Transient solutions in Markovian queueing systems

by

W. K. Grassmann
Purdue University

Department of Statistics
Division of Mathematical Sciences
Mimeograph Series #451

April 1976

Transient solutions in Markovian queueing systems.

Scope:

In many applications, one has to find transient solutions in queueing systems, such as parallel queues, sequential queues or queueing networks. Many of these queueing systems are Markovian, i.e. they can be formulated in terms of Markov processes. There are several methods available to find transient solutions of Markov processes. However, since queueing systems lead to Markov processes with huge, but sparse transition matrices, only methods preserving sparsity are promising. Such methods will be discussed in the paper.

Abstract:

This paper discusses the methods available to find transient solutions for huge, but sparse Markov processes, as they arise in connection with queueing systems. The methods discussed include Runge-Kutta, Liou's method, and randomization. It is shown that all these methods are closely related, but that the method of randomization is superior to the other two methods. Our own experience and experience of others clearly indicate that all the methods mentioned above are viable for finding transient solutions in problems having 600 states or more.

Introduction

A queueing system, i.e. a system consisting of one or several queues, is Markovian if it can be formulated as a Markov process. The simplest Markovian queue is the M/M/1 queue. The states of the Markov process describing this queue are i, where i is the number of elements in the system. The transition matrix of this process is very sparse: Its only non-zero elements are $a_{i\ i+1} = \lambda$ and $a_{i\ i-1} = \mu$.

If a system has d interdependent Poisson queues, the system is still Markovian, but its states are the d-tuples $i_1, i_2, \ldots, i_d$, where i_j, $j = 1,2,3,\ldots,d$, is the length of the j^{th} queue.

An Erlang queue is another example of a Markovian queueing system. The state space of the $M/E_k/1$ queue is e.g. characterized by the two numbers i_1 and i_2, where i_1 is the queue length and i_2 the phase of the server. Clearly, systems consisting of d such Erlang queues can be characterized by a Markov process whose states are described by 2 d integers $i_1, i_2, \ldots, i_{2d}$.

Even when considering only steady state solutions, Markovian queueing systems are complex. In some cases, closed solutions are available. A good description of these cases is found in Kleinrocks book [7]. However, most practical problems are too involved to be solved analytically, and a numerical approach is indicated. In particular, Schassberger [14] developped an iterative approach to find the equilibrium probabilities for two queues in parallel. His method was later used by Wallace and Rosenberg [15] for solving queueing problems arising in computer systems. Furthermore, Hillier

and Boling [3] used a different iterative method to find equilibrium solutions for sequential queues.

The main problem faced by these authors is the enormous size of the transition matrix. A queueing system leads to a Markov process whose states are the d-tuples $i_1, i_2, \ldots, i_d$. If each i_j can assume a different values, there are thus a^d different states, and even for small a and d, this number can easily exceed 1000. The transition matrix of such a process has over 1000^2 = one million entries, and that many entries are difficult to store in today's computers. Fortunately the transition matrices are usually sparse, and it is sufficient to store only the non-zero entries (see [12]).

If a process has N states, the transition matrix provides N-1 equations for the N unknown steady state probabilities. An Nth equation is obtained because the sum of all these probabilities must equal one.

When N is large, say 1000, the solution of these equations becomes difficult. A straight-forward application of Gauss-Jordan would result in N^3, or one billion operations. To do that many operations is extremely expensive, and to reduce the computer costs, one has to exploit the sparsity of the transition matrix. Unfortunately, even if one starts out with a sparse matrix, the later operations of Gauss-Jordan result in a dense matrix. For this reason, the authors cited above did not use the method of Gauss-Jordan, but other algorithms which were able to preserve sparsity better.

The problems encountered when finding steady state solutions persist when one desires to find transient solutions. Again, the transition matrices are huge, but sparse. Any efficient algorithm has to preserve sparsity. This already rules out methods using Eigenvectors, which are normally used to find transient solutions of small Markov processes [4,6]. However, there are

other methods available. Several authors [5, 9, 10, 11, 16] have employed the Runge-Kutta method for solving differential equations for finding transient solutions in queueing systems. In particular, Olson [11] observed that it takes less computer time to find transient solutions of the M/M/1 queue by Runge-Kutta than it does by using the explicit formula given in Prabhu's book [13]. Whitlock [16] solved substantial queueing networks by the same method. We ourselves used randomization as it will be described below to find transient solutions of processes with over 600 states without difficulties. Thus, finding transient solutions in queueing systems is quite feasible.

The method of Runge-Kutta.

Let $A = [a_{ij}]$ be the transition matrix of a continuous Markov process with N states. As usual, a_{ii} is defined as:

$$a_{ii} = -\sum_{j=1}^{N} a_{ij}$$

The problem is now to find $p_j(t)$, the probability of being in state j at time t, given $p_j(0) = q_j$, $j = 1,2,\ldots,N$. As is well known, the $p_j(t)$ can be found from the following differential equations:

$$p_j'(t) = \sum_{i=1}^{N} a_{ij} p_i(t) \qquad i = 1,2,\ldots,N \tag{1}$$

with

$$p_i(0) = q_i \qquad i = 1,2,\ldots,N \tag{2}$$

These equations can be solved by the method of Runge and Kutta, which particularizes in this case to:

$$\underline{p}[(g+1)h] = \underline{p}(gh) + \frac{1}{6}\underline{K}_1 + \frac{1}{3}\underline{K}_2 + \frac{1}{3}\underline{K}_3 + \frac{1}{6}\underline{K}_4 + O(h^5) \tag{3}$$

with:

$$\underline{K}_1 = \underline{p}(gh)(Ah)$$
$$\underline{K}_2 = [\underline{p}(gh) + \tfrac{1}{2}\underline{K}_1](Ah)$$
$$\underline{K}_3 = [\underline{p}(gh) + \tfrac{1}{2}\underline{K}_2](Ah)$$
$$\underline{K}_4 = [\underline{p}(gh) + \underline{K}_3](Ah)$$

In these equations, $\underline{K}_i$ are row vectors, and $\underline{p}(t)$ is equal to $[p_1(t), p_2(t),\ldots,p_N(t)]$. Equation (3) is applied recursively to calculate $\underline{p}(h), \underline{p}(eh),\ldots,\underline{p}(gh), \underline{p}[(g+1)h]$.

The essential question is how this method performs as N becomes large. In order to see this, the number of operations and the storage requirements will be evaluated.

The storage requirement, as it will be calculated for all methods under discussion, will never include the aray or arays needed to store A.

When discussing the number of operations, we will use the following conventions:

a) A vector-matrix multiplication is the multiplication of a row-vector with a matrix of size NxN.

b) A vector addition is the addition of two row vectors of length N.

c) A scalar multiplication is the multiplication of a row-vector by a scalar.

Of these operations, the vector-matrix multiplication is by far the most time-consuming one. Vector addition and scalar multiplication are about equal as far as computer time is concerned.

Storagewise, the method of Runge-Kutta requires four one-dimensional arrays of length N. Two of those arrays are required to store $\underline{p}(gh)$ and $\underline{p}[(g+1)h]$, and the remaining two are needed to store $\underline{K}_1$ and $\underline{K}_2$. $\underline{K}_3$ can be stored in the same array as $\underline{K}_1$. To do this, one adds $\frac{1}{6}\underline{K}_1$ to the appropriate term in equation 3 as soon as $\underline{K}_2$ has been calculated. Then, $\underline{K}_1$ is no longer needed, and its space is available to store $\underline{K}_3$. For similar reasons, it is possible to store $\underline{K}_4$ in the array originally used to store $\underline{K}_2$.

The total number of operations to evaluate equation (3) is 4 vector-matrix multiplications, corresponding to the 4 $\underline{K}_i$, 7 vector additions, corresponding to the 7 plus signs, and 6 scalar multiplications. Also, A has to be multiplied by h, but this has to be done only once and need not be repeated for any $g > 1$.

As mentioned, the method of Runge-Kutta was successfully employed to solve a variety of queueing problems. Whitlock [16] used it actually for systems with non time-homogeneous transition matrices. In this case,

equation (3) has to be slightly modified. The numerical experience reported by the authors cited indicates that Runge-Kutta is a viable method for calculating transient solutions. Whitlock [16] also found that Runge-Kutta is much faster than simulation.

Modified Runge-Kutta and Liou's method

It is possible to write equation (3) in a more compact way. One finds by simple substitution:

$$\underline{K}_1 = \underline{p}(gh)(Ah)$$

$$\underline{K}_2 = \underline{p}(gh)(Ah) + \frac{1}{2}\underline{p}(gh)(Ah)^2$$

$$\underline{K}_3 = \underline{p}(gh)(Ah) + \frac{1}{2}\underline{p}(gh)(Ah)^2 + \frac{1}{4}\underline{p}(gh)(Ah)^3$$

$$\underline{K}_4 = \underline{p}(gh)(Ah) + \underline{p}(gh)(Ah)^2 + \frac{1}{2}\underline{p}(gh)(Ah)^3 + \frac{1}{4}\underline{p}(gh)(Ah)^4$$

Using these values for $\underline{K}_i$, equation (3) can be written:

$$\underline{p}[(g+1)h] = \underline{p}(gh) + \frac{1}{6}\underline{K}_1 + \frac{1}{3}\underline{K}_2 + \frac{1}{6}\underline{K}_4 + 0(h^5)$$

$$= \underline{p}(gh)+\underline{p}(gh)(Ah) + \frac{1}{2}\underline{p}(gh)(Ah)^2 + \frac{1}{6}\underline{p}(gh)(Ah)^3+\frac{1}{24}\underline{p}(gh)(Ah)^4+0(h^5). \quad (5)$$

This method shall be called the modified Runge-Kutta method.

Before discussing this equation in detail, we introduce Liou's approach, which is as follows:

The solution of differential equations (1), subject to the initial conditions (2) can be written as:

$$\underline{p}(t) = \underline{p}(0)\ \exp(At) = \underline{q}\ \exp(At).$$

Here, $\underline{q} = [q_1, q_2, \ldots, q_N]$. $\underline{p}(t)$ can now be found from the following power series expansion:

$$\underline{p}(t) = \sum_{n=0}^{\infty} \underline{q}(At)^n/n! = \sum_{n=0}^{m-1} \underline{q}(At)^n/n! + R_m. \quad (6)$$

Here, R_m is a row vector which indicates the trancation error, given one only retains the first m terms of the power series. Liou now suggested to evaluate $\underline{p}(t)$ by equation (6). It turns out that the modified Runge-Kutta method is a special case of Liou's method.

First, since the process is time-homogeneous, one finds for $t_2 > t_1$ after having shifted the origin of the time axis to t_1:

$$\underline{p}(t_2) = \underline{p}(t_1)\exp[A(t_2-t_1)] = \sum_{n=0}^{m-1} \underline{p}(t_1)A^n(t_2-t_1)^n/n! + R_m. \qquad (7)$$

If m is replaced by 5, t_1 by gh and t_2 by (g+1)h, this expression is exactly the same as equation (5), which shows that (6) implies (5).

Since Liou's method is a generalization of the modified Runge-Kutta method, it is sufficient to discuss Liou's method.

The terms $\underline{q}(At)^n/n!$ of Liou's method can be calculated recursively as:

$$\underline{q}(At)^n/n! = \{[\underline{q}(At)^{n-1}/(n-1)!]\cdot A\}t/n.$$

To do this recursion, one has one vector-matrix multiplication, and one scalar multiplication for multiplying the result by t/n. To calculate the m terms $\underline{q}$, $\underline{q}(At)/1!$, $\underline{q}(At)^2/2!,\ldots,\underline{q}(At)^{m-1}/(m-1)!$, this recursion has to be applied m-1 times, giving (m-1) vector matrix multiplications and (m-1) scalar multiplications. To obtain $\underline{p}(t)$, one has to sum all these terms which gives (m-1) vector addition. In the modified Runge-Kutta method, one has m=5, which gives four vector matrix multiplications, four scalar multiplications and four vector additions. This compares with four vector matrix multiplications, 7 vector additions and 6 scalar multiplications in the direct Runge-Kutta method. The modified Runge-Kutta method results thus in a considerable saving in computer time.

Furthermore, Liou's method can be done using three one-dimensional arrays only: One is used to store the vectors $\underline{q}(At)^n/n!$, one to store $\underline{q}(At)^{n-1}/(n-1)!$, and one to calculate $\underline{p}(t)$. This implies also that the modified Runge-Kutta needs less storage than the original Runge-Kutta.

Transient solutions are primarily interesting for systems that converge slowly toward their steady state. In such cases, $\underline{p}(t)$ for high t has to be evaluated. For high t, Liou's method results in substantial round-off errors. This is so because $\underline{q}At$, and, with it, $\underline{q}(At)^n/n!$ have negative elements. When summing up the terms, which are huge for large t, the negative and positive elements almost cancel each other out, because the sum of all terms must be a probability vector which has elements between zero and one. There are two possibilities to avoid having differences between large numbers. One is to break down the interval from 0 to t into several subintervals $[0,t_1)$, $[t_1,t_2),\ldots,[t_i,t)$, and calculate the $\underline{p}(t_i)$ recursively using equation (7). Alternatively, one can look for methods not requiring any subtractions. Such a method will be described in the next section.

Randomization

As mentioned above, the negative diagonal elements of A cause high round-off errors when calculating $\underline{p}(t)$ according to equation (6). It is now possible to find $\underline{p}(t)$ by a series expansion of a matrix P which contains no negative elements.

To do this, let P be equal to:

$$P = A/F + I \tag{8}$$

Here, $F \geq |a_{ii}|$, but otherwise an arbitrary number.

The off-diagonal elements of P are non-negative as are the ones of A. The diagonal elements of P are:

$$p_{ii} = a_{ii}/F+1 \geq a_{ii}/|a_{ii}| + 1 = 0.$$

Thus, the p_{ii} are non-negative as well.

The row sums of P are one:

$$\sum_{j=1}^{N} p_{ij} = \sum_{j=1}^{N} a_{ij}/F+1 = 0+1$$

Consequently, P is a stochastic matrix.

One now finds for $\underline{p}(t)$

$$\underline{p}(t) = \underline{q}\exp(At) = \underline{q}\exp[(A/F+I)Ft-IFt] = \underline{q}\exp(PFt)\exp(-Ft)$$

$$= \sum_{n=0}^{\infty} \underline{q}P^n[(Ft)^n \exp(-Ft)/n!] = \sum_{n=0}^{m-1} \underline{q}P^n[(Ft)^n \exp(-Ft)/n!]+R_m. \quad (9)$$

The method given by formula (9) is called randomization, because it can be interpreted as a discrete Markov process imbedded in a Poisson process. The Poisson process generates Poisson events at a rate F, and the probability of having n events during an interval of length t is thus $(Ft)^n \exp(-Ft)/n!$. $\underline{q}P^n$ gives the probability vector of being in state j in a discrete Markov process which has the transition matrix P. Consequently, $[(Ft)^n\exp(-Ft)/n!]qP^n$ gives the probability vector of having n Poisson-events and being in state j after these events, and summing these vectors gives $\underline{p}(t)$.

The trancation error of equation (9), R_m, can be estimated with reasonable accuracy. Actually, for each probability $p_j(t)$, one has:

$$p_j(t) = \sum_{n=0}^{m-1} p_j^{(n)}[(Ft)^n\exp(-Ft)/n!] + \sum_{n=m}^{\infty} p_j^{(n)}[(Ft)^n\exp(-Ft)/n!]$$

Here, $p_j^{(n)}$ is the jth component of the vector $\underline{q}P^n$, or, what is the same, it is the probability of being in state j after n Poisson-events. Clearly, $p_j^{(n)} \leq 1$, and R_{mj}, the j^{th} element of R_m becomes:

$$R_{mj} = \sum_{n=m}^{\infty} p_j^{(n)}[(Ft)^n \exp(-Ft)/n!] \leq \sum_{n=m}^{\infty} (Ft)^n \exp(-Ft)/n!$$

The sum to the right is the complementary cumulative Poisson distribution. It follows:

$$R_{mj} \leq 1 - \sum_{n=0}^{m-1} (Ft)^n \exp(-Ft)/n!$$

For small Ft, the sum to the right can easily be evaluated for all m of interest, and m can thus be determined such that R_{mj} is below a prescribed value α. For large Ft, one can approximate the Poisson distribution by the normal distribution. In our programs, we set m equal to:

$$m = Ft + 4\sqrt{Ft} + 5 \qquad (10)$$

We found from Poisson tables ($Ft \leq 20$) and from normal tables ($Ft > 20$) that this choice of m guaranties that R_{mj} is less than 10^{-4}.

Equation (10) suggests that F should be as small as possible. Consequently, we set:

$$F = \max|a_{ii}|, \qquad (11)$$

which is the smallest value F can assume.

When calculating $\underline{p}(t)$ for two values of t, say t_1 and t_2, one can apply randomization twice, as is suggested by equation (7). Preferably, one calculates $\underline{p}(t_1)$ and $\underline{p}(t_2)$ simultaneously from equation (9). This requires an additional one-dimensional array to store $\underline{p}(t_2)$, but it avoids some calculations because of the nature of (10): m as a function of t grows slower when t is high than when it is low, making it advantageous to have long intervals before restarting randomization in the fashion suggested by (7).

For a given m, the storage requirements and the calculation times of randomization and Liou's method are practically identical. In both cases, one has (m-1) vector-matrix multiplications, (m-1) scalar multiplications and (m-1) vector additions. To see this, note the $\underline{q}P^n$ can be calculated recursively, giving one vector-matrix multiplication for each term, the resulting vector can be multiplied by the Poisson-distribution, a scalar, and added, giving the operations just mentioned before. The storage requirement is also 3 vectors (provided, of course, one calculates $\underline{p}(t)$ for one t only).

By numerical experimentation and theoretical considerations, we found that for givem m, the trancation error R_m in randomization is usually smaller than in Liou's method. This is what one would expect. Really, P, being a stochastic matrix, resembles the result matrix, exp(At), which is also stochastic. These two matrices are in a way close together, whereas A, which is not stochastic, is more distant. Unfortunately, we cannot give more details on the topic of convergency because the numerical results do not make sense without the theoretical background, and the theoretical background involves complex arguments based on the Eigenvalues of the matrices in question. Also, compared with the problem of roundoff-errors, the question of convergency is not that crucial.

Numerical results.

We wrote a program to implement the method of randomization, and solved a number of problems with this program. The program was written in FORTRAN G and run on an IBM 370/158.

The results of these runs are given in table 1. The actual nature of the problems run is of minor importance. We just note that the structure of these problems was varied. The first problem deals with 3 parallel queues, the second with 3 sequential queues, the third is the M/M/1 queue, the fourth problem is the $M/E_3/2$ queue, etc.

The performance of the algorithm depends on the size of the transition matrix A as it is determined by the number of states. More important than the size of A are the number of non-zero entries, since only they are used when calculating the vectors $\underline{q}P^n$ recursively.

Of the problems considered, two were of such a dimension that storing the entire transition matrix would have been problematic. In particular, the transition matrix of problem 7 has 625x625 = 390,625 elements, whereas the transition matrix of problem 8 has 459x459 = 210,681 elements. In both cases, matrix inversion or calculation of Eigenvectors is close to impossible. Since the method suggested above preserves sparsity, the problem of storing and/or calculating matrices of such huge dimensions is unnecessary.

All the problems summarized in table 1 were run with a high value for Ft. In particular, problem 2 has an Ft 10000, and problem 7, the largest problem as far as the number of states is concerned, has an Ft of 390. The usage of high Ft is necessary in processes that converge slowly toward equilibrium, and these processes are exactly the ones for which the transient behaviour is interesting.

The value m for which the power series was trancated was calculated according to equation (10). This value converges toward Ft as Ft becomes large. Then highest value for m was 10,405, followed by 474.

Table 1: Summary of some problems solved.

Problem number	1	2	3	4	5	6	7	8
number of states	216	72	21	240	121	125	625	459
non-zero entries in transition matrix	971	144	61	725	461	525	4625	1681
F	9	4	13	15	23	19	39	9
Ft	36	10000	52	60	115	95	390	36
m	65	10405	86	96	163	139	474	65
CPU-time (seconds)	6.2	34.7	0.7	8.3	4.2	3.8	77.0	17.8

p_n, the Poisson probabilities, were calculated as follows: First, $[(Ft)^n/n!]/[(Ft)^i/i!]$ was evaluated recursively, for $n \geq i$. Afterwards, the numbers obtained this way were divided by an appropriate constant, such that their sum was one. Thus, we really calculated p_n (except for a scaling factor) for $n \geq i$. Here, i was chosen such that the sum of the p_n, $n < i$, was negligible. In this way, exponent underflows and overflows are avoided. On the other hand, the sum of the Poisson probabilities under the method is always 1.

We mention this numerical detail for the following reason:

To check the accuracy of the results, we added all probabilities of some of the problems, and calculated by how much this sum deviates from 1. In problem 1, this deviation was e.g. 0.000036, and in problem 2 it was 0.000016. These deviations are due to roundoff-errors, and, because of the

way the Poisson distribution was calculated, from roundoff-errors arising from calculation $\underline{q}P^n$. These roundoff-errors are small. To see this, note that the IBM 370/158 has only 8 significant digits. Furthermore, when the sum of all deviations added together is only 0.000036 respectively, 0.000016, the deviations of the individual probabilities from their correct values are presumably much smaller.

The execution times on the IBM were negligible. In particular, it took only 0.7 seconds to find the transient solution of the M/M/1 queue. Even the largest problem, problem 7, required only 77 seconds.

The numbers presented in table 1 clearly indicate that it is feasible to calculate transient solutions for a wide variety of queueing problems. The M/M/1 queue, e.g., turns out to be a trivial problem. However, none of the other problems taxed the capabilities of the program fully. The times recorded seem to indicate that it is still feasible to solve problems 10 times as large as the largest problem solved by us. In other words: Even for problems having 6000 states and 5000 non-zero transitions, it should be feasible to find transient solutions.

References

[1] Feller, W. An Introduction to Probability Theory and Its Applications, Vol. I, Second Edition, John Wiley 1962, page 380.

[2] Feller, W. An Introduction to Probability Theory and Its Applications, Vol. II, John Wiley 1966, page 321.

[3] Hillier, F. S., Boling, R. W., Finite queues in series with exponential or Erlang service times--a numerical approach, Operations Research, 2(March), Vol. 15(1967), pp. 286.

[4] Howard R. A. Dynamic Programming and Markov Processes, MIT Press, 1960.

[5] Huk, J., On the numerical determination of the state distribution in the system M/M/1 (in Polish), Zastosowania Matematyki, 11, p. 423.

[6] Karlin, S., A First Course in Stochastic Processes, Academic Press, 1966, page 208.

[7] Kleinrock, L., Queueing Systems, Vol. I, John Wiley, 1975.

[8] Liou, M. L., A novel method of evaluating transient response, Proceedings of the IEEE, Vol. 54 (1966), No. 1, p. 20.

[9] Liitschwager, J., Ames, W. F., On transient queues-Practice and Pedagogy, Proc. 8th Symp. Interface, Los Angeles, CA, p. 206.

[10] Neuts, M. F., Algorithms for the waiting time distribution under various queue disciplines in the M/G/1 queue with service time distribution of phase type, Dept. of Statistics, Mimeograph Series No. 420, July 1975.

[11] Olson, S. W., The numerical solution of transient queueing problems, Tech. Rept. PAA-TR1-72, Systems Analysis Div, U.S. Army Weapons Command, Rock Island, IL.

[12] Pooch, U. W., Nieder, A., A survey of indexing techniques for sparse matrices, Computing surveys, Vol. 5, No. 2, June 1973, page

[13] Prabhu, N. U., Queues and Inventories, John Wiley 1965, page 19.

[14] Schassberger, R., Ein Warteschlangensystem mit zwei parallelen Warteschlangen, Computing 3, 110,124 (1968).

[15] Wallace, W. L., and Rosenberg, R. S., Markovian models and numerical analysis of computer systems behaviour, Proceedings, Spring Joint Computer Conference, 1966, pp. 141.

[16] Whitlock, J. S., Modeling computer systems with time-varying Markov chains, Ph.D. Thesis, Univ. of North Carolina, 1973.

Acknowledgements:

It is gratefully acknowledged that this study was partly supported by the NRC of Canada (Grant A8112). This grant allowed me to hire some capable students that searched the literature for me and that did the main programming jobs. Particular valuable work in this respect was done by the students Rodney Churchman and Mario DeSantis. Finally, I would like to thank Marcel Neuts from the Purdue University for his encouragement and his constructive criticism.