1

EMPIRICAL COMPARISON OF UNIFORMIZATION METHODS FOR CONTINUOUS-TIME MARKOV CHAINS

John D. Diener[†] **and William H. Sanders**[‡]

[†] *Department of Electrical and Computer Engineering*
The University of Arizona
[‡] *The Center for Reliable and High-Performance Computing*
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

ABSTRACT

[1] Computation of transient state occupancy probabilities of continuous-time Markov chains is important for evaluating many performance, dependability, and performability models. A number of numerical methods have been developed to perform this computation, including ordinary differential equation solution methods and uniformization. The performance of these methods degrades when the highest departure rate in the chain increases with respect to a fixed time point. A new variant of uniformization, called adaptive uniformization (AU), has been proposed that can potentially avoid such degradation, when several state transitions must occur before a state with a high departure rate is reached. However, in general, AU has a higher time complexity than standard uniformization, and it is not clear, without an implementation, when AU will be advantageous. This paper presents the results of three different AU implementations, differing in the method by which the "jump probabilities" are calculated. To evaluate the methods, relative to standard uniformization, a *C++* class was developed to compute a bound on the round-off error incurred by each implementation, as well as count the number of arithmetic instructions that must be performed, categorized both by operation type and phase of the algorithm they belong to. An extended machine-repairman reliability model is solved to illustrate use of the class and compare the adaptive uniformization implementations with standard uniformization. Results show that for certain models and mission times,

[1]From Computations with Markov Chains, ed. William J. Stewart, Kluwer Academic Publishers, Boston, pp. 547-570.

adaptive uniformization can realize significant efficiency gains, relative to standard uniformization, while maintaining the stability of standard uniformization.

1 INTRODUCTION

Continuous-time Markov chains are often used to evaluate the performance, dependability, and performability of computer systems and networks. For many models [5, 6], transient measures are of particular interest. Closed-form solutions exist for certain models, but often numerical approaches must be utilized. These include ordinary differential equation methods such as Runge-Kutta-Fehlberg or an implicit method TR-BDF2 [17], Jensen's method [9] (commonly known as *uniformization*), and combinations thereof [12].

Uniformization has often been considered the best method [8, 18] for computing transient state probabilities of continuous-time Markov chains, but suffers a dramatic degradation in performance as the highest state departure rate increases for any fixed time point [12]. For these so-called "stiff models," the truncation point needed to achieve acceptable accuracy becomes prohibitively large. *Adaptive uniformization* (AU) [14] was introduced to avoid this problem in models where states with high transition rates are not reached for several state transitions, and is particularly applicable for models with short mission times, where ordinary differential equation methods are inefficient. Additionally, AU shares a nice feature with uniformization in that an error bound can be prespecified, and computes an absolute lower bound result within the specified error. Adaptive uniformization takes advantage of the fact that there is typically a single initial state, and several transitions are necessary until all states are reachable, or *active*. Thus the uniformization rate at each step need only be larger than the largest outgoing rate from the set of currently active states, not larger than the largest rate in the entire state space. This allows bigger *jumps* to be made, reducing the number of iterations necessary to reach a solution.

While adaptive uniformization will always take a number of iterations less than or equal to standard uniformization to achieve a specified level of accuracy, each iteration requires more computation, so it is not clear when AU will be computationally advantageous. Theoretical results in [14] suggest that there are situations when this will be the case, but no implementations have been made that show that this is indeed true, or the magnitude of savings (or expenses) that may be incurred. In this paper, we present the results of three different implementations of adaptive uniformization, and compare them with regular uniformization. The three implementations differ in how the "jump probabilities" are calculated, the most critical part of the adaptive uniformization algorithm. In order to make a careful assessment of the various implementations, a *C++* class was developed to instrument the code. This new class calculates a bound on the round-off error incurred by each method, and the number of floating point operations required for solution of a model. The number of operations is broken down by operation type (i.e., multiplication, division, addition, and subtraction) and by the portion of the algorithm the operations are associated with. The fine-grained nature of this data collection permits a clear understanding of the exact costs of the different portions of each algorithm, as well as when it will encounter numerical difficulties.

The contributions of the paper are two-fold. First, we develop a method for instrumenting algorithms to gain a clear understanding of their exact costs and numerical issues. The *C++* class developed to do this is algorithm independent, and can be easily added to any algorithm implemented in *C* or *C++*. Second, we provide an accurate comparison, using the instrumented code, of the three adaptive uniformization implementations and standard uniformization in the context of a delayed machine repairman model. The results show that the advantages/disadvantages of the new approach are clearly problem dependent, but there are realistic situations where adaptive uniformization is dramatically more efficient than standard uniformization, while retaining the stability of standard uniformization.

The remainder of the paper is organized as follows. After the theoretical background of uniformization and adaptive uniformization is reviewed in the next section, the details of computing the jump probabilities for AU are given in Section 3. Three methods, ACE [16], a modified ACE algorithm [14] and uniformization, are discussed in the context of our implementation. Section 4 compares and describes the overall algorithm implementation, while in Section 5 our *C++* class for evaluating the viability of the methods and results is explained. Finally, Section 6 details results based upon instrumentation, empirically comparing uniformization and adaptive uniformization.

2 BACKGROUND

We begin by explaining the theory behind the methods used in this paper to solve for the transient state probabilities of continuous-time Markov chains

(CTMCs). Let CTMC $Y = \{Y(s); s \geq 0\}$ be defined on the state space S where it is assumed $S = \{0, 1, 2, \ldots\}$. Let $Q = (q(i,j)), i, j \in S$, be the infinitesimal generator matrix of Y, with $q_i = -q(i,i)$ for $i \in S$. The row vector $\underline{\pi}(0) = (\pi_0(0), \pi_1(0), \ldots)$ denotes the initial probability distribution of S. Our objective is then to obtain the transient state probability vector $\underline{\pi}(t)$, with $\pi_i(t) = \text{Prob}\{Y(t) = i\}$ for any $i \in S$.

Kolmogorov's differential equations [3] show that $\underline{\pi}(t)$ follows

$$\frac{d\underline{\pi}(t)}{dt} = \underline{\pi}(t)Q. \tag{1.1}$$

Therefore, this problem has the solution $\underline{\pi}(t) = \underline{\pi}(0)e^{Qt}$. However, directly solving e^{Qt} does not lead to satisfactory numerical algorithms [7].

In standard uniformization (SU), which is often used in the context of Markov models, the CTMC is decomposed into a discrete time Markov chain (DTMC) and a Poisson process. For later use, we briefly review standard uniformization. Let $\lambda \geq \max\{q_i\}, i \in S$, be the so-called *uniformization rate*. Then, the DTMC $X = \{X_n, n = 0, 1, \ldots\}$, can be defined, with transition matrix $P = I + (1/\lambda)Q$. $\underline{\pi}(t)$ is then obtained from

$$\underline{\pi}(t) = \underline{\pi}(0)\sum_{n=0}^{\infty} e^{-\lambda t}\frac{(\lambda t)^n}{n!}P^n. \tag{1.2}$$

By introducing $\underline{\pi}_n = \underline{\pi}_{n-1}P, n = 1, 2, \ldots,$ where $\underline{\pi}_0 = \underline{\pi}(0)$, Equation 1.2 can be written recursively as

$$\underline{\pi}(t) = \sum_{n=0}^{\infty} U_n(t)\underline{\pi}_n \tag{1.3}$$

where for later use we have defined the *jump probabilities* $U_n(t)$ as

$$U_n(t) = e^{-\lambda t}\frac{(\lambda t)^n}{n!}. \tag{1.4}$$

$\underline{\pi}_n$ is interpreted as the state probability distribution of X after n state transitions. Thus $\underline{\pi}(t)$ can be thought of as the discrete probability vector after n jumps multiplied by the probability of having n transitions occur, summed over all possible number of jumps. In SU, these jump probabilities form a Poisson distribution. In our implementation of SU, the infinite sum of Equation 1.3 is both left and right truncated so that

$$\underline{\pi}(t) \approx \sum_{n=N_{sl}}^{N_{sr}} e^{-\lambda t}\frac{(\lambda t)^n}{n!}\underline{\pi}_n, \tag{1.5}$$

and

$$\sum_{n=0}^{N_{sl}-1} e^{-\lambda t}\frac{(\lambda t)^n}{n!} + \sum_{n=N_{sr}+1}^{\infty} e^{-\lambda t}\frac{(\lambda t)^n}{n!} < \epsilon. \tag{1.6}$$

N_{sl} and N_{sr} are then selected so that the resulting *truncation error* is less than an $\epsilon > 0$. Notice that this left truncation does not effect the number of matrix-vector multiplications that must occur, only reducing the amount of summation necessary. Fox and Glynn [4] have given a method to find N_{sl} and N_{sr} given a predefined ϵ, λ, and t, and using this, SU allows solution of $\underline{\pi}(t)$ to any desired accuracy $\epsilon > 0$.

In adaptive uniformization, the uniformization rate can change with the number of jumps, or current *epoch*, and depends on the set of states the DTMC X can be in at some epoch. The set of *active states at epoch* $n, n = 0, 1, \ldots$, of X is defined as $A_n \subseteq S$ where

$$A_n = \{i \in S \mid \underline{\pi}_n(i) > 0\}. \tag{1.7}$$

Then the *adapted uniformization rates* can be defined as $\lambda_n \geq \max\{q_i \mid i \in A_n\}$, for $n = 0, 1, \ldots$. Restricted state-transition rate matrices for each step n, called *adapted infinitesimal generators* in [14], can be defined as the $Q_n = (q_n(i,j)), i, j \in S$, such that

$$q_n(i,j) = \begin{cases} q(i,j) & \text{if } i \in A_n \\ 0 & \text{otherwise} \end{cases}$$

Only rows of Q corresponding to active states in each epoch are considered. Next, the *adapted transition matrices* are given by

$$P_n = I + \frac{1}{\lambda_n} Q_n, n = 0, 1, \ldots. \tag{1.8}$$

If $U_n(t)$ is defined to be the probability of n jumps in time t in the birth process formed by the adaptive uniformization rates $\lambda_0, \lambda_1, \ldots$, then the transient state probability vector can be defined as

$$\underline{\pi}(t) = \underline{\pi}(0) \sum_{n=0}^{\infty} U_n(t) \prod_{i=0}^{n-1} P_i = \sum_{n=0}^{\infty} U_n(t)\underline{\pi}_n, \tag{1.9}$$

where

$$\underline{\pi}_n = \underline{\pi}_{n-1} P_{n-1}, n = 1, 2, \ldots \text{ with } \underline{\pi}_0 = \underline{\pi}(0). \tag{1.10}$$

As with SU, the sum is truncated so that

$$\underline{\pi}(t) \approx \sum_{n=0}^{N_a} U_n(t)\underline{\pi}_n \text{ with } \sum_{n=N_a+1}^{\infty} U_n(t) < \epsilon. \tag{1.11}$$

Left truncation is not employed since the jump probabilities no longer form a Poisson distribution. Like SU, AU solutions can be found to any accuracy $\epsilon > 0$. By using a pure birth process to find the jump probabilities, it is possible to make probabilistically larger jumps with AU than SU. Thus, for identical ϵ and t, $N_a \leq N_{sr}$, potentially resulting in a reduction of computation time.

3 JUMP PROBABILITY CALCULATION

The basic difference between AU and SU lies in calculation of the jump probabilities. For SU, the jump process has a Poisson distribution, while AU's takes the form of a general birth process. It can be defined by $B = \{B(t); t \geq 0\}$ over the state space $S = \{0, 1, \ldots, N_a\}$. Then $U_n(t) = \text{Prob}\{B(t) = n\}$, with B's infinitesimal generator matrix Q_B of the form

$$Q_B = \begin{pmatrix} -\lambda_0 & \lambda_0 & 0 & \ldots & \ldots & 0 \\ 0 & -\lambda_1 & \lambda_1 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & 0 & -\lambda_{N_a-1} & \lambda_{N_a-1} \\ 0 & \ldots & \ldots & \ldots & 0 & -\lambda_{N_a} \end{pmatrix} \tag{1.12}$$

where λ_i is as defined in Section 2. In this section, the three alternatives used in this study for computing the jump probabilities for AU, an acyclic Markov chain evaluator (ACE) developed by [19] then refined in [16], a modified ACE scheme from [14], and standard uniformization, will be discussed. Special attention will be given to issues related to their implementation in the context of having no prior knowledge of the adaptive uniformization rates.

ACE ACE is applicable to solving this problem both because of B's acyclic nature and that the ACE algorithm is recursive. The method is based upon the fact that

$$U_i(t) = \pi_{B_i}(t) = \sum_{\gamma_j \in \Gamma} \sum_{k=0}^{K_i(\gamma_j)} a_{i,j,k} t^k e^{\gamma_j t} \tag{1.13}$$

for an acyclic CTMC, where , is the set of unique poles and $K_i(\gamma_j)$, given $\gamma_j \in$, , is the number of poles with value γ_j in the Laplace transform expression [16]. The coefficients a are indexed by i, j, and k, representing the ith state, jth unique pole, and $(k+1)$th appearance of the jth pole. Thus γ_j represents the jth unique pole. Calculation of $a_{i,j,k}$ can be accomplished recursively through the method of [16]. In this study we have incorporated numerical improvements shown in [11] and [10] to reduce cancellation error and chance of overflow. First, the generator matrix $Q_B = (q_B(i,j)), i,j \in S$, is scaled by the mission time t, resulting in

$$\overline{Q}_B = tQ_B,\ \overline{t} = 1.0,\ \overline{\gamma}_j = t\gamma_j \qquad (1.14)$$

and thus

$$\overline{\lambda}_i = t\lambda_i = -\overline{q}_B(i,i),\ \text{for } i = 0,1,\ldots,N_a. \qquad (1.15)$$

Second, we have for the initial state 0

$$\overline{\gamma}_1 = -\overline{\lambda}_0,\ K_0(\overline{\gamma}_1) = 0,\ \overline{a}_{0,1,0} = e^{-\overline{\lambda}_0}. \qquad (1.16)$$

Finally, the other coefficients are derived recursively by the following scheme starting with the second state

$$\overline{a}_{i,j,k} = \begin{cases} \overline{a}_{i-1,j,k}\frac{\overline{\lambda}_{i-1}}{\overline{\gamma}_j+\overline{\lambda}_i} & \text{iff } \overline{\lambda}_i \neq \overline{\gamma}_j,\ k = K_i(\overline{\gamma}_j) \quad (a) \\ \overline{a}_{i-1,j,k}\frac{\overline{\lambda}_{i-1}}{\overline{\gamma}_j+\overline{\lambda}_i} - (k+1)\frac{\overline{a}_{i,j,k+1}}{\overline{\gamma}_j+\overline{\lambda}_i} & \text{iff } \overline{\lambda}_i \neq \overline{\gamma}_j,\ 0 \leq k < K_i(\overline{\gamma}_j) \quad (b) \\ \overline{a}_{i-1,j,k-1}\frac{\overline{\lambda}_{i-1}}{k} & \text{iff } \overline{\lambda}_i = \overline{\gamma}_j,\ 1 \leq k \leq K_i(\overline{\gamma}_j) \quad (c) \\ -\sum_{\overline{\gamma}_m \in \overline{\Gamma}, \overline{\gamma}_m \neq -\overline{\lambda}_i} \overline{a}_{i,m,0}e^{-\overline{\lambda}_i-\overline{\gamma}_m} & \text{otherwise} \quad (d) \end{cases} \qquad (1.17)$$

Given these improvements from [10], computing $\pi_{B_i}(t)$ from the ACE coefficients is a simple matter of summation, so

$$\pi_{B_i}(t) = \sum_{\overline{\gamma}_j \in \overline{\Gamma}} \sum_{k=0}^{K_i(\overline{\gamma}_j)} \overline{a}_{i,j,k}. \qquad (1.18)$$

In our implementation the sum of Equation 1.17 (d) is calculated such that positive and negative terms are added in ascending order according to their absolute values, with the addition of these partial results equalling the total sum, thereby reducing cancellation error [10]. This error-reduction scheme is not employed in our $\pi_{B_i}(t)$ summation. The additional overhead is high because of ACE's complexity, and more importantly, it would not contribute to the stability of the method. Since the coefficients for the current state's transient probability are only dependent upon the previous state's, ACE for this specialized type of acyclic CTMC can be implemented dynamically, with only two sets of coefficients held in memory simultaneously. However, each new iteration adds another coefficient, making ACE an $O(N_a^2)$ algorithm.

Modified ACE To reduce the computational complexity from $O(N_a^2)$ to $O(N_a)$, [14] introduced a modified version of ACE for when λ_i converges to a fixed rate λ. Convergence is typical behavior in AU and is also a by-product of all states becoming active. Although it is not necessarily the case if the strict rules of adaptive uniformization are followed, our implementation designates the maximum rate λ to be the converged rate. As soon as the state with rate λ becomes active, the birth process is assumed to have converged. As will be seen, this notion of convergence is important to making adaptive uniformization computationally beneficial.

So, if convergence occurs at epoch m, it can be said $\lambda_{m+l} = \lambda$ for $l = 0, 1, \ldots$, implying that the birth process B behaves as a Poisson process from epoch m onwards. By convolving a hypo-exponential and Poisson distribution, and taking into account scaling by t, an equation for $U_{m+l}(t)$ can be derived:

$$\begin{aligned} U_{m+l}(t) &= \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \overline{b}_{m,j,k} (-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda}+\overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{\overline{\gamma}_j} \\ &- \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \overline{b}_{m,j,k} (-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda}+\overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{-\overline{\lambda}} \cdot \\ &\sum_{v=0}^{r+l} \frac{(\overline{\lambda}+\overline{\gamma}_j)^v}{v!}, \end{aligned} \tag{1.19}$$

given that $\overline{\lambda} \neq \overline{\lambda}_i$ for $0 \leq i < m$. By creating terms A, C and D, $U_{m+l}(t)$ can be computed recursively, with a constant number of operations per step. So, let

$$A^{(\zeta)}_{\overline{\gamma}_j,k,r}(l) = \overline{b}_{m,j,k}(-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda}+\overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{\zeta} \tag{1.20}$$

$$C_{\overline{\gamma}_j,r}(l) = \sum_{v=0}^{r+l} \frac{(\overline{\lambda}+\overline{\gamma}_j)^v}{v!} \tag{1.21}$$

for $l \geq 0$. Recursive relations, necessary for implementation, for $l > 0$ are

$$A^{(\zeta)}_{\overline{\gamma}_j,k,r}(l) = A^{(\zeta)}_{\overline{\gamma}_j,k,r}(l-1) \frac{\overline{\lambda}}{(\overline{\lambda}+\overline{\gamma}_j)} \frac{(r+l)}{l} \tag{1.22}$$

$$C_{\overline{\gamma}_j,r}(l) = C_{\overline{\gamma}_j,r}(l-1) + D_{\overline{\gamma}_j,r}(l) \tag{1.23}$$

where

$$D_{\overline{\gamma}_j,r}(l) = \frac{(\overline{\lambda}+\overline{\gamma}_j)^{r+l}}{(r+l)!} = D_{\overline{\gamma}_j,r}(l-1) \frac{(\overline{\lambda}+\overline{\gamma}_j)}{r+l}. \tag{1.24}$$

Therefore, $U_{m+l}(t)$ can be expressed more succinctly as

$$U_{m+l}(t) = \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \left[A^{(\overline{\gamma}_j)}_{\overline{\gamma}_j,k,r}(l) - A^{(-\overline{\lambda})}_{\overline{\gamma}_j,k,r}(l) C_{\overline{\gamma}_j,r}(l) \right]. \qquad (1.25)$$

In the implementation, for $i \leq m$, $U_i(t)$ is found via ACE. When the converged rate λ is encountered, values for $A^{(\zeta)}_{\overline{\gamma}_j,k,r}(0)$, $C_{\overline{\gamma}_j,r}(0)$ and $D_{\overline{\gamma}_j,r}(0)$ are calculated. The conversion [15] from $\overline{a}_{m,j,k}$ in Equation 1.17 to $\overline{b}_{m,j,k}$ in Equation 1.19 is given by

$$\begin{aligned} \overline{b}_{m,j,0} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,0} \\ \overline{b}_{m,j,k} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,k} + (k+1)\overline{a}_{m,j,k+1}, \; 0 < k < K_m(\overline{\gamma}_j) \\ \overline{b}_{m,j,K_m(\overline{\gamma}_j)} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,K_m(\overline{\gamma}_j)} \end{aligned} \qquad (1.26)$$

Then $U_{m+l}(t)$ for $l > 0$ is found from Equation 1.25 and by updating the A, C and D coefficients via Equations 1.22, 1.23 and 1.24. As can be seen, from this point onwards the computational complexity is linear, and thus modified ACE is an $O(N_a)$ algorithm assuming N_a is large compared to m. Further simplification is possible when $\lambda_i \neq \lambda_j, i \neq j$ for all $i, j \in \{0, \ldots, m\}$. Then $K_m(\overline{\gamma}_j) = 0$ for all $\overline{\gamma}_j$, and the sums over k and r disappear, thus requiring less calculation per epoch. See [14] for details.

The summation for $U_{m+l}(t)$ in Equation 1.25 is done by adding the terms in ascending order of their magnitude, for both positive and negative numbers, then summing these two intermediary results. This reduces cancellation error, and is reasonable to implement within modified ACE because it is of order $O(N_a)$. Since the coefficients A, C and D are independent of one another, all terms can be held in a single array that is updated each jump.

Uniformization The final method used to compute jump probabilities is standard uniformization itself. The variant of AU that uses this is called *layered uniformization* (LU) [14]. If we let $\lambda = \max\{\lambda_i\}, i = 0, \ldots, N_a$, then

$$U_n(t) = \sum_{k=N_{Bl}}^{N_{Br}} e^{-\lambda t} \frac{(\lambda t)^k}{k!} \underline{\pi}^B_k(n), \text{ for } n = 0, \ldots, N_a \qquad (1.27)$$

where

$$\underline{\pi}^B_k = \underline{\pi}^B_{k-1} P_B, \text{ with } \underline{\pi}^B_0(0) = 1 \qquad (1.28)$$

and

$$P_B = I + \frac{1}{\lambda} Q_B \qquad (1.29)$$

As shown in Section 2, the truncation bounds N_{Bl} and N_{Br} can be selected such that the resulting truncation error is smaller than an $\epsilon_B > 0$. The bound on the total error from truncation in LU is then given by $\epsilon + \epsilon_B$.

In the actual implementation, Q_B is found a step at a time based on the set of active states, which due to its special nature can be stored as an array rather than a sparse matrix. Thus there exists a P_{B_n} rather than a P_B, for each n such that

$$P_{B_n} = I + \frac{1}{\lambda_k} Q_{B_n} \text{ with } \lambda_k = \max\{\lambda_i\}, i = 0, \ldots, n. \tag{1.30}$$

So, each state (jump) probability is calculated separately, as each new adapted uniformization rate is found. As with modified ACE, reaching the converged rate λ at some epoch m is an important step. Then it is assumed that $\lambda_i = \lambda$ for $i = m, \ldots, N_a, \ldots$ and all the rest of the jump probabilities can be found immediately via standard uniformization. Since we take the converged rate to be λ, the maximum rate in the CTMC Y, the vector matrix multiplication in Equation 1.28 reduces to $\underline{\pi}_k^B(n) = \underline{\pi}_{k-1}^B(n-1)$ for $n > m$. This combined with using the Fox and Glynn algorithm results in a total order of complexity of $O(N_a\sqrt{N_{Br}})$ [13]. Notice that *a priori* knowledge of all adaptive uniformization rates would reduce the total computation necessary, but not the complexity.

Summary We have discussed three alternatives for calculation of the jump probabilities for adaptive uniformization. Implementations for all three options were described in the context of dynamic calculation of adaptive uniformization rates, that is, new rates are found per iteration as opposed to knowing them all in advance. The issue of reducing error in ACE and modified ACE was also addressed. Since the coefficients in these algorithms are unbounded and possibly negative, these two methods are potentially unstable. However, testing is required to effectively gauge their instability. On the other hand, uniformization is inherently stable because it uses only positive, bounded elements. In terms of complexity, modified ACE is best, followed by uniformization and then ACE. Experimental results will be given in Section 6 to compare these method's usefulness. Table 1 summarizes these three methods and the jump probability calculation for SU in terms of asymptotic complexity and potential stability.

Technique	*Complexity*	*Stability*
ACE	$O(N_a^2)$	Unstable
Modified ACE	$O(N_a)$	Unstable
Uniformization (LU)	$O(N_a\sqrt{N_{Br}})$	Stable
Poisson Probabilities (SU)	$O(\sqrt{N_{sr}})$	Stable

Table 1 Complexity and Stability of Methods

4 ALGORITHM IMPLEMENTATION

Given these methods to compute the jump probabilities, we can outline the algorithms used in the four uniformization methods considered. Inputs to the algorithms are t, ϵ, the infinitesimal generator matrix Q of the CTMC, and the initial state v. LU also requires a value for ϵ_B. Both the SU and AU algorithms are given in Figure 1, with parts common to both methods centered. Notice that adaptive uniformization varies slightly depending upon which method is chosen to calculate the jump probabilities.

In both SU and AU, Q is read into a sparse matrix structure, holding only the non-zero elements, with λ found during read-in. In SU, P is calculated immediately, along with the jump probabilities $U_{N_{sl}}(t), \ldots, U_{N_{sr}}(t)$. On the other hand, AU calculates $U_n(t)$ on a per epoch basis from Equations 1.13, 1.25 or 1.27, except for when the converged rate is reached in LU. In that case all remaining probabilities are calculated directly from Equation 1.27 as outlined in Section 3. Given the nature of the AU algorithm, it can be seen without much difficulty that Q could be generated as the transient solution was being found, making it applicable to unbounded state space problems. The implementation here depends on a pre-generated Q from the software package *UltraSAN* [1], and thus is limited to finite state space models.

Both algorithms use loops to control the summation of Equations 1.5 and 1.11. Each iteration calculates the next $\underline{\pi}_n$, then sums the product of it and epoch n's probability into $\underline{\pi}(t)$. In AU, finding $\underline{\pi}_n$ from $\underline{\pi}_{n-1}$ is dependent upon the adaptive uniformization rate λ_{n-1} and whether or not the rate has converged. If the rate is unconverged, $\underline{\pi}_n$ is calculated, and λ_n is found from the new set of active states. If, on the other hand, the converged rate λ has already been reached, $\underline{\pi}_n$ is found as in SU. In our implementation, as soon as the maximum, or converged, rate in the infinitesimal generator matrix is reached, Q is permanently converted to P so that no repetitive computation is done.

Standard Uniformization	*Adaptive Uniformization*
Read Q into sparse matrix structure	
$\lambda = \max(-q(i,i)) \mid i \in S$, v = initial state	
$P = I + Q/\lambda$	$\lambda_0 = -q(v,v)$
Calculate jump probabilities	Calculate $U_0(t)$ with ACE, Modified
$U_{N_{sl}}(t), \ldots, U_{N_{sr}}(t)$ given t,	ACE or LU using t, λ_0 and ϵ_B
λ and ϵ, with $U_i(t) = 0$	*sum* $= U_0(t)$; $n = 1$; *done* $= 0$;
for $i < N_{sl}$	*converged* $= 0$
$\underline{\pi}_0(v) = 1$, $\underline{\pi}_0(i) = 0$ for $i \neq v, i \in S$	
$\underline{\pi}(t) = \underline{\pi}_0 U_0(t)$	
For $n = 1$ to N_{sr} $\{$	While *done* $= 0$ $\{$
	If *converged* $= 0$
	$\underline{\pi}_n = \underline{\pi}_{n-1}(I + Q_{n-1}/\lambda_{n-1})$
	Find λ_n given $A_n = \{i \in S \mid \underline{\pi}_n(i) > 0\}$
	If $\lambda_n = \lambda$
	converged $= 1$; $P = I + Q/\lambda$
$\underline{\pi}_n = \underline{\pi}_{n-1} P$	LU: Find $U_n(t), \ldots, U_{N_a}(t)$
	Modified ACE: linear conversion
	Else If LU Calculate $U_n(t)$
	Else
	$\underline{\pi}_n = \underline{\pi}_{n-1} P$
	ACE or Modified ACE: Calculate $U_n(t)$
If $n \geq N_{sl}$,	$\underline{\pi}(t) = \underline{\pi}(t) + \underline{\pi}_n U_n(t)$
$\underline{\pi}(t) = \underline{\pi}(t) + \underline{\pi}_n U_n(t)$	
$\}$	*sum* $=$ *sum* $+ U_n(t)$
	If $(1.0 -$ *sum*$) < \epsilon$
	done $= 1$; $N_a = n$
	$n = n + 1$
	$\}$

Figure 1 SU and AU algorithms

The other main difference in the two methods is in how the main loop terminates. In SU, the end N_{sr} is determined *a priori* from the Fox and Glynn algorithm [4]. On the other hand, our implementation of AU finds the λ_i's on the fly, and thus jump probabilities are calculated only an epoch at a time. These are summed iteration by iteration and checked at every update to see if the ϵ requirement has been met. Thus N_a is not determined until the end of the computation.

5 PROGRAM INSTRUMENTATION

In order to fairly assess the accuracies and computational complexities of the different algorithms, we developed a *C++* class with which we instrumented our programs. First, it allows us to count floating point arithmetic operations by both operation type and program phase. And second, it tracks the bound on the machine round-off error for all floating point data. Thus this class is very useful for comparing numerical algorithms in terms of complexity and accuracy. Through its use of operator overloading [21], this extended double class easily instruments numerical *C* or *C++* code, requiring only changes to the variable type declarations and output statements.

Associated with each class element are pointers to counters for addition, subtraction, multiplication and division operations. These counters are double floating point types. Every floating point operation involving an extended double type increments the appropriate counter(s) pointed to by the variable on the left side of the assignment. By assigning different counters to variables in different segments of the program, computational complexity can be broken down into finer algorithmic detail.

The computed data $\tilde{x}$ and round-off error bound b are stored as double floating point types in each element of the class, such that the true x must lie in the range $\tilde{x} \pm b$. The bound b is not related to approximations due to the algorithm in use, but is only a bound on the machine round-off error based on the previous computations necessary to find $\tilde{x}$. Initial round-off error is determined by the machine precision and $\tilde{x}$'s absolute value as a default, although external values can be assigned. For a given arithmetic operation, b is propagated as follows

$$\text{For } \tilde{x}_r = \tilde{x}_1 + \tilde{x}_2,\ b_r = b_1 + b_2 \tag{1.31}$$

$$\text{For } \tilde{x}_r = \tilde{x}_1 - \tilde{x}_2,\ b_r = b_1 + b_2 \tag{1.32}$$

$$\text{For } \tilde{x}_r = \tilde{x}_1 \times \tilde{x}_2,\ b_r = b_1 \text{abs}(\tilde{x}_2) + b_2 \text{abs}(\tilde{x}_1) + b_1 b_2 \tag{1.33}$$

For $\tilde{x}_r = \tilde{x}_1/\tilde{x}_2$,

$$b_r = \begin{cases} \max\left\{\text{abs}\left(\frac{\tilde{x}_1}{\tilde{x}_2} - \frac{(\tilde{x}_1 - b_1)}{(\tilde{x}_2 + b_2)}\right), \text{abs}\left(\frac{(\tilde{x}_1 + b_1)}{(\tilde{x}_2 - b_2)} - \frac{\tilde{x}_1}{\tilde{x}_2}\right)\right\} & \text{if } \tilde{x}_1 \times \tilde{x}_2 \geq 0 \\ \max\left\{\text{abs}\left(\frac{\tilde{x}_1}{\tilde{x}_2} - \frac{(\tilde{x}_1 + b_1)}{(\tilde{x}_2 + b_2)}\right), \text{abs}\left(\frac{(\tilde{x}_1 - b_1)}{(\tilde{x}_2 - b_2)} - \frac{\tilde{x}_1}{\tilde{x}_2}\right)\right\} & \text{if } \tilde{x}_1 \times \tilde{x}_2 < 0 \end{cases} \tag{1.34}$$

where abs($\tilde{x}$) is the absolute value of $\tilde{x}$ and max{} returns the maximum of its arguments. Note that the last term in Equation 1.33 is not implemented because it is negligible for small round-off errors. This method provides an approximate upper bound [20] on the error accumulated due to machine round-off. This upper bound is approximate because the bound computation is itself susceptible to round-off error. In the next section we will give results based upon this class.

6 RESULTS

In order to study the viability of AU, and to illustrate the code instrumentation, we will use an extended machine-repairman (EMR) model with delayed repairs, similar to ones given in [2, 22]. In this model there are K components, all with the same failure rate, ρ. There are two classes of failure, *hard* and *soft*, where the probability of a failure being soft is c and that of a hard failure $1 - c$. c is known as the *coverage factor*. Repair begins when r components have failed, with repairs taking place at a rate of μ for a hard failure and ν for a soft failure, with $\nu > \mu$ typically. The failures and repairs are all negative exponentially distributed in time with ρ, μ and ν as parameters. Each component has independent repair capability, and repair continues until all units are operational. Correct service is provided whenever at least one component is functional. Usually μ, $\nu \gg \rho$, and thus there are a number of epochs with low adaptive uniformization rates followed by higher rates and the converged rate.

For our studies, we have chosen two sets of models with a range of K values and constant r, ρ, μ, ν and c. Notice that c has no effect on the state space size, and that the adaptive uniformization rates are a function of K, r, ρ and $\max\{\mu, \nu\}$. Also, the converged rate $\lambda = (K-1)\max\{\mu, \nu\} + \rho$. For the first set $K = 20$ or 50, $r = 10$, $\rho = 1$, $\nu = 1000$, $\mu = 800$ and $c = 0.5$. Thus while μ and c may effect the reliability, they do not effect the computation time in the discussed algorithms. The second set consists of larger models, where $K = 200$, 250 or 300, $r = 100$, $\rho = 1$, $\nu = 100$, $\mu = 80$ and $c = 0.5$. For all LU results, $\epsilon_B \leq 10^{-14}$, whereas parameter ϵ is varied along with t as part of the experiment. We used mission times t on the order of $1/\rho$, the mean time of a component failure, which is reasonable for highly reliable applications [14]. A typical measure of interest in this model is then system reliability, or the probability of at least one unit functioning.

Several aspects of our results will be discussed. First, the accuracy of the different methods will be compared in terms of their resulting bounds on the round-

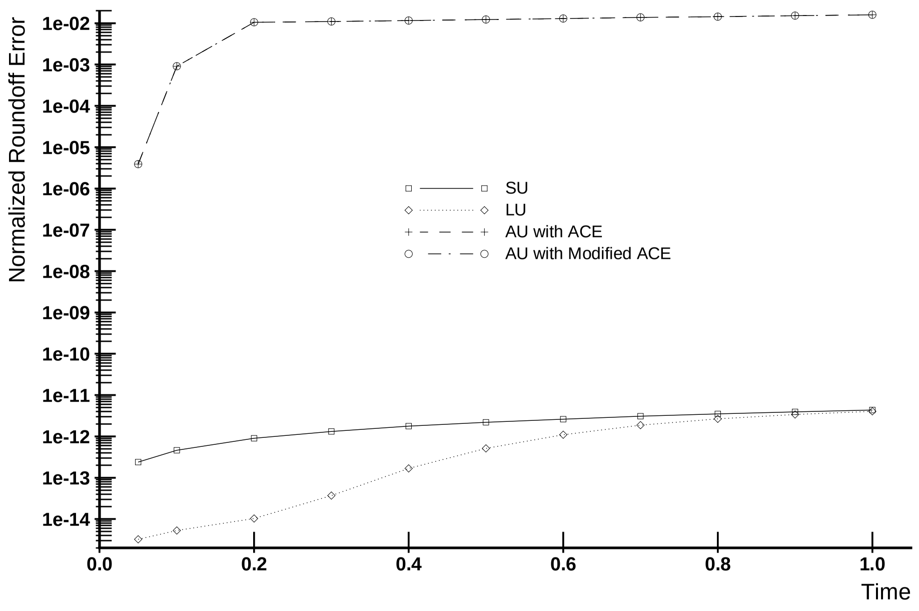

Figure 2 Reliability Roundoff Error Bounds: $K = 20$, $r = 10$, $\epsilon \leq 10^{-4}$

off error. Second, an illustration of operation counting in the instrumented code will be given. Finally, the computational complexity of the algorithms are compared. Two methods of comparison are used, computation time (CPU usage) and number of floating point operations. Since all AU algorithms have a higher order of complexity in computing jump probabilities than SU with respect to time, the results exhibit *crossover points* t', where AU is less computationally complex for $t < t'$ and SU performs better for $t > t'$, whenever $N_a < N_{sr}$. We will designate t_T to be the crossover point based on computation time, while the floating point operation crossover time is labeled t_F. The two measurements will show slight differences because some overhead is not accounted for by operation counts.

Both SU and LU are numerically stable owing to their use of probability matrices with positive, bounded elements, and the Fox and Glynn method to calculate Poisson probabilities, while ACE and modified ACE are not guaranteed to be stable. We will first look at the limitations of these algorithms. The round-off error results are from system reliability, and are normalized such that the error shown is relative to this reliability. As can be seen in Figure 2, the accuracy of both ACE and modified ACE is limited. The tests from Figure 2 were run with the requirement $\epsilon < 10^{-4}$, and yet the bounds on the resulting round-off error were often orders of magnitude larger than this for a large range of t. Other models with different K values, and smaller ϵ requirements, showed worse performance, even to the point where viable solution was unattainable. Due to the limited capabilities of the ACE and modified ACE algorithms, they are probably not useful for implementations of adaptive uniformization. Therefore the rest of our results focus on the LU variant of AU.

On the other hand, SU and LU show good results in the area of numerical stability, with similar results found over a wide range of models and ϵ values. LU shows better performance for small t simply because fewer calculations were necessary to achieve the result. The truncation error in SU can be made arbitrarily close to 0 through the use of Fox and Glynn's algorithm for computing Poisson probabilities. However, making ϵ less than the machine's epsilon, i.e., the smallest number x such that $1.0 + x \neq 1.0$, has little value. LU's truncation error is limited by this machine epsilon, since a running total sum of the jump probabilities is used to detect when to truncate. A specified ϵ less than the machine epsilon would not allow the infinite sum to truncate.

We now show results to illustrate use of the instrumentation method to count operations. Figure 3 shows a breakdown by operation type for a 50 component EMR model. The operation types are important because their individual computational complexities vary from machine to machine, and thus comparisons may want to take this into account. In this example it can be seen that additions and multiplications dominate the divisions and subtractions in both the SU and LU algorithms. However, the operation distributions are not the same, as LU requires a larger percentage of subtractions and divisions for solution. The operation crossover point in computational complexity can be seen to lie somewhere between $t = .6$ and $t = .8$.

In Figure 4 the counts are divided between the different aspects of the algorithm that contribute to the computational complexity:

- INIT: Operations done during the read-in of Q.
- JPR: Computation of jump probabilities $U_n(t)$.
- QP: Calculation of P and/or P_n.
- MVM: Matrix vector multiplication $\underline{\pi}_{n-1}P$ and/or $\underline{\pi}_{n-1}P_{n-1}$.
- SOL: $\underline{\pi}(t) + \underline{\pi}_n U_n(t)$ calculation.

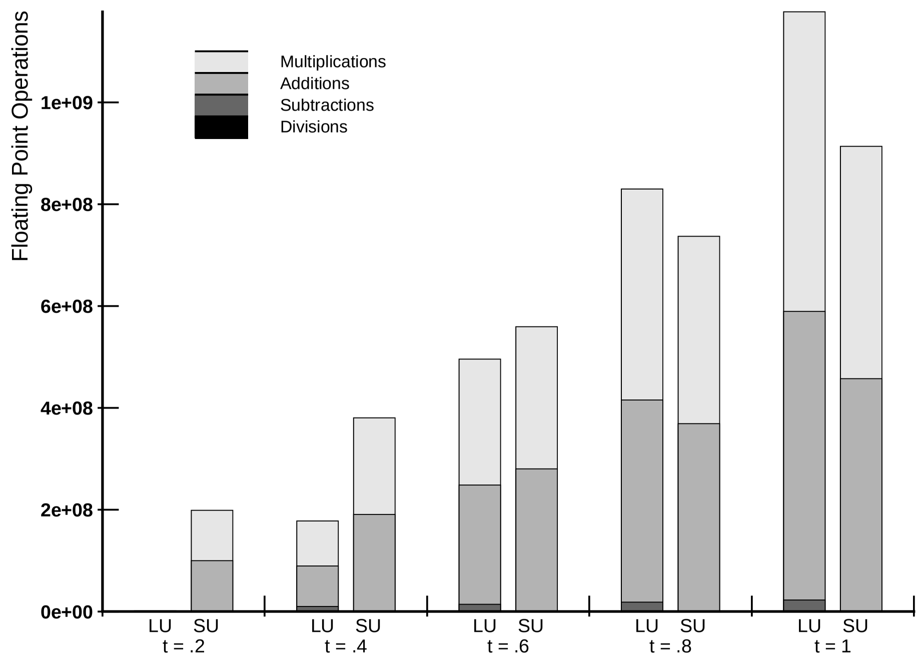

Figure 3 $K = 50$, $r = 10$, 2095 states, $\epsilon \leq 10^{-12}$

The time $t = .6$ was selected because total computational complexity was almost equal, allowing fair comparison. The histogram clearly shows the tradeoff between the two methods. LU trades increased computational complexity in calculating the jump probabilities (JPR) for a reduction in the number of matrix vector multiplications (MVM), as compared to SU. The latter follows from the fact that LU truncates earlier because the adaptive uniformization rates allow it to make probabilistically larger jumps. Note that since the scale is logarithmic, the difference in MVM complexity is actually greater than the difference in JPR. LU also shows increased activity in QP since P_n is calculated multiple times and in SOL because LU does not left truncate.

The next two figures demonstrate the computational advantages of LU, as well as illustrate the two kinds of crossover points. Larger state space-sized models with $K = 200, 250, 300$ and $r = 100$ were selected to show AU and SU's ability to solve very large problems. Figure 5 depicts the computational complexity in terms of the total number of floating point operations. Within these results,

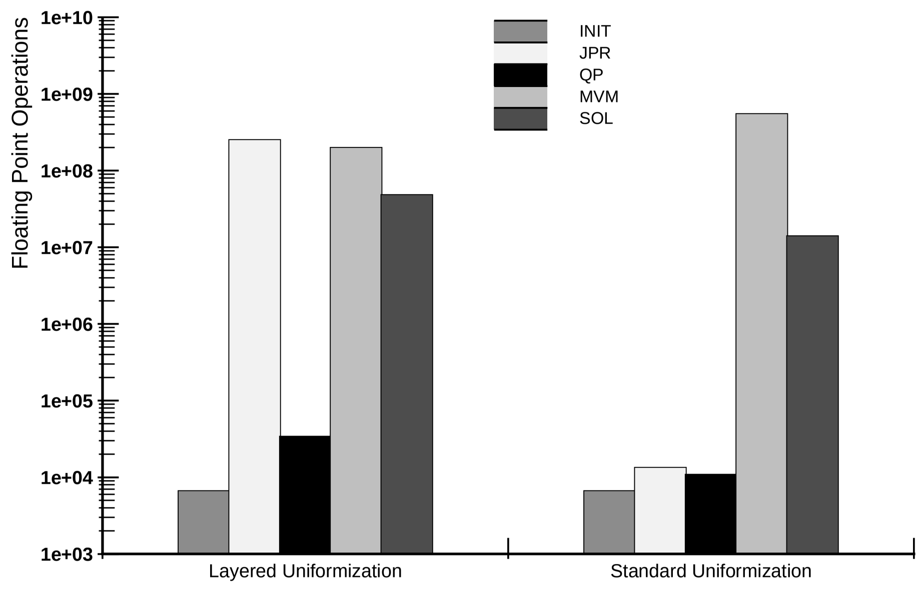

Figure 4 $K = 50$, $r = 10$, 2095 states, $\epsilon \leq 10^{-12}$, $t = .6$

all operation types are given equal value. For a given machine these operations could easily be weighted to reflect their individual computational intensities. It can be seen that t_F, the operation crossover point, occurs at approximately $t = 1.9$. For $t < .5$, the converged rate has not been reached when the truncation point occurs, and thus it can be seen that savings of about 2 orders of magnitude are reaped for this particular model. The large jump in complexity at $t \approx .5$ occurs when the converged rate is reached, and no further savings are possible.

A graph of computation time on a Sparcstation 10/30 versus time t is given in Figure 6. t_T occurs at slightly less than $t = 1$. This is less than t_F because of the extra control overhead necessary to implement LU. Better optimized software will move t_T closer to t_F. Also note that the savings for $t < .5$ are less because operation counts do not reflect computational time spent on matrix read-in, etc. The remaining results will be based upon instruction counts, since these are reliable and repeatable. The irregular nature of Figure 6 shows clearly that CPU usage is not the most dependable measure since it can be unpredictably influenced by concurrent execution of the operating system and other application programs.

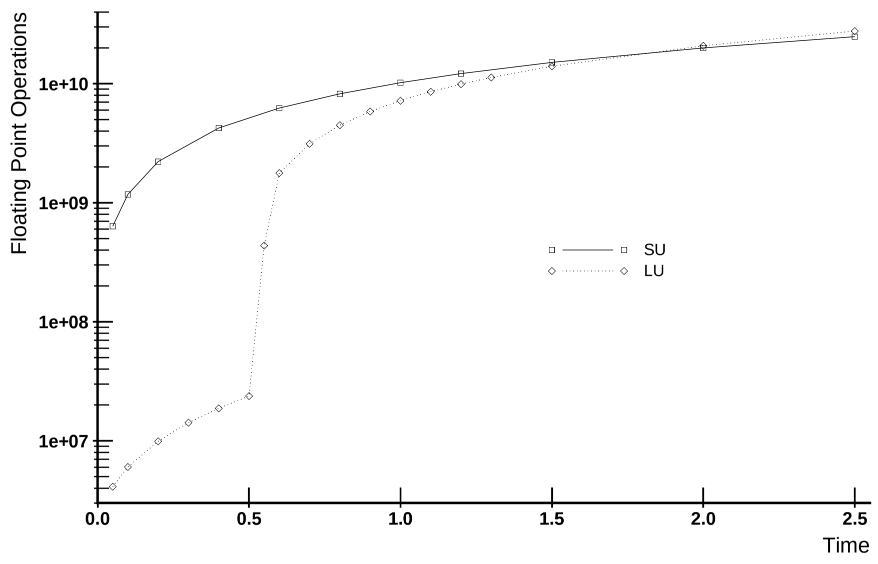

Figure 5 Operations vs. Time: $K = 250$, $r = 100$, 42700 states, $\epsilon \leq 10^{-8}$

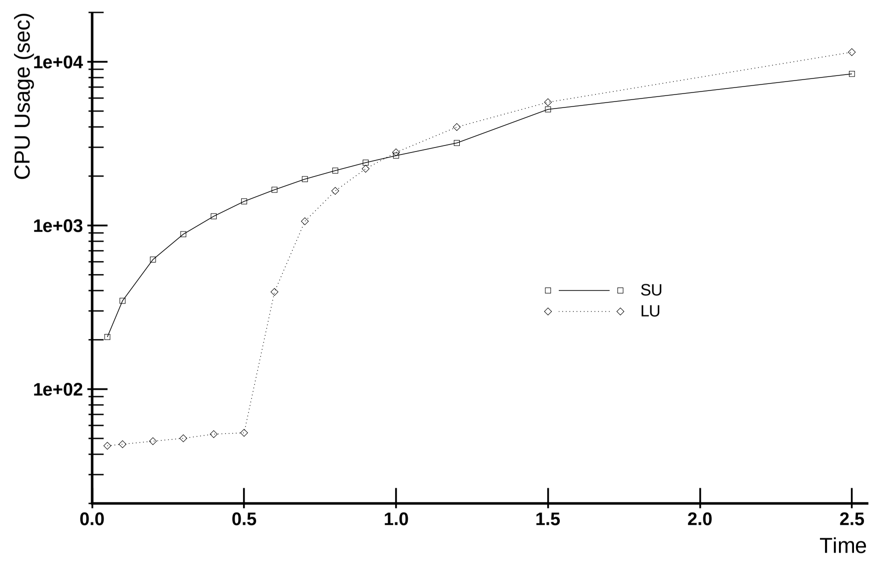

Figure 6 CPU Usage vs. Time: $K = 250$, $r = 100$, 42700 states, $\epsilon \leq 10^{-8}$

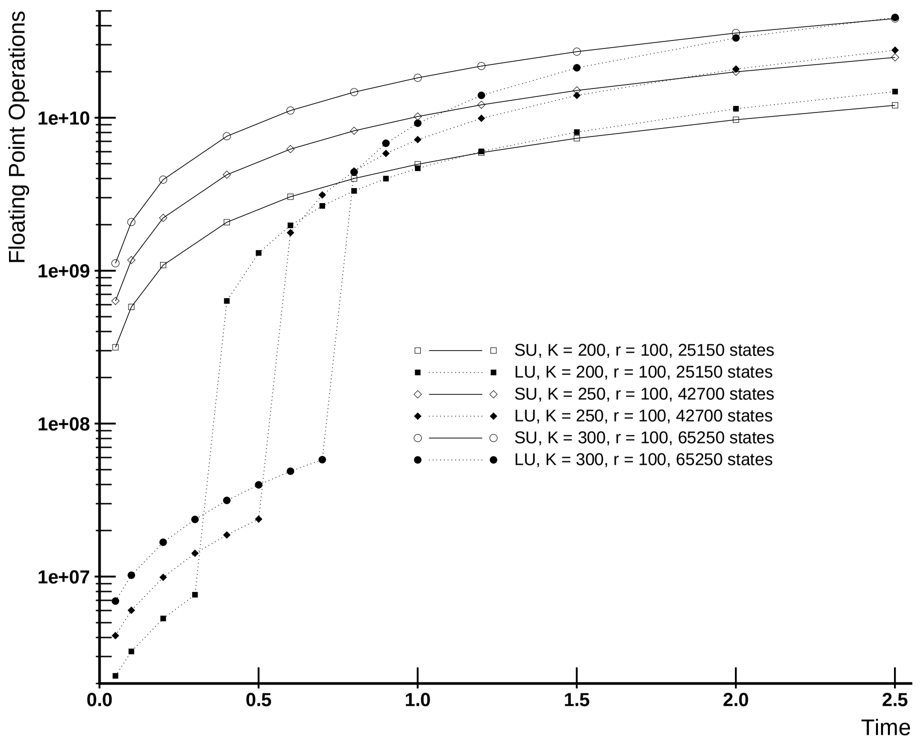

Figure 7 Computational Complexity vs. Time, $\epsilon \leq 10^{-8}$

Figure 7 shows the computational complexity versus the time parameter for 3 different EMR models of varying state space size. This graph shows how increased state space size increases t_F because a reduction in the number of vector matrix multiplications becomes a more important factor. t_F rises from less than 1.2, to approximately 1.8, and finally to just under 2.5 for the 65250-state model. Note that this effect on t_F is reduced to some extent since as the state space size scales, so does its maximum, or converged rate, which has a greater effect on LU jump probability calculation than for SU.

Figure 8 illustrates how the requested truncation error limit effects the computation. Obviously, SU is insensitive to accuracy demands. This is because the Poisson distribution approaches zero rapidly in either tail, making the amount of truncation differ little with changes in ϵ. The distribution of the jump probabilities decreases much more slowly from its maximum in LU, and therefore

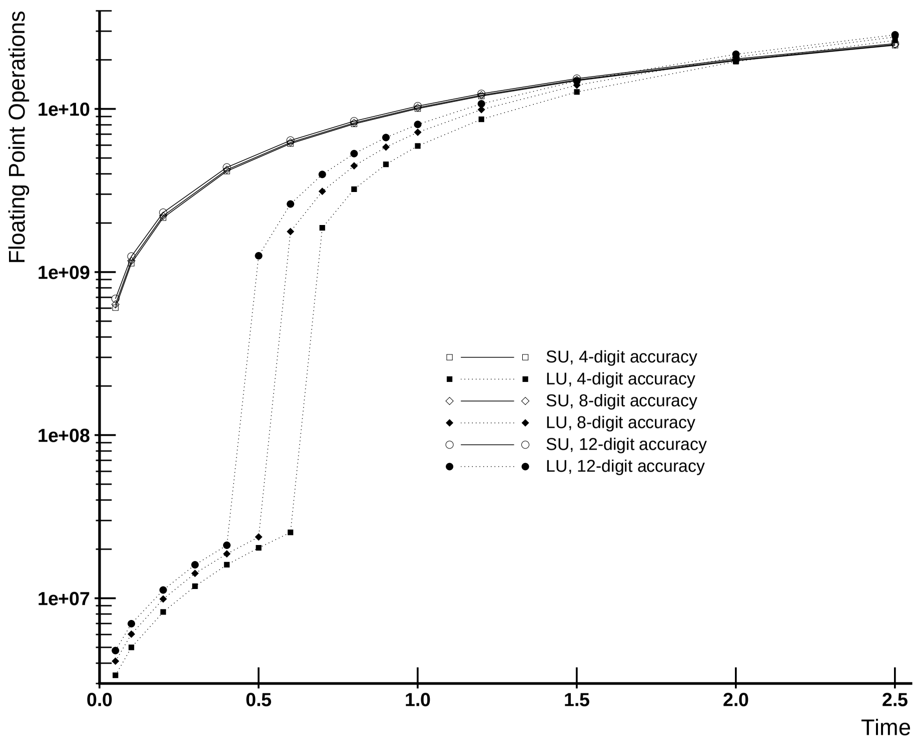

Figure 8 Computational Complexity vs. Time: $K = 250$, $r = 100$, $\epsilon \leq 10^{-4}, 10^{-8}, 10^{-12}$

changes in ϵ significantly effect the truncation point N_a and thus the computational effort needed for solution. Therefore we see that for $\epsilon \leq 10^{-4}$, $t_F \approx 2.1$, while for $\epsilon \leq 10^{-8}$, t_F is about 1.9 and for $\epsilon \leq 10^{-12}$, $t_F \approx 1.7$.

7 CONCLUSION

Many methods have been proposed to compute the transient state occupancy probabilities in continuous-time Markov chains, but no method has proven best for all models and time points of interest. Uniformization is popular, but results in extremely long run times to achieve acceptable accuracy when models are stiff. Adaptive uniformization has been proposed to alleviate these problems for some models. In this paper, we compare the efficiency and stability of three implementations of AU to that of standard uniformization. The efficiency and stability is compared by instrumenting the code using a *C++* class. This class we developed tracks a bound on the round-off error incurred during the computation, as well as its computational complexity, measured in number of floating point operations. Operation counts can be refined by operation types and the algorithm phase in which they occur. The class is general, and can instrument other numerical algorithms written in *C* or *C++* code with minimal effort.

The three adaptive uniformization implementations studied differ in the method by which they compute the jump probabilities. Of the three methods, ACE has the highest order of complexity, but is theoretically applicable to any jump process resulting from AU. modified ACE has the least complexity, but can only be used when and if a converged rate exists. Standard uniformization is the third method, hence the name layered uniformization for this AU variant. LU's order of complexity falls between the two previous methods, and because of its probabilistic nature, is numerically stable. Comparing the AU alternatives with SU via our instrumented code, we found the methods based on ACE and modified ACE not very useful because of their high round-off errors. Although not investigated, it is possible that these methods could be improved, for instance, by aggregating close-valued poles into a single pole value, avoiding some cancellation error problems. Computation of the jump probabilities in AU via uniformization resulted in small round-off errors which were, in fact, smaller than those for SU for many time points of interest, since the number of operations performed was smaller.

Comparisons of computational efficiency of the methods were limited to LU and SU, due to the severe round-off errors exhibited by ACE and modified ACE. With regard to these methods, we found for short mission times in extended machine-repairman models, AU was orders of magnitude better than SU. For large time points, however, AU is less attractive than SU, due to its higher overall complexity. The two algorithms thus exhibit a "crossover point" where, on one side, AU is better than SU, and on the other, SU better than AU. The time crossover point t_T was found to be smaller than the operation crossover point t_F, but better optimized software should reduce this difference. Studies of varying size models showed that AU's performance improves significantly with larger state spaces, as the reduction in the number of matrix vector multiplications becomes more important. Finally, we showed that the computational cost of AU is more sensitive to the desired accuracy than is SU. Thus choosing a lower accuracy with AU will reduce its run time more significantly than in SU. Furthermore, like SU, AU with uniformization to compute the jump probabilities is stable and exhibits very low round-off errors for the models studied.

These results suggest applications that could benefit directly from adaptive uniformization and, due to the crossover point, suggest that AU could be profitably combined with other methods. In particular, a solver that estimates the crossover point, then selects the best method given the mission time may be built. Furthermore, multiple methods might be combined, in some way, to achieve more efficient solutions. Further study is necessary to determine if and when these hybrid approaches would be advantageous.

Acknowledgements

We thank Dr. A.P.A. van Moorsel and the anonymous reviewers for their many helpful suggestions.

REFERENCES

[1] J.A. Couvillion, R. Freire, R. Johnson, W.D. Obal II, M.A. Qureshi, R. Rai, W.H. Sanders and J. Tvedt, "Performability Modeling with UltraSAN," *IEEE Software*, **8**, (Sept. 1991), pp. 69-80.

[2] J. Dunkel and H. Stahl, "On the Transient Analysis of Stiff Markov Chains," *Proc. Third Working Conference on Dependable Computing for Critical Apllications*, Modello, Italy, Sept. 1992.

[3] W. Feller, *An Introduction to Probability Theory and Its Applications, Volume I and II*, New York, 1966.

[4] B.L. Fox and P.W. Glynn, "Computing Poisson Probabilities," *Comm. ACM* **31**, pp. 440-445, 1988.

[5] R. Geist and K.S. Trivedi, "Ultra-high Reliability Prediction for Fault-Tolerant Computer Systems," *IEEE Trans. on Comp.*, C-32(12), pp. 1118-1127, December 1983.

[6] A. Goyal, S. Lavenberg and K.S. Trivedi, "Probabilistic Modeling of Computer System Availability," *Annals of Operation Research*, **8**, pp. 285-306, 1987.

[7] W.K. Grassman, “Transient Solutions in Markovian queueing systems,” *Computers & Operations Research* **4**, pp. 47-53, 1977.

[8] W.K. Grassman, “Finding Transient Solutions in Markovian Event Systems through Randomization,” in *Numerical Solution of Markov Chains*, W.J. Stewart (Ed.), Marcel Dekker, New York, 1991.

[9] A. Jensen, “Markoff chains as an Aid in the Study of Markoff Processes,” *Skand. Aktuarietidskrift.*, **36**, pp. 87-91, 1953.

[10] C. Lindemann, Private Communication, November 1993.

[11] C. Lindemann, M. Malhotra and K.S. Trivedi, “Numerical Methods for Reliability Evaluation of Closed Fault-tolerant Systems,” *Technical Report*, Duke University, 1992.

[12] M. Malhotra, “A Unified Approach for Transient Analysis of Stiff and Non-Stiff Markov Models,” Technical Report DUKE-CCSR-92-001, Center for Computer Systems Research, Duke University, 1992.

[13] A.P.A. van Moorsel, “Performability Evaluation Concepts and Techniques,” Ph.D. Thesis, University of Twente, 1993.

[14] A.P.A van Moorsel and W.H. Sanders, “Adaptive Uniformization,” *Stochastic Models*, **10:3**, 1994.

[15] A.P.A. van Moorsel and W.H. Sanders, “Adaptive Uniformization: Technical Details,” *PMRL Technical Report 93-4*, University of Arizona, 1992.

[16] R.A. Marie, A.L. Reibman and K.S. Trivedi, “Transient Analysis of Acyclic Markov Chains,” *Perf. Eval.* **7**, pp. 175-194, 1987.

[17] A. Reibman, R. Smith and K.S. Trivedi, “Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches,” *European Journal of Operational Research*, **40**, pp. 257-267, 1989.

[18] A. Reibman and K.S. Trivedi, “Numerical Transient Analysis of Markov Models,” *Comput. Opns Res.* **15**, pp. 19-36, 1988.

[19] N.C. Severo, “A Recursion Theorem on Solving Differential- Difference Equations and Applications to some Stochastic Processes,” *J. Appl. Prob.* **6**, pp. 673-681, 1969.

[20] P.H. Sterbenz, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, 1974.

[21] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1987.

[22] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, 1982.