E[\tilde{y}_A]}.
$$

The random variables $\tilde{y}_A$ and $\tilde{y}_M$ represent the length of the busy period [26] in the actual system and the modified systems respectively. They are always equal, because by modifying the system we have not changed the dynamics of the busy period. The expected length of the busy period is given by

$$
E[\tilde{y}_A] = E[\tilde{y}_M] = \frac{E[\tilde{z}_A]}{1 - \lambda E[\tilde{z}]}.
$$

where $\tilde{z}_e$ is the length of the exceptional service and $\tilde{z}$ is the length of the ordinary service, which are calculated above. The expected length of the idle period of the M/G/1 system (system M) described in figure 4.6 is $1/\lambda$. The expected length of the idle period in the actual system is calculated using the following expression for the noncentral moments Neuts[5]:

$$
E[\tilde{z}_A] = (-1)(P_0 T^{-1} e).
$$

2.2 Alternate Method to Calculate the Probability Vector $P_0$.

The level zero probabilities can be calculated by expressing all the probability masses of the states on level zero in terms of the state $P_{0,n-g}$. That is, the vector $P_0 = f(P_{0,n-g})$. This can be done because the system does not transition into level zero at any state other than $(0, n - g)$. Summing the joint probability masses for the states in level zero enables us to get the probability of being in level zero in terms of the probability $P_{0,n-g}$.

The probability of being in level zero can be calculated by using renewal theory and is readily found to be [26]

$$
P_{0e} = f(P_{0,n-g}) = \frac{E[\tilde{z}_A]}{E[\tilde{z}_A] + E[\tilde{y}_A]}.
$$
This technique provides an additional check to verify the probability masses for the level zero states.

2.3 Alternate Method to Calculate the Rate Matrix

The matrix generator for the Markov chain for the states above the level zero has the same form as the M/PH/1 queueing system. The rate matrix of such a process can be obtained in closed form as described in Section 3.2 of [5]. From (3.2.3) of [5], we find

\[ R = \lambda[\lambda I - \lambda\epsilon b - N]^{-1}, \]

where \( N \) has been previously defined and \( b = [1 \ 0 \ 0 \ldots] \) is the vector of initial phase probabilities for the ordinary service process. This gives us another method to verify that the rate matrix computed using eigenanalysis is indeed correct.

The rate matrix of this system can also be calculated using the Neuts’s iterative method, which provides an additional check on the correctness.

3. Results

In a typical cellular telephone network, a number of different cell service units are located in a given geographical area. Each of these service units are assigned an appropriate number of frequency channels. A few of these channels are reserved as guard channels. Given a level of service required, a designer has to determine the number of frequency channels and guard channels that are required. The level of service can be characterized by two parameters: the probability of an ongoing call being terminated and the time it takes before a channel become available to place a call. The first parameter \( (P_B) \) can be calculated using

\[ P_B = \sum_{i=0}^{\infty} P_{i,0}. \]

The expected waiting time before a caller gets a channel to place a call is related to the expected occupancy through Little’s result. These occupancies are given in figure 4.2 and 4.3. All the methods listed above were used to verify the results presented in figures 4.2 and 4.3. Our results correlated to each other to machine precision. However, the results presented in this chapter differ from
those presented in Guérin[15]. We verified some of the formulae in [15] using the software package Mathematica. We found that the value of \( P_0 \) obtained by using equation (9) in [15] checks with the value of \( P_0 \) calculated using techniques described above. The matrix \( T \) in [15] is the transpose of the matrix \( R \) defined in this thesis. We used the system of equations on page 161 of [15] to calculate \( T \), but the results obtained differ from the known correct calculation. Our conclusion is that the formulae for the entries of \( T \) in [15] appear to be incorrect, but at this time we have not identified a specific error.
Chapter 5

Conclusions

In this thesis we studied solution techniques to solve continuous time Markov chain having
infinitesimal generator of the type

\[
Q = \begin{pmatrix}
B_0 & A_0 & 0 & \cdots & \cdots & \cdots \\
B_1 & A_1 & A_0 & 0 & \cdots & \cdots \\
0 & A_2 & A_1 & A_0 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \cdots \\
0 & 0 & A_2 & A_1 & A_0 & \ddots \\
\end{pmatrix},
\]

where the matrices \( B_i \) and \( A_i \) have dimensions \((K + 1) \times (K + 1)\). In order to solve for the
probability masses of the chain described by this generator, we need to calculate two quantities, the rate
matrix, \( R \), and the level zero probabilities, \( P_0 \). The rate matrix is a fundamental descriptor for the
Markov chain of type described above. A knowledge of \( R \) allows one to compute not only occupancy
distributions but other quantities of interest such as waiting times.

In this thesis we reviewed some techniques to calculate the rate matrix for a quasi birth and
death process and discussed the successive substitution and the eigeanalysis techniques in detail.
We pointed out that eigenanalysis method, as described in Daigle and Lucantoni[1], is an efficient
method of calculating the rate matrix, but its application is limited to a somewhat restricted class
of systems. A modification to the eigenanalysis method of [1] was developed. This development
constitutes an original contribution of the thesis. The modifications allow the use of eigenanalysis
for more general systems than in [1]. In particular, it can be applied to systems with singular \( M \)
matrices.

The QBD model discussed in this thesis has wide applications in communication technology
and manufacturing, some of which were reviewed in Chapter 1. As an illustration of the modified
eigenanalysis, we solved for the equilibrium state probabilities for the cellular telephone access
system of Guérin[15]. We calculated the expected queue length using the modified eigenanalysis
method. The results were verified by calculating the expected queue lengths by three independent
methods. Though the results using various techniques agreed with each other, they did not agree
with those presented in Guérin[5]. As discussed in Chapter 4, we concluded that there appears to be an error in the determination of the matrix $T$ in [15].

The results obtained by the analysis of the cellular phone access system can be used in designing the cell service units. In particular they can be used to determine both the total number of channels required and the number that should be reserved as guard channels to obtain a satisfactory level of service. The results obtained in chapter 4 also give a probabilistic understanding of the cellular telephone access system.

In conclusion we reiterate that the objective of the thesis was to modify the eigenanalysis approach to solve more general systems and to demonstrate the validity of our results. We not only achieved that objective but also developed an effective methodology to examine the cellular telephone access system problem. In doing so, we demonstrated that the results relating to the cellular telephone problem, previously presented in the literature, are incorrect.
Bibliography


Appendix

This appendix contains listings of two programs, A and B. The first listing is the implementation of the successive substitution algorithm to determine the matrix $R$. The second listing is for the program that determines $R$ using the modified eigenanalysis procedure described in Chapter 3 and finds the vector $P_0$ for the cellular telephone access problem described in Chapter 4. These programs were implemented using Symantec Corporation's "Think's LightspeedC" on a Macintosh SE/30 with 5 Mbytes of RAM. The "C Scientific Library" by Eigenware Technologies of Saratoga, CA were used to do the eigenanalysis. We now describe the input for these programs.

Both programs are designed to work with the same input text file, which must have the following information in the order given below:

i) $K$, which is equal to one less than the dimension of the matrix $R$.

ii) The diagonal terms of the matrix $A$.

iii) The diagonal terms of the matrix $M$.

iv) All the terms of the $P$ matrix, row-by-row.

v) The total number of channels, $n$, as in figure 4.1.

vi) The matrix $B[R]$ (see Chapter 4), row-by-row.

Listings of some minor routines are omitted. Their functions are evident from the context of their use, and they can be supplied by the reader with minimal effort.
A. Matrix Geometric Routines

```c
#define MC68881
#include <math.h>
#include <stdio.h>
#include <unix.h>
#include <storage.h>
#include <strings.h>
#include "util.h"
#include "matrix_util.h"
define file_in "data"
define rate_matrix_out "phase4_luc_nik1_output"
define step_size (int)100
define TINY (double)1.0e-20
define SMALL (double)1.0e-10
define max_improve (int)50
define TRUE (int)1
define FALSE (int)0

void get_model_parameters();
int K;
double tau;
double *lambda, *mu, *beta, *delta, **P_matrix, time_scale;
main()
{
    void nerror();
    int index, index1,*permut, flag, improve_steps, sgn, nonsingular;
    double *a_1, *a_0, *a_2, *identity, **B_0, **B_1, **R_M, **P, *X, *pi_0, d,
        **BIGA, *wr, *wi, *zr, *zi, **ScriptA, **nullvects, **tri_SA, **S,
        new_new_z(), *multvect, *pi_z_0, **pfe_coef, mass, distribution_check();
    double *Y, *Z, z_;
    FILE *fopen(), *fp1;
    time_scale=1.0;
    P_matrix = dmatrix(0,K,0,K);
    get_model_parameters();

    /* input parameters are K, birth and death rates */
    a_1 = dmatrix(0,K,0,K);
    assign_a_1(a_1); /* lambda-P+M */
    identity = d_identity_matrix(0,K);
    a_0 = dmatrix(0,K,0,K);
    solve_linear_matrix(K,a_1,identity,a_0);
    B_0 = row_transform_matrix(K,lambda,a_0);
    B_1 = row_transform_matrix(K,mu,a_0);
    free_dmatrix(a_0,0,K,0,K);
    free_dmatrix(a_1,0,K,0,K);
```
/* determine the rate matrix for the matrix geometric method */
R_M = dmatrix(0,K,0,K);
solve_R(R_M,B_0,B_1,step_size);
fp1 = fopen(rate_matrix_out,"w");
fprintf(fp1,"The matrix-geometric rate matrix follows:\n");
for (index=0;index<=K;index++) {
    for(index1=0;index1<=K;index1++) {
        fprintf(fp1,"\%d[\%d]: %f\n",index,index1,R_M[index][index1]);
        if((index1+1) % 5==0) fprintf(fp1,"\n");
    }
}
fclose(fp1);

void tridag(a,b,c,r,u,n)
double *a,*b,*c,*r,*u;
int n;
{
    int j;
    double bet,*gam;
    gam=dvector(1,n);
    if (b[1] == 0.0) nerror("error 1 in tridag");
u[1] = r[1]/(bet-b[1]);
    for(j=2;j<=n;j++) {
        gam[j] = c[j-1]/bet;
        bet=b[j-1]*gam[j];
        if(bet == 0.0) bet = TINY;
        u[j] = (r[j]-a[j]*u[j-1])/bet;
    }
    for(j=(n-1);j>=1;j--) u[j] = gam[j+1]*u[j+1];
    free_dvector(gam,1,n);
}

ludtri(a,b,c,n,m)
int n; double *a,*b,*c,*m;
/* routine performs a lower-upper decomposition of a tridiagonal matrix. 
The vectors of the entering tridiagonal matrix are as follows: 
lower diagonal a[1..n-1]; diagonal b[1..n]; upper diagonal c[1..n-1] 
The decomposition leaves c unchanged except for scaling, and the revised 
lower diagonal and diagonal vectors are returned in a and b, respectively. 
No pivoting is provided. If the matrix is singular, then it is assumed 
that b[n] will turn out to be zero. If it does not, then some sort of 
iterative procedure to improve the input must be undertaken. m[1..n] is 
a scaling vector which attempts to improve roundoff by normalizing the 
columns so that the largest element in a column is unity in absolute 
value. A procedure to improve the candidate null value is new_x */

{ 
    int i;
    m[1] = fabs(a[1]);
    ii (fabs(b[1]) > m[1]) m[1] = fabs(b[1]);
    a[1] /= m[1]; b[1] /= m[1];
    for (i=2;i<=n-1;i++) 
        { 
            m[i] = fabs(a[i]);
            if (fabs(b[i]) > m[i]) m[i] = fabs(b[i]);
            if (fabs(c[i-1]) > m[i]) m[i] = c[i-1];
            a[i] /= m[i]; b[i] /= m[i]; c[i-1] /= m[i];
        }
    m[n] = fabs(c[n-1]);
    if (fabs(b[n]) > m[n]) m[n] = fabs(b[n]);
    b[n] /= m[n]; c[n-1] /= m[n];
    for (i=1;i<=n-1;i++) 
        { 
            a[i] /= b[i];
            b[i+1] = a[i]*c[i];
        }
    /* if(fabs(b[n]) > SMALL) merror("ludtri failed to yield b[n] = 0"); */
    /* the above commented statement can be reinserted if it is desired to 
stop if the value of b[n] is not sufficiently close to zero. */
}

adjtri(a,b,c,m,n,x,s)
int n; double *a,*b,*c,*m,*x,*s;

/* routine begins with the output of ludtri and attempts to find nullvectors 
and scalefactors in order to compute the adjoint of a singular tridiagonal 
matrix resulting from evaluating the tridiagonal lambda-matrix at its 
null values. The candidate null vector is returned in x[1..n] while the
scaling row vector is returned in s[1..n]. That is the columns of \( \text{adj a(z)} \)
are proportional to the null vectors; s[1..n] provides the proportionality
constants. */

```c
{
    int i,j,sgn;
    double *bprod, *cprod;

    bprod = dvector(1,n);
    cprod = dvector(1,n);
    bprod[1] = 1.0;
    for (j=2;j<=n;j++) {
        bprod[j] = bprod[j-1] * b[j-1];
    }
    cprod[n] = 1.0;
    for (j=n-1;j>=1;j--) {
        cprod[j] = cprod[j+1] * c[j];
    }
    if (n % 2 == 0) {
        sgn = 1.0;
    } else {
        sgn = -1.0;
    }
    for (j=1;j<=n;j++) {
        sgn = -1.0 * sgn;
        x[j] = sgn * bprod[j] * cprod[j];
    }
    /* now scale the nullvectors back to original norms */
    for (j=1;j<=n;j++) {
        for (i=1;i<=n;i++) {
            if (i != j) x[j] *= m[i];
        }
    }
    s[n] = 1;
    for (j=n-1;j>=1;j--) {
        s[j] = - s[j+1] * a[j];
    }
}
```

`zero_probs_neuts(x,mat,y)`
`double *x,**mat,*y;`
\{ 
    int i, j;
    for (i=0; i<\text{\texttt{K}}; i++) {
        y[i] = 0.0;
        for (j=0; j<\text{\texttt{K}}; j++) {
            if (j == i) y[i] += (1.0 - mat[j][i]) \times x[j];
            if (j != i) y[i] -= mat[j][i] \times x[j];
        }
    }
\}

\texttt{assign\_BIGA(a)}
\texttt{double \star\star a;}

\{ 
    int i, j;
    a[1][1] = (\text{\texttt{lambda}}[0] + \text{\texttt{mu}}[0] + \text{\texttt{beta}}[0]) / \text{\texttt{mu}}[0];
    a[1][2] = -\text{\texttt{beta}}[0] / \text{\texttt{mu}}[1];
    a[\text{\texttt{K}}+1][\text{\texttt{K}}] = -\text{\texttt{delta}}[\text{\texttt{K}}] / \text{\texttt{mu}}[\text{\texttt{K}}-1];
    a[\text{\texttt{K}}+1][\text{\texttt{K}}+1] = (\text{\texttt{lambda}}[\text{\texttt{K}}] + \text{\texttt{mu}}[\text{\texttt{K}}] + \text{\texttt{delta}}[\text{\texttt{K}}]) / \text{\texttt{mu}}[\text{\texttt{K}}];
    for (i=2; i<\text{\texttt{K}}; i++) {
        a[i][i-1] = -\text{\texttt{delta}}[i-1] / \text{\texttt{mu}}[i-2];
        a[i][i] = (\text{\texttt{lambda}}[i-1] + \text{\texttt{mu}}[i-1] + \text{\texttt{beta}}[i-1] + \text{\texttt{delta}}[i-1]) / \text{\texttt{mu}}[i-1];
        a[i][i+1] = -\text{\texttt{beta}}[i-1] / \text{\texttt{mu}}[i];
    }
    for (i=1; i<\text{\texttt{K}}+1; i++) {
        a[\text{\texttt{K}}+1+i][i] = 1;
        a[i][\text{\texttt{K}}+1+i] = -\text{\texttt{lambda}}[i-1] / \text{\texttt{mu}}[i-1];
    }
\}

\texttt{assign\_inv\_BIGA(a)}
\texttt{double \star\star a;}

\{ 
    int i, j;
    a[1][1] = ((\text{\texttt{lambda}}[0] + \text{\texttt{mu}}[0] + \text{\texttt{beta}}[0]) / \text{\texttt{lambda}}[0]);
\}

49
a[1][2] = -(mu[0]/lambda[0])*(beta[0]/mu[1]);
a[K+1][K+1-1] = -(mu[K]/lambda[K])*(delta[K]/mu[K-1]);
a[K+1][K+1] = (lambda[K] + mu[K] + delta[K])/lambda[K];
for (i=2;i<=K;i++)
    
    a[i][i-1] = -(mu[i-1]/lambda[i-1])*(delta[i-1]/mu[i-2]);
    a[i][i] = ((lambda[i-1] + mu[i-1] + beta[i-1]) + delta[i-1])/lambda[i-1];
    a[i][i+1] = -(mu[i-1]/lambda[i-1])*(beta[i-1]/mu[i]);

for (i=1;i<=K+1;i++)
    
    a[K+1+i][i] = 1;
    a[i][K+1+i] = -mu[i-1]/lambda[i-1];


Eval_ScriptA(a,z)

double **a,z;
{
    double zsqr;
    int i,j;
    zsqr = z*z;
    a[1][1] = lambda[0]*zsqr - (lambda[0] + mu[0] + beta[0])*z + mu[0];
    a[1][2] = beta[0]*z;
    a[K+1][K] = delta[K]*z;
    a[K+1][K+1] = lambda[K]*zsqr - (lambda[K] + mu[K] + delta[K])*z + mu[K];
    for (i=2;i<=K;i++)
        
        a[i][i-1] = delta[i-1]*z;
        a[i][i] = lambda[i-1]*zsqr
               -(lambda[i-1] + mu[i-1] + beta[i-1] + delta[i-1])*z + mu[i-1];
        a[i][i+1] = beta[i-1]*z;

}

Eval_tri_ScriptA(a,z)

double **a,z;
{
    double zsqr;

  50
int i;
zqr = z*z;
for(i=0;i<=K-1;i++) {
    a[0][i] = beta[i]*z;
}
a[1][0] = lambda[0]*zqr - (lambda[0] + mu[0] + beta[0])*z + mu[0];
a[1][K] = lambda[K]*zqr - (lambda[K] + mu[K] + delta[K])*z + mu[K];
for(i=1;i<K;i++) {
    a[1][i] = lambda[i]*zqr
    - (lambda[i] + mu[i] + beta[i]+delta[i])*z
    + mu[i];
}
for(i=1;i<=K;i++) {
    a[2][i] = delta[i]*z;
}
}

void norm(x) double *x;
{
    int i;
    double sum;
    sum = 0.0;
    for (i=0;i<=K;i++) sum += fabs(x[i]);
    for (i=0;i<=K;i++) x[i] = x[i]/sum;
}

assign P(p)
doable **p;
{
    int col;
    for (col=2;col<=K+1;col++) p[col-1][col]= delta[col-1];
    p[1][1] = beta[0];
    if (K > 1) for (col=2;col<=K;col++) p[col][col]= delta[col-1]+beta[col-1];
    p[K+1][K+1] = delta[K];
    for (col=1;col<=K;col++) p[col+1][col]= -beta[col-1];
}

51
assign_X(x)
double *x;
{
    int col;
    for (col=0;col<=K;col++) x[col] = 1/(double)(K+1);
}

free_model_vectors()
{
    free_dvector(lambda,0,K);
    free_dvector(mu,0,K);
    free_dvector(beta,0,K);
    free_dvector(delta,0,K);
}

assign_a_1(a)
double **a;
{
    int row,i,j;
    for (i=0;i<K+1;i++) {
        for (j=0;j<K+1;j++) {
            a[i][j]=P_matrix[i][j];
        }
    }
    for (i=0;i<K+1;i++) a[i][i]=lambda[i]+mu[i]+a[i][i];
}
assign_a_0(a)
double *a;
{
    int row;
    for (row=0;row<=K;row++) {
        a[row] = lambda[row]/tau;
    }
}

assign_a_2(a)
double *a;
{
    int row;
    for (row=0;row<=K;row++) {
        a[row] = mu[row]/tau;
    }
}

solve_R(r,a,b,iter_step)
double **r,**a,**b;
int iter_step;
{
    int i,j,t,iter,tot_count;
    double **old_r, **a_2, **a_1, **r_2;
    double step_diff;
    old_r = dmatrix(0,K,0,K);
    for(i=0;i<=K;i++) for(j=0; j<=K;j++) r[i][j] = a[i][j];
    a_2 = dmatrix(0,K,0,K);
    a_1 = dmatrix(0,K,0,K);
    r_2 = dmatrix(0,K,0,K);
    step_diff=10.0;
    tot_count = 0;
    while (step_diff > 0.0000000001) {
        iter = iter_step;
        for(i=0;i<=K;i++) {
for(j=0;j<=K;j++) {
    old_r[i][j] = r[i][j];
}
}
do {
    matrix_mul(K,r,r_2);
    matrix_mul(K,r_2,b,a_1);
    matrix_add(K,a,a_1,a_2);
    for(i=0;i<=K;i++) for(j=0;j<=K;j++) r[i][j] = a_2[i][j];
    tot_count++;
} while(--iter);
step_dif = 0.0;
for(i=0;i<=K;i++) {
    for(j=0;j<=K;j++) {
        step_dif += fabs(r[i][j] - old_r[i][j]);
    }
}
free_dmatrix(old_r,0,K,0,K);
free_dmatrix(a_2,0,K,0,K);
free_dmatrix(a_1,0,K,0,K);
free_dmatrix(r_2,0,K,0,K);


dmatrix_dense_tri_product(n,x,y,z)
int n; double **x,**y,**z;
{
    int row, col;
    for(row=0;row<=n;row++) {
        z[row][0] = x[row][0]*y[1][0] + x[row][1]*y[2][1];
    }
    if (n > 1) {
        for(row=0;row<=n;row++) {
            for(col=1;col<n;col++) {
                z[row][col] = x[row][col-1]*y[0][col-1] + x[row][col]*y[1][col]
                  + x[row][col+1]*y[2][col+1];
            }
        }
    }
}
for(row=0;row<n;row++) {
    z[row][n] = x[row][n-1]*y[0][n-1] + x[row][n]*y[1][n];
}

void get_model_parameters() {
    int index, index2, ka;
    double temp, try;
    FILE *fopen0, *fp;
    fp = fopen(file_in, "r");
    if (fp != NULL) {
        fscanf(fp, "%d", &K);
        printf("n This is lambda \n");
        lambda = dvector(0, K);
        for (index = 0; index <= K; index++) {
            fscanf(fp, "%f", &temp);
            lambda[index] = temp/time_scale;
            printf("%5.3f \n", temp);
        }
        mu = dvector(0, K);
        fprintf(stdout, "n This is mu \n");
        for (index = 0; index <= K; index++) {
            fscanf(fp, "%f", &temp);
            printf("%5.3f \n", temp);
            mu[index] = temp/time_scale;
        }
        P_matrix = dmatrix(0, K, 0, K);
        fprintf(stdout, "n This is P \n");
        for (index = 0; index <= K; index++) {
            printf("\n");
            for (index2 = 0; index2 <= K; index2++) {
                fscanf(fp, "%f", &temp);
                fprintf(stdout, "%5.3f \n", temp);
                P_matrix[index][index2] = temp/time_scale;
            }
        }
    index = fscanf(fp, "%d", &tau);
    fprintf(stdout, "%5.3f \n", tau);
    fclose(fp);
} else {
printf("no input file\n");
}
}

#define RADIX 2.9
#define SWAP(g,h) (y=(g);(g)=(h);h=y;)
#define SIGN(a,b) ((b) > 0 ? fabs(a) : -fabs(a))
B. Modified Eigenanalysis Routines

```c
#define _MC68881_
#include <math.h>
#include <stdio.h>
#include <unix.h>
#include <storage.h>
#include <strings.h>
#include "util.h"
#include "matrix_util.h"
#include<complex_util.h>
#define step_size (int)100
#define TINY (double)1.0e-20
#define SMALL (double)1.0e-10
#define max_improve (int)50
#define TRUE (int)1
#define FALSE (int)0
#define RADIX 2.0
#define SWAP(g,h) {y=(g);(g)=(h);(h)=y;}

void norm(),sqr_norm(), uridag(), mprove(),
    norm_nullvecs0, eigen_norm();
extern int K, *find_t0(), total_phases;
extern double tau, *mu, *lambda, **P_matrix, *ergodic_phase, *P_zero, **almost_B_R_matrix, time_scale;
extern FILE *fpout;
extern void check_stability();
double **extract_final_nullvecs0, **form_sigma();
int show_detail;
FILE *fopen0, *fp1;
void show_matrix0();

main_program()
{
    unsigned long aaa,
    int index,index1,index2, flag, improve_steps, sgn, nonsingular, r, *order,
    double **I_plus_a_1, **R_M, *a_0, *a_2, **P, *X, d,
        **BIGA, *wr, *wi, *zr, *zi, **ScriptA, **ScriptA_lud, **tri_SA, **S,
        **adjScriptA, **adjScriptA_lud, detScriptA, **BIGB,
        *nullvec, *pi_z_0, **pfe_coef, mass_distribution_check0, zero_probs_x0,
        **change_matrix0, **change_to_one_matrix0, **form_P_hat_transform0;
    double *BIGA_MOD, *D, *PV, **PV_mat, p_0, **CID, **BIGB.Inv, **BIGAB,
        **BIGABC, **BIGX.Inv, **BIGR, *wr0, **BIGR1, **EIGEN, *work_vect2,
        **BIGX, **eigen_val_mat, **work2, *x_0, *x_1;
    double **r-nulls, **I_null_vecs, *I_null_vals, **I_null_vecs_inverse, **sigma,
```
**R1_M;
int number_l_nulls, number_r_nulls;
double **I_minus_R, **I_minus_R_Inv, **real_R, **work1, **work_vec, occupancy;
c_rect *eigen_value, **eigen_vector, **rel_nulvec_Inv, **rel_nulvec,
**CID_nul, **work_R, **nullvecs;
double *eigen_value_abs;
double *Y, *Z, z_i, **identity;
show_detail=1; /* Set this to one if need to see details
of the various matrices */
time_scale = 1.0;

/* Find the matrix T and assign the matrix A in equation (3.12) */
T = find_t(&no_pos_mu);
BIGA = dmatrix(0,(K+no_pos_mu),0,(K+no_pos_mu));
assign_BIGA_transpose(BIGA,T,no_pos_mu);
show_matrix_f(BIGA,0,K+no_pos_mu,"BIGA");

/* find the matrix B in equation (3.12) */
BIGB = dmatrix(0,(K+no_pos_mu),0,(K+no_pos_mu));
assign_BIGB_transpose(BIGB,T,no_pos_mu);
show_matrix_f(BIGB,0,K+no_pos_mu,"BIGB");

/* find B inverse */
CID = dmatrix(0,(K+no_pos_mu),0,(K+no_pos_mu));
BIGB_Inv = dmatrix(0,(K+no_pos_mu),0,(K+no_pos_mu));
assign_CID(CID,K+no_pos_mu);
solve_linear_matrix(K+no_pos_mu,BIGB,CID,BIGB_Inv);

/* find the matrix AB inverse */
BIGAB = dmatrix(0,(K+no_pos_mu),0,(K+no_pos_mu));
matrix_mul(K+no_pos_mu,BIGA,BIGB_Inv,BIGAB);
free_dmatrix(BIGA,0,(K+no_pos_mu),0,(K+no_pos_mu));
BIGA = BIGAB;
BIGA = change_matrix(BIGA,K+no_pos_mu);

/* change the rectangular matrix to a linear array for using Eigenware routines */
BIGA_MOD = dvector(1,(K+no_pos_mu+1)*(K+no_pos_mu+1));
matrix_lin_array(BIGA,BIGA_MOD+1,(K+no_pos_mu+1));

/* balance the target matrix */
D = dvector(1,(K+no_pos_mu+1));
LH = ivector(1,2);
balance(BIGA_MOD+1,D+1,LH+1,(K+no_pos_mu+1));

/* convert to Heensenberg form */
BAK = ivector(1,(K+no_pos_mu+1));
elmhes(BIGA_MOD+1,(K+no_pos_mu+1), BAK+1);
/* find the eigenvalues and eigenvectors */
wr = dvector(1,(K+no_pos_mu+1));
wi = dvector(1,(K+no_pos_mu+1));
CNT = ivector(1,(K+no_pos_mu+1));
PV = dvector(1,(K+no_pos_mu+1),K+no_pos_mu+1));
elmtrans(BIGA_MOD+1, PV+1, LH+1, BAK+1, (K+no_pos_mu+1));
hqr2(BIGA_MOD+1, PV+1,wr+1,wi+1,LH+1,CNT+1, (K+no_pos_mu+1));
eigen_value = c_rect_vector(0, K+no_pos_mu);
eigen_vector = c_rect_matrix(0, K+no_pos_mu, 0, K+no_pos_mu);
assim_eigen_value(wr+1, wi+1, eigen_value, K+no_pos_mu+1);
free_dvector(wr, 1, (K+no_pos_mu+1));
free_dvector(BIGA_MOD, 1, (K+no_pos_mu+1), K+no_pos_mu+1));
bakbak(PV+1, D+1, LH+1, (K+no_pos_mu+1), (K+no_pos_mu+1));
assign_eigen_vectors(eigen_vector, wi+1, PV+1, K+no_pos_mu+1);
{
  int i,j;
  if (show_detail==1){
    fprintf(stderr,"This is the PV vector \n");
    j=0;
    for(i=1;i<=(K+no_pos_mu+1)+(K+no_pos_mu+1);i++)
      if (j>=5){
        j=0;
        fprintf(stderr,"\n");
      }
    j=j+1;
    fprintf(stderr,"%.4f \t", PV[i]);
  }
}
free_dvector(wi, 1, (K+no_pos_mu+1));
nulvecs = c_rect_matrix(0, K+no_pos_mu, 0, K+no_pos_mu);
{
c_rect **BIGB_Inv_Comp;
  int ii,jj;
  BIGB_Inv_Comp = c_rect_matrix(0, K+no_pos_mu, 0, K+no_pos_mu);
  for (ii=0;ii<K+no_pos_mu+1;ii++)
    for (jj=0;jj<K+no_pos_mu+1;jj++)
      BIGB_Inv_Comp[ii][jj].re = BIGB_Inv[ii][jj];
      BIGB_Inv_Comp[ii][jj].im = 0.0;
}
show_complex_matrix_f(BIGB_Inv_Comp, 0, K+no_pos_mu, " BIGB_Inv");
show_complex_matrix_f(eigen_vector, 0, K+no_pos_mu, " This is Eigen_vector");
complex_matrix_mul(K+no_pos_mu, BIGB_Inv_Comp, eigen_vector, nullvecs);
show_complex_matrix_f(nullvecs, 0, K+no_pos_mu, " 8 times eigen vector");
}
/* order the eigenvector and eigenvalues in the increasing order of the absolute values */
/* create a vector of absolute eigenvalues */
eigen_value_abs = dvector(0, K+no_pos_mu);
{
    int ii;
    for (ii=0; ii<K+no_pos_mu+1; ii++)
        eigen_value_abs[ii] = mag_c_rect(eigen_value[ii]);
}

order = ivector(0,K+no_pos_mu);
fsheld_index(eigen_value_abs, order, (K+no_pos_mu+1));
complex reorder nullvecs(order, K+no_pos_mu+1);
rel_nulvect = c_rect_matrix(0,K,0,K);
complex extract Relevant Egen Matrix top right(nullvecs, rel_nulvect, no_pos_mu);
show_matrix_f(nullvecs, 0,K+no_pos_mu,"eigen vector after sorting");
show_complex_matrix_f(rel_nulvect, 0,K,"extracted from nullvect real nulvect");
rel_nulvect Inv= c_rect_matrix(0,K,0,K);
CID_nul= c_rect_matrix(0,K,0,K);
complex assign_CID(CID_nul, 0);
solve COMPLEX linear matrix(K, K, rel_nulvect, CID_nul, rel_nulvect Inv);
show_complex_matrix_f(rel_nulvect Inv, 0,K,"inverse of the extracted matrix");
work_R= c_rect_matrix(0,K,0,K);
{
    int ii, jj;
    for (jj=0; jj<K+1; jj++)
        for (ii=0; ii<K+1; ii++)
            work_R[ii][jj] = c_mul(rel_nulvect[ii][jj], eigen_value[no_pos_mu+jj]);
}

show_matrix_f(work_R, 0,K,"extracted times eigen value");
complex matrix_mul(K, work_R, rel_nulvect Inv, rel_nulvect);
{
    int ii, jj;
    for (ii=0; ii<K+1; ii++)
        for (jj=0; jj<K+1; jj++)
            work_R[T[ii][jj]].re = rel_nulvect[ii][jj].re;
            work_R[T[ii][jj]].im = rel_nulvect[ii][jj].im;
}

/* The complex matrix rel_nulvect contains the matrix R, the imaginary parts of this matrix should be close to zero */
show_complex_matrix_f(work_R,0,K, " R");
{
    int i,j;
    for(i=0;i<K;i++){
        for(j=0;j<K;j++){
            rel_nulvect[i][j].re=work_R[j][i].re;
            rel_nulvect[i][j].im=work_R[j][i].im;
        }
    }
    for(i=0;i<K;i++){
        for(j=0;j<K;j++){
            work_R[i][j].re=rel_nulvect[i][j].re;
            work_R[i][j].im=rel_nulvect[i][j].im;
        }
    }
}
show_complex_matrix_f(work_R,0,K, " R transposed");
I_minus_R=dmatrix(0,K,0,K);
I_minus_R.Inv=dmatrix(0,K,0,K);
real_R=dmatrix(0,K,0,K);
work1=dmatrix(0,K,0,K);
work2=dmatrix(0,K,0,K);
work_vect=dvector(0,K);
{int i, j;
    for(i=0;i<=K;i++){
        work1[i][i]=mu[i];
    }
    for(i=0;i<=K;i++){
        for(j=0;j<=K;j++){
            real_R[i][j]=work_R[i][j].re;
        }
    }
}
matrix_mul(K, real_R, work1, work2);
for(i=0;i<=K;i++){
    for(j=0;j<=K;j++){
        almost_B_R_matrix[total_phases-(K-i)]
        [total_phases-(K-j)]=almost_B_R_matrix[total_phases-(K-i)]
        [total_phases-(K-j)]+work2[i][j];
    }
}
show_matrix_f(almost_B_R_matrix,0,total_phases,"B_R_matrix");
{
    int i,j;
    double *zero_vector,**work;
    ergodic_phase=dvector(0,total_phases);
zero_vector=dvector(0,total_phases);
work=dmatrix(0,total_phases,0,total_phases);
zero_vector[0]=1.000;
for(i=0;i<=total_phases;i++) almost_B_R_matrix[i][0]=1.000;
for(i=0;i<=total_phases;i++){
    for(j=0;j<=total_phases;j++){
        work[j][i]=almost_B_R_matrix[i][j];
    }
}
solve_linear_system(total_phases,work,zero_vector
,ergodic_phase);
}

/* Alternate method of calculating the level zero distributions ie P₀ */

/*

int i, j;
double temp,**tempq;
tempq = dmatrix(0,total_phases,0,total_phases);
temp=0;
for(i=0; i<=total_phases;i++){
    if(almost_B_R_matrix[i][i]<temp){
        temp=almost_B_R_matrix[i][i];
    }
}
temp*=1.5;
temp=temp;
for(i=0;i<=total_phases;i++)
    for(j=0;j<=total_phases;j++){
        if(i==j) tempq[i][j] = 1.0 + almost_B_R_matrix[i][j]/temp;
        else tempq[i][j] = almost_B_R_matrix[i][j]/temp;
    }
for(i=0;i<=10;i++)
    matrix_raul(total_phases,tempq,tempq,tempq);
for(i=0;i<=total_phases;i++)
    ergodic_phase[i]=tempq[1][i];
free_dmatrix(tempq,0,total_phases,0,total_phases);
}*/

x_0=dvector(0,total_phases-K-1);
x_1=dvector(0,K);
work_vect2=dvector(0,K);{
    int i;
    for(i=0;i<=K;i++){
        x_1[i]=ergodic_phase[total_phases-K+i];
    }
}
find_l_minus_R(work_R,l_minus_R);
assign_CID(CID,K);
solve_linear_matrix(K,l_minus_R,CID,l_minus_R_inv);
matrix_mul(K,l_minus_R,l_minus_R_inv,CID);
show_matrix_f(CID,0,K,"This should to unit matrix");
vector_matrix_mul(K,x_1,l_minus_R_inv,work_vect);
{
    int i;
    double prop_constant, temp;
    prop_constant=0;
    for (i=0;i<=total_phases-K-1;i++) {
        x_0[i]=ergodic_phase[i];
        prop_constant+=x_0[i];
    }
    for (i=0;i<=K;i++) {
        prop_constant+=work_vect[i];
    }
    temp=prop_constant;
    prop_constant=1/temp;
    for (i=0; i<=total_phases-K-1; i++) {
        x_0[i]=x_0[i]*prop_constant;
    }
    for (i=0; i<=K; i++) {
        x_1[i]=x_1[i]*prop_constant;
    }
}
matrix_mul(K,l_minus_R_inv,real_R, work1);
vector_matrix_mul(K,x_1,l_minus_R_inv, work_vect2);
vector_matrix_mul(K, work_vect2, work1, work_vect);
{
    int i;
    occupancy=0;
    for(i=0;i<=K;i++) occupancy=occupancy+work_vect[i];
}
{
    int i;
    double number_in_system;
    number_in_system=occupancy;
    for(i=1;i<=total_phases-K-1;i++) {
        number_in_system=number_in_system+(double)i*x_0[i];
    }
    for(i=0;i<=K;i++) {
        number_in_system=number_in_system+(double)(total_phases-
K+i)*x_1[i];
    }
    fprintf(fout, "\n number in the system %f\n\n", number_in_system);

63

}  
fprintf(fpout,"\n L_q is \%f \n\n",occupancy);  
free_dmatrix(L_minus_R,0,K,0,K);  
free_dmatrix(L_minus_R_last,0,K,0,K);  
free_dmatrix(real_R,0,K,0,K);  
free_dmatrix(work1,0,K,0,K);  
free_dvector(work_vec,0,K);  
P_zero=dvector(0,K);  
find_P_zero(work_R_ergodic_phase,P_zero);  
fprintf(fpout,"\n This is X_1\n");  
for (index=0;index<=K;index++) fprintf(fpout,"[\%d]: \%10.4e;\n",index,x_1(index));  
fprintf(fpout,"\n This is X_0\n");  
for (index=0;index<=total_phases-K-1;index++) fprintf(fpout,"[\%d]: \%10.4e;\n",index,x_0(index));  
printf("The vector of real parts of eigenvalues follows\n");  
for (index=0;index<=K+no_pos_mu;index++) printf("[\%d]: \%f;\n",index,eigen_value(index).re);  
printf("\n");  
printf("The vector of imaginary parts of eigenvalues follows\n");  
for (index=0;index<=K+no_pos_mu;index++)  
printf("[\%d]: \%f;\n",index,eigen_value(index).im);  
printf("\n");  
fclose(fpout);

/* End of the main program routines follow */

complex_reorder_nulls(values,vectors,index,n) c_rect *values,**vectors; int *index,n;
{
  int i,j;
  c_rect *value_temp,**vector_temp;
  value_temp = c_rect_vector(0,n-1);
  vector_temp = c_rect_matrix(0,n-1,0,n-1);

  for (i=0;i<n;i++)
    value_temp[i].re=values[i].re;
    value_temp[i].im=values[i].im;
  }
  for (i=0;i<n;i++)
    for(j=0;j<n;j++)
      vector_temp[i][j].re = vectors[i][j].re;

64
vector_temp[i][j].im = vectors[i][j].im;
}

for(i=0;i<n;i++) {
    values[i].re = value_temp[index[i]].re;
    values[i].im = value_temp[index[i]].im;
    for(j=0;j<n;j++) {
        vectors[i][j].re = vector_temp[i][index[j]].re;
        vectors[i][j].im = vector_temp[i][index[j]].im;
    }
}
free_c_rect_vector(value_temp,0,n-1);
free_c_rect_matrix(vector_temp,0,n-1,0,n-1);

complex_extract_Relevant_Eigen_Matrix_top_right(nullvcts,rel_nulvect,no_pos_mu)
c_rect **nullvcts,**rel_nulvect;
int no_pos_mu;
{
    int i,j;
    for (i=0;i<K+1;i++) {
        for (j=0;j<K+1;j++) {
            rel_nulvect[i][j].re = nullvcts[i][no_pos_mu+j].re;
            rel_nulvect[i][j].im = nullvcts[i][no_pos_mu+j].im;
        }
    }
}

complex_extract_Relevant_Eigen_Matrix_top_left(nullvcts,rel_nulvect,no_pos_mu)
c_rect **nullvcts,**rel_nulvect;
int no_pos_mu;
{
    int i,j;
    for (i=0;i<K+1;i++) {
        for (j=0;j<K+1;j++) {
            rel_nulvect[j][i].re = nullvcts[j][i].re;
        }
    }
}
rel_nulvect[j][i].im=nullvects[j][i].im;
}
}

complex_extract_Relevant_Eigen_Matrix_bottom_right(nullvects,rel_nulvect,no_pos_mu)
c_rect **nullvects,**rel_nulvect;
int no_pos_mu;
{
    int i,j;
    for (i=0;i<K+1;i++){
        for (j=0;j<K+1;j++){
            rel_nulvect[j][i].re=nullvects[no_pos_mu+j][no_pos_mu+i].re;
            rel_nulvect[j][i].im=nullvects[no_pos_mu+j][no_pos_mu+i].im;
        }
    }
}

complex_extract_Relevant_Eigen_Matrix_bottom_left(nullvects,rel_nulvect,no_pos_mu)
c_rect **nullvects,**rel_nulvect;
int no_pos_mu;
{
    int i,j;
    for (i=0;i<K+1;i++){
        for (j=0;j<K+1;j++){
            rel_nulvect[i][j].re=nullvects[i][no_pos_mu+j][j].re;
            rel_nulvect[i][j].im=nullvects[i][no_pos_mu+j][j].im;
        }
    }
}
complex_assign_CID(c, n)
c_rect **c;
int n;
{
    int i;
    for (i=0; i<=K+n; i++)
    {
        c[i][i].re=1;
    }
}

assign_CID(c, n)
double **c;
int n;
{
    int i, j;
    for (j=0; j<n+1; j++)
        for (i=0; i<=n+1; i++)
        {
            if (i==j) c[i][j]=1.0;
            else c[i][j]=0.0;
        }
}

show_complex_matrix_f(a, st, n, name)
c_rect **a;
int n, st;
char name[10];
{
    int i, j;
    if (show_detail==1)
    {
        fprintf(fput, "\n Complex matrix %s ", name);
        for (i=st; i<=n; i++)
        {
            fprintf(fput, "\n");
            for (j=st; j<=n; j++)
            {
                fprintf(fput, "%6.3f + j%6.3f \t", a[i][j].re, a[i][j].im);
            }
        }
    }

void show_matrix_f(a_st,n,name)
double **a;
int n,st;
char name[10];
{
    int i,j;

    if (show_detail==1){
        fprintf(fout,"\n \n Complex matrix \%s \".name);
        for (i=st;i<=n;i++){
            fprintf(fout,"\n");
            for (j=st;j<=n;j++){
                fprintf(fout,"%6.3f \", a[i][j]);
            }
        }
    }
}

assign_BIGA_transpose(a,t,c)
double **a;
int *t, c;
{
    int i,j,k;
    double **P_hat;
    P_hat=form_P_hat_transform(t);
    k=K;
    for (i=0;i<=k+c;i++){
        for (j=0;j<=k+c;j++){
            if (i<c && j<c)
if(i==j)
    a[i][j]=lambda[i]+mu[i]-P_hat[i][j];
else
    a[i][j]=-P_hat[i][j];
if(j>c-1 & & j<K+1 & & i<c) a[i][j]=-P_hat[i][j];
if(j>k & & i<c) {
    if (j-k-1==i)
        a[i][j]=lambda[t[i-k-1]];}
if(j>c-1 & & i<k+1 & & j>c-1 & & j<k+1){
    if (i==j) a[i][j]=lambda[i][j];
}
if(i>k & & j<c & & (i-k-1==j))
    a[i][j]=1;
}

assign_BIGB_transpose(a,t,c)
double **a;
int *t, c;
{
    int i,j,k;
double **P_hat;
P_hat=form_P_hat_transform(t);
k=K;
    for (i=0;i<=k+c;i++) {
        for (j=0;j<=k+c;j++) {
            if (i< c & & j< c) if (i==j) a[i][j]=mu[i][j];
            if (i>=c-1 & & i<k+1 & & j<c) a[i][j]=P_hat[i][j];
            if (i>=c-1 & & i<k+1 & & j>c-1 & & j<k+1) {
                if (i==j) a[i][j]=lambda[t[i][j]]+P_hat[i][j];
                else a[i][j]=P_hat[i][j];
            }
            if (i>k & & j<k & & (i==j)) a[i][j]=1;
        }
    }
}
69
fshell_index(v,index,n)
int *index,n; double *v;
{
    int gap, i, j,temp_index;
double temp,*w;
w=dvector(0,n-1);
for(i=0;i<n;i++) w[i] = v[i];
for(i=0;i<n;i++) index[i] = i;
for(gap = n/2; gap > 0; gap /= 2) {
    for (i=gap; i<n;i++) {
        for (j=i-gap;j>=0 & & w[j]<w[j+gap];j -= gap) {
            temp = w[j];
w[j] = w[j+gap];
w[j+gap] = temp;
temp_index = index[j];
index[j] = index[j+gap];
index[j+gap] = temp_index;
        }
    }
}
free_dvector(w,0,n);
}

int *find_t(address_n) int *address_n;

/* routine to define the column transformation matrix to transform mu matrix
into a matrix having it s 0 elements in last rows. */
{
    int *temp, index,count;
temp = ivector(0,K);
count = 0;
for(index=0;index<K;index++) {
    if(mu[index] > 0.0) {
        temp[ count] = index;
count += 1;
    }
}
*address_n = count;
for(index=0;index<K;index++) {
    if(mu[index] == 0.0) {
        temp[count] = index;
        count += 1;
    }
}
return temp;

double **form_P_hat_transform(T) int *T;
{
    int i,j;
    double **temp;
    temp=dmatrix(0,K,0,K);
    for (i=0; i<=K; i++) {
        for (j=0; j<=K; j++) {
            temp[i][j] = P_matrix[T[j]][T[i]];
        }
    }
    show_matrix_f(temp,0,K,"P hat transform");
    return temp;
}

find_ergodic_Phase_Prob(p,k,x)
double **p,*x;
int k;
{
    int i, j,BAK,*permut,*LH,*CNT, zero_vect;
    c_rect **eigen_vector;
    p_temp=dmatrix(1,k+1,1,k+1);
    for(i=0;i<k+1;i++) {
        for (j=0;j<k+1;j++) {
            p_temp[i+1][j+1]=p[j][i];
        }
    }
}
/* now change the rectangular matrix to a linear array for using Eigenware Corporation's scientific C routines */

LIN_P = dvector(1,(K+1)*(K+1));
matrix_lin_array(p_temp,LIIN_P+1,(K+1));
free_dmatrix(p_temp,1,k+1,1,k+1);

/* balance the target matrix */
D = dvector(1,(K+1));
LH = ivector(1,2);
balance(LIN_P+1,D+1,LH+1,(K+1));

/* now convert to Heessenberg form */
BAK = ivector(1,(K+1));
climhes(LIN_P+1,(K+1),BAK+1);

/* now find the eigenvalues and eigenvectors */
wr = dvector(1,(K+1));
wi = dvector(1,(K+1));
CNT = ivector(1,(K+1));
PV = dvector(1,(K+1)*(K+1));
elmtrans(LIN_P+1,PV+1,LH+1,BAK+1,(K+1));
hqr2(LIN_P+1,PV+1,wr+1,wi+1,LH+1,CNT+1,(K+1));
{
   int i;
   double r;
   zero_vect=1;
   r=abs(wr[1]);
   for (i=1;i<=K+1;i++)
      if(abs(wr[i])<r)
         r=wr[i];
         zero_vect=i;
   }
}

free_dvector(wr,1,(K+1));
free_dvector(LIN_P,1,(K+1)*(K+1));
balbak(PV+1,D+1,LH+1,(K+1),(K+1));
free_ivector(CNT,1,(K+1));
eigen_vector=c_rect_matrix(0,K,0,K);
assign_eigen_vectors(eigen_vector,wi+1,PV+1,K);
free_dvector(wi,1,(K+1));
free_dvector(PV,1,(K+1)*(K+1));
show_complex_matrix_f(eigen_vector,0,K,"This is Phase prop");
printf("\n smallest eigen value of the phase = \n", wr[zero_vect]);
{
    int i,j;
    for(i=0;i<k+1;i++){
        x[i]=eigen_vector[i][zero_vect].re;
    }
}
free_c_rect_matrix(eigen_vector,0,K,0,K);

assign_X(x)
double *x;
{
    int col;
    for (col=0;col<=K;col++) x[col] = 1/(double)(K+1);
}

void norm(x) double *x;
{
    int i;
    double sum;
    sum = 0.0;
    for (i=0;i<K;i++) sum += fabs(x[i]);
    if (sum != 0.0) {
        if(x[0]>0) {
            for (i=0;i<=K;i++) x[i] /= sum;
        } else {
            for (i=0;i<=K;i++) x[i] /= -sum;
        }
    }
}
assign_eigen_value(real_part, imag, eigen_values, num_ele)
  double *real_part, *imag;
  int num_ele;
  c_rect *eigen_values;
  {
    int ii;
    for (ii=0; ii<num_ele; ii++) {
      eigen_values[ii].re=real_part[ii];
      eigen_values[ii].im=imag(ii);
    }
  }

/*
 * This is a general utility to convert the storage model of an array of numbers
 * from the format a[0..n*n-1] to the format b[1..n][1..n]. The target matrix has
 * n rows and n columns with the index of the upper left hand element being (1,1)
 * and the final element being (n,n). The array a is defined in the calling routine
 * by the procedure b = dmatrix(1,n,1,n) and a is defined by a=dvector(0,n*n-1). The
 * linear array is stored by rows; i.e. b[1][1]=a[0], b[1][2]=a[1], etc.
 */
assign_eigen_vectors(b,c,a,n)
  int n; double *a,*c; c_rect **b;
  {
    int i,j,flag;
    flag = 0;
    for(i=0; i<n; i++){
      for(j=0; j<n; j++){
        if (c[j]==0) {
          b[i][j].re = a[i*n+j];
          b[i][j].im=0.0;
        }
        else {
          b[i][j].re = a[i*n+j];
          b[i][j].im=a[i*n+j+1];
          b[i][j+1].re=a[i*n+j];
          b[i][j+1].im=-a[i*n+j+1];
          j++;
        }
find_L_minus_R(work_R,L_minus_R)
double **L_minus_R; c_rect **work_R;
{
    int i,j;
    for (i=0;i<K+1;i++)
        for(j=0;j<K+1;j++)
            {
                if(i==j) L_minus_R[i][j]=1.000-work_R[i][j].re;
                else L_minus_R[i][j]=0.0-work_R[i][j].re;
            }
}

find_P_zero(work_R,ergodic_phase,P_zero)
double *ergodic_phase,*P_zero; c_rect **work_R;
{
    int i,j;
    for (i=0;i<K+1;i++)
        P_zero[i]=0;
    for(j=0;j<K+1;j++)
        {
            if(i==j){
                P_zero[i]=P_zero[i]+(1-work_R[i][i].re)*ergodic_phase[j];
            }
            else {
                P_zero[i]=P_zero[i]-work_R[i][j].re*ergodic_phase[j];
            }
        }
}
Vita

Nikhil Jain was born in Jaipur, India on January 25, 1962. He graduated from St. Joseph's Academy high school as valedictorian in 1980 and earned the bachelor's degree (with honors) in electrical engineering from the Indian Institute of Technology, Madras in 1985. On graduating, he received a grant from German Council for Technical Exchange (DAAD) to work on automation of high voltage measurements at Physikalische Technische Bundesanstalt (PTB) in Braunschweig, Germany. He subsequently earned the MBA degree with specialization in Finance and Information Systems at University of Rochester, Rochester, New York, in 1987. He has published a book chapter and an article on library information systems. He has taught the junior-level random process course at Virginia Polytechnic Institute and State University, where he entered the Ph.D. program in Operations Research in 1989. He is a member of Alpha Pi Mu.