

Normalization via Efficient Search

Jorge R. Ramos and Vernon Rego^{1 2}
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907
U.S.A.

Abstract

In many multidimensional systems, the equilibrium distribution of state probabilities and other performance measures hinges on the explicit numerical determination of a normalization constant. When the normalization constant is a sum of simple functions defined on constrained space of lattice-points, the normalization constant can be obtained in a time that is directly proportional to the size of the space. We present an efficient algorithm that is a tree-based variant of depth-first search which minimizes the usage of a stack. Using simple complexity arguments as well as experimental results, we show that the algorithm offers run-times that are a many-fold improvement over previously proposed multidimensional recurrences.

1 Introduction

Normalization constants are key to the solution of a host of problems in the design of systems based on dynamic entities (packets, jobs) which compete for resources (buffers, network links, processors). In the performance analyses of computer and communication systems, these critical constants appear in expressions for state probabilities in closed queueing networks. They are critical because the understanding of system behaviour hinges on state probabilities which, in turn, cannot be evaluated without knowledge of the appropriate normalization constants [3, 4]. Indeed, we motivate the basic problem using an example involving packet-loss probabilities in broadband networks [5], though the solution technique may be exploited in more general situations. Our proposed solution makes no assumptions other than that the contribution of each lattice-point in the normalizing space and the normalizing space both be well-defined.

Broadly speaking, there are two classes of methods that have been developed for the computation of normalization constants, more generally known as partition functions [7, 8]. When dimensionality is small, the standard method is to exploit recursion [9], such as in the evaluation of convolutions [10], and when dimensionality is large, useful methods based on asymptotic expansions of integral representations have been very successful [11, 12]. A very different, and also highly successful approach, is the numerical inversion of the generating function of the partition function [14]. This technique has its roots in

²This work was supported by DoD DAAG55-98-1-0246 and PRF-6903235.

²E-mail: jrramos@cs.purdue.edu, rego@cs.purdue.edu.

the early transform inversion work of Dubner and Abate [15]. An excellent survey and example of generating function inversion can be found in [13].

The approach we take is direct and algorithmic, without the complexity of numerical inversion or the exploitation of special structure in recursive schemes. We present an algorithm that views the space as a tree of lattice points and visits each lattice point in turn, without repetition. The advantage is simplicity and robustness. The technique is a significant improvement over certain other methods, such as the multidimensional recurrence proposed in [16], and we have found run-times to be small for high-dimensional problems.

2 Normalization Constants

Following the conventions established in [5, 6] and consider a broadband network with m links, with link j having a capacity of C_j resource units, for $1 \leq j \leq m$. There are n distinct classes of calls supported by the network, and a call of class k is characterized by an offered load ρ_k and a bandwidth requirement of $r_{j,k}$ on link j , where the latter is zero when link j is not used by a class- k call. Let the vector $\mathbf{r}_j = (r_{j,1}, \dots, r_{j,n})$ denote the bandwidth requirements of the distinct classes on link j , $1 \leq j \leq m$, and let the state of the system be represented by the vector $\mathbf{X} = (X_1, \dots, X_n)$, where component X_k captures the number of class- k calls in progress.

The system can be seen to move between states in the set $\mathbf{S} = \{\mathbf{x} \mid \mathbf{x} \geq 0\}$, where each component x_i of the state vector \mathbf{x} belongs to the set of nonnegative integers. If a call is always accepted whenever there is capacity to handle the call, and blocked calls are always cleared, then under appropriate assumptions on the nature of the call-arrival process, the probability that the system is in a given state \mathbf{x} is obtained as

$$P\{\mathbf{X} = \mathbf{x}\} = \prod_{k=1}^n e^{-\rho_k} \frac{\rho_k^{x_k}}{x_k!} \quad (1)$$

provided that link-capacities are infinite. Let $Z_{j,k} = r_{j,k}X_k$ be a random variable representing the occupancy level of link j , due to demands exercised by calls of type k , so that

$$P\{Z_{j,k} = z\} = e^{-\rho_k} \frac{\rho_k^{z/r_{j,k}}}{(z/r_{j,k})!} \quad (2)$$

whenever $r_{j,k} > 0$ and $z = ir_{j,k}$ for some nonnegative integer i , and the probability is defined as zero otherwise. When link capacities are finite, complications arise because the state-space becomes restricted. That is, the set of permissible states shrinks to $\mathbf{R} = \{\mathbf{x} \in \mathbf{S} \mid \forall j : \mathbf{r}_j \cdot \mathbf{x} \leq C_j\}$ where $\mathbf{r}_j \cdot \mathbf{x} = \sum_k r_{j,k}x_k$ for $1 \leq j \leq m$. The steady-state distribution must now be normalized by conditioning on the probability mass of

the constrained space \mathbf{R} , so that the probability that the constrained system is in state \mathbf{x} at equilibrium is given by

$$\pi(\mathbf{x}) = \frac{1}{G} \prod_{k=1}^n e^{-\rho_k} \frac{\rho_k^{x_k}}{x_k!} = \frac{1}{G} \prod_{k=1}^n f(x_k, \rho_k) \quad (3)$$

where $f(x_k, \rho_k) = e^{-\rho_k} \rho_k^{x_k} / x_k!$, and the value of the normalization constant

$$G = \sum_{\mathbf{x} \in \mathbf{R}} \prod_{k=1}^n e^{-\rho_k} \frac{\rho_k^{x_k}}{x_k!} \quad (4)$$

is what we are interested in explicitly determining. Once the value of G is known, a number of quantities of interest can be computed — such as, for example, queue-size distributions or blocking probabilities for the system. For example, the set of blocking-states for a class- k call is given by those states in \mathbf{R} that violate capacity, i.e., $\mathbf{S}_k = \{\mathbf{x} \in \mathbf{R} \mid \exists j : \mathbf{r}_j \cdot (\mathbf{x} + \mathbf{u}_k) > C_j\}$ where \mathbf{u}_k is an n -vector with a 1 in the k -th component and zeros elsewhere. If Y_k is a Bernoulli random variable which takes on the value 1 when a class- k call is blocked and takes on value 0 otherwise, the blocking probability of a class- k call is given by

$$P(Y_k = 1) = \sum_{\mathbf{x} \in \mathbf{S}_k} \pi(\mathbf{x}) = \frac{P\{\mathbf{X} \in \mathbf{S}_k\}}{P\{\mathbf{X} \in \mathbf{R}\}} \quad (5)$$

3 Depth-First Tree Search

We modify the standard depth-first search algorithm [1, 2] to make it non-recursive, exploit optimizations to minimize the number of graph-edges traversed, and ultimately reduce stack usage a minimum. The optimization is achieved by exploiting the special property of the set \mathbf{R} . Instead of using a generalized graph to represent the neighborhoods of lattice-points in \mathbf{R} , we use a tree. We construct a tree such that each lattice-point in the tree is visited just once. The origin is taken to be both the root of the tree and the starting point of the traversal; movement is always in a positive direction, and one lattice-point is consumed at each step. All edges in the tree will be the unit vectors of the given dimension n , and vertices of the tree will be the lattice-points that are connected by edges. Since only positive values are considered for the unit vectors, all edges are presumed to be directed edges and there are no back-edges. Finally, the tree is constructed in such a way that each point is reachable only via a single edge. That is, each point can be approached from only one direction. This minimizes the number of edges traversed while searching for points not yet visited, consequently eliminating repeated visits to lattice-points.

The tree T representing lattice-points in \mathbf{R} is constructed as follows. Initially, T is defined to be empty. In the very first step the origin in \mathbf{R} is added to T and taken to

be the root of the tree. Next, starting from a lattice-point (vertex) v already in the tree (e.g., the origin) one edge $E (v \rightarrow v')$ and one vertex v' are added to the tree T , if the traversal allows a move from lattice-point v to lattice-point v' in \mathbf{R} . Repeating this procedure recursively for all lattice-points and each permissible direction, the tree T is eventually completed. In essence, T is a tree because each vertex acts as the root of a new sub-tree. Clearly, the order in which dimensions are traversed, and the allowable directions, ultimately determine how the tree is built — a procedure that depends on both the algorithm as well as \mathbf{R} . The permissible directions are made to depend on the vertices, so that the result of the traversal is a tree.

The concept of degree is important for understanding the algorithm, and so we present the following definitions. The **Degree of a vertex** is the number of edges emanating from a vertex. Recall that we use directed edges. By definition, only one edge reaches into a vertex, though many edges reach out of it. The **Degree of an edge** is the direction of the edge, and the **Degree of a tree** is the degree of the root of that tree. If a branch is a collection of consecutive lattice-points possessing the same degree, the degree of a branch is the degree of its lattice-points.

The Algorithm. When the algorithm runs, it implicitly builds tree T . It utilizes the following rules [17] to assign each lattice-point a degree when it visits the point. First, the **Degree of root** = ndim . Second, the **Degree of a vertex** is the direction of the edge reaching the vertex. Observe that this edge is unique. This also defines the number of edges emanating from the vertex. Finally, since vertex edges emanate in up to K distinct directions, $1, 2, \dots, K$, where K is the degree of the starting vertex, the degree of an edge is simply the direction of the edge.

During traversal, when the algorithm encounters a vertex of degree K , $K > 1$, it proceeds to traverse the edges emanating from the vertex in the natural order $1, 2, \dots, K$. While the degree of a vertex is the number of edges emanating from the vertex, it also the label of the direction that must be traversed from that vertex. A stack is used to guarantee that traversal occurs in this specific order. Upon finding lattice-points of degree greater than one, along traversed edges, the algorithm invokes the above procedure recursively, storing points that have already been encountered in the stack alongside the additional information detailed below. The algorithm repeatedly compares the degree of a point to the direction being traversed, stopping only when the K -th direction has finally been traversed. Direction 1, a special case, is traversed without pushing any point onto the stack. By following these rules for traversal, utilization of the stack is kept to a minimum.

The stack, which enables the traversal to be done nonrecursively, store records containing the the coordinates of a lattice-point, the direction (edge) to be traversed, after a lattice-point is popped off the stack, and the degree of the lattice-point. The stack stores

a lattice-point with multiple edges (i.e., degree greater than one) and the next direction to be traversed by the algorithm when this point is popped off the stack. Whenever a boundary of \mathbf{R} is met, a lattice-point is popped off the stack and the next edge of that lattice-point is considered for traversal. This procedure is repeated until all edges have been traversed. The degree of the lattice-point is stored on the stack to enable simple detection of a termination condition. The root is a special case that is handled at the very start of the algorithm — by pushing it onto the stack, if the tree possesses a degree greater than one.

Proof of Correctness. For $\text{ndim} = 1$, it is clear that the entire constrained space is traversed, because the algorithm systematically moves from the origin, along direction 1, up until the boundary of the graph. The result is a tree of degree 1.

Now assume that the algorithm covers all of the lattice-points given by a tree of degree D . If we build a tree of degree $(D + 1)$, the algorithm proceeds by constructing a branch of degree $(D + 1)$ in which all vertices are roots of trees in dimension D . The branch of degree $(D + 1)$ starts at the origin and goes in direction $(D + 1)$, up until the boundary for dimension $(D + 1)$. This effectively enables all the lattice-points in that direction to be covered. Since those points are the roots of trees of degree D , and they cover all points of dimension D , all points in the set \mathbf{R} are covered.

Upon effectively building a tree of degree $(D + 1)$, the algorithm adds one branch that ties together trees of degree D . In turn, each tree of degree D is constructed by tying together trees of degree $(D - 1)$, until the termination condition, in which trees of degree 2 are constructed by a branch that ties together trees of degree one.

Complexity. Let \mathbf{R}^* be the set of lattice-points $\mathbf{x} \in \mathbf{R}$ for which equality is attained for at least one constraint in the system given in (8), and define the boundary $b(\mathbf{R})$ to be the set of lattice-points that are not in \mathbf{R} but are neighbours of points in \mathbf{R}^* . Let d be the number of lattice-points in $\mathbf{R} \cup b(\mathbf{R})$. During the traversal of any set \mathbf{R} , the algorithm must touch points in $b(\mathbf{R})$ while querying lattice-points for membership in \mathbf{R} . At the boundary it is able to verify that it has indeed arrived at a limit in some direction. Considering stack processing, each lattice-point is visited a number of times that is equal to its vertex degree, and is also equal to the number of edges emanating from that vertex. This includes edges reaching outside the constrained set for boundary points. Thus, the run-time $g(d) \in O(\sum v_i E(v_i)) = O(E)$, from where, by associating each lattice-point with the edge used to reach that point, we can conclude that $d = E - 1$. This is because each lattice-point is approached only once, so that every point has one approaching edge, except for the origin which is the start point and has no approaching edge.

Thus, the run-time complexity is given by $g(d) = d$, implying that the execution time is directly proportional to the number of points in the constrained set \mathbf{R} . Indeed, this

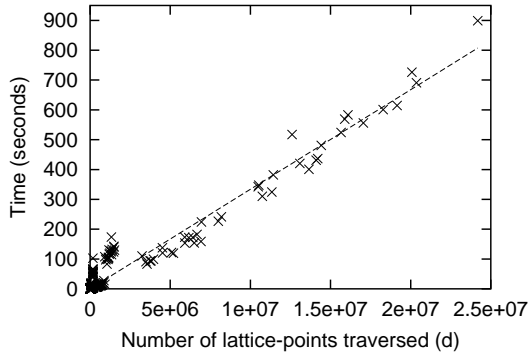


Figure 1: Run-time vs. average set-size \bar{d}

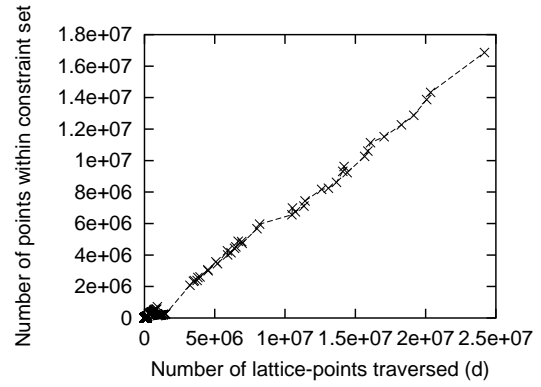


Figure 2: Correlation of d with $(d - |b(\mathbf{R})|)$

appears to be a marked improvement over the generating function-based recursion proposed in [16] which offers a run-time complexity of $O(n \prod_{j=1}^m C_j)$ and a space complexity of $O(\prod_{j=1}^m C_j)$. In contrast, observe that the stack-space requirement of the tree-search algorithm is bounded from above by $n - 1$.

4 Experimental Results

While the complexity of the algorithm suggests that the procedure may not be feasible when the size of \mathbf{R} is prohibitively large, the procedure is feasible when (a) d is of reasonable size, (b) the number of constraints m is moderate or large, relative to the number of dimensions n , and (c) the parameters (i.e., ρ_k) along each dimension are such that the tree-building procedure enables run-time pruning, so that traversals along certain directions can be terminated when lattice-points yield meager contributions to the total sum G . We present the results of a set of experiments showing how the size of an average constrained set varies with the dimension of the problem space.

Experiment 1. We ran the traversal algorithm on randomly constructed constrained regions in n -dimensional space, where $2 \leq n \leq 100$. A total of 840 such runs were made using uniformly randomly generated constraints and coefficients, and run-times were measured and graphed against the total number of points. The constant G was computed using all the points in each space, and the number of points traversed ranged from 10 to 25×10^6 , with the number of points within the randomly generated sets \mathbf{R} ranging from 6 to 17×10^6 (see Figure 1).

The correlation of 0.95 in Figure 2 suggests that, on average, the number of points on the boundary is directly proportional to the number of points within the constrained set, with the constant of proportionality being approximately $1/2$.

The determination of the number of points d within an arbitrary set \mathbf{R} is a nontrivial

problem. Since the run-time complexity of the algorithm is $O(d)$, it is instructive to get a handle on how d varies with the parameters defining set \mathbf{R} , at least in some “average” sense. We accomplished this experimentally in the following manner. First we fixed $C_j = C$, for $1 \leq j \leq m$, without suffering any loss in generality as far as measures of run-time are concerned. Second, values of the constraint coefficients $r_{j,k}$ were generated uniformly randomly from integers in the set $\{1, \dots, 10\}$, for $1 \leq j \leq m$, and $1 \leq k \leq n$. A single application of the depth-first tree algorithm on a randomly generated “problem-space” (i.e., one complete set of randomly generated constraint coefficients $r_{j,k}, \forall(j,k)$) offered a single observation of the algorithm’s run-time, for fixed n and C . The independent generation of 30 such problem-spaces, with associated tree traversals, offered 30 observations from which an average, maximum, minimum and standard-error were obtained, for fixed n and C . The next step was to repeat the procedure for different values of n and C , in order to study the effect of dimensionality on run-time.

Experiment 2. The objective here was to examine the relationship between dimensionality n and the number of lattice-points d the algorithm must traverse, on average. Runs were made for $n = 2, 3, 5, 7, 8, 9, 11$ and 12 , with $C = 100$, and $m = n$, with average, maximum, minimum and standard deviation obtained over 30 independent runs on uniformly randomly generated constrained spaces, as before. The results of this experiment can be seen in Figure 3 and also in in Table 1, which presents the minimum (d_{min}) and maximum (d_{max}) sizes of the constrained sets, along with average size \bar{d} and standard-error (σ_d) over 30 independent runs, for each value of n .

The last two columns of the table present relative efficiency ratios $\bar{\eta} = \bar{d}/nC^n$ and $\eta_{max} = d_{max}/nC^n$ for the run-times given by the worst case and average case complexity, respectively, of our tree-based search algorithm; here, nC^n is the actual number of operations required by the generating-function based multidimensional recurrence presented in [16]. It is interesting to note that both ratios converge rapidly to 0 for relatively small but increasing n , with the average case converging slightly faster.

n	d_{min}	d_{max}	\bar{d}	σ_d	$\bar{\eta}$	η_{max}
2	80	630	210.97	24.06	0.010548	0.0315
3	384	1953	847.23	76.21	0.000282	0.000651
5	4714	25271	11755.10	933.58	2.35e-07	5.05e-07
7	49813	305678	124870.77	12644.34	1.78e-10	4.37e-10
8	141599	985526	348653.33	38856.70	4.36e-12	1.23e-11
9	298454	1549614	792181.23	67874.05	8.8e-14	1.72e-13
11	2012261	8824529	4366967.33	336553.11	3.97e-17	8.02e-17
12	4599583	16895489	9223994.53	601605.51	7.69e-19	1.41e-18

Table 1: Statistics on n versus \bar{d}

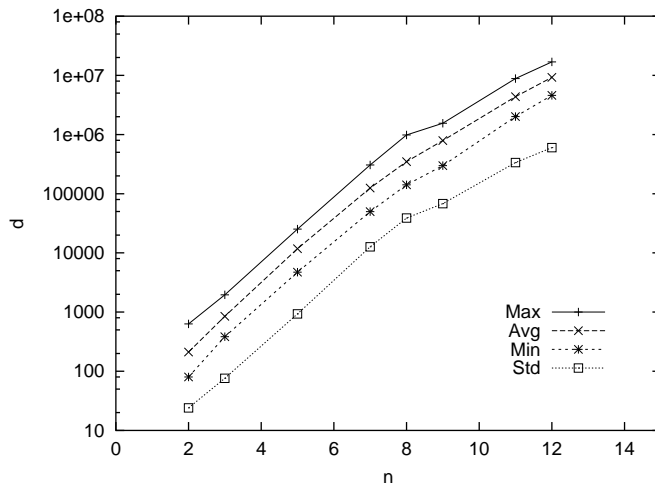


Figure 3: Number of points traversed vs. dimension n

Experiment 3. We graph the ratio $\gamma = C/r$ versus the number of points in the constrained set, for $\gamma = 5, 10, 15, 20, 30, 40,$ and 50 , $r = (1 + 10)/2 = 5.5$, and $m = n$. Here C is the average number of available resource units and r is the average size of a constraint coefficient. In Figure 4, the average number of points in a constrained set is graphed against dimension n .

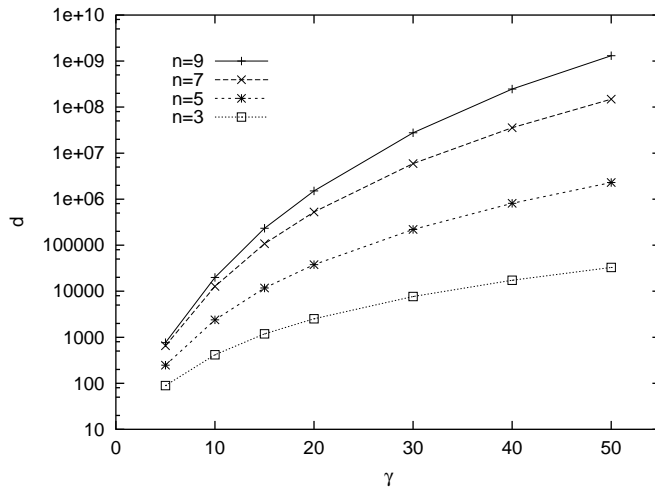


Figure 4: Number of points traversed vs. γ

Experiment 4. We graph the relationship between $\delta = m/n$ and the number of points in the constrained set, on average. Runs were made for a fixed value of $n = 8$, with γ ranging from 10 to 25 in steps of 5. The ratio δ took on the values $1/8, 1/2, 1/4, 1, 2, 3,$ and 4 . In Figure 5, the average number of points in a constrained set is graphed against selected values of γ , for increasing δ . Observe that beyond some point, an increasing number of constraints m reduces run-time complexity, but with diminishing returns.

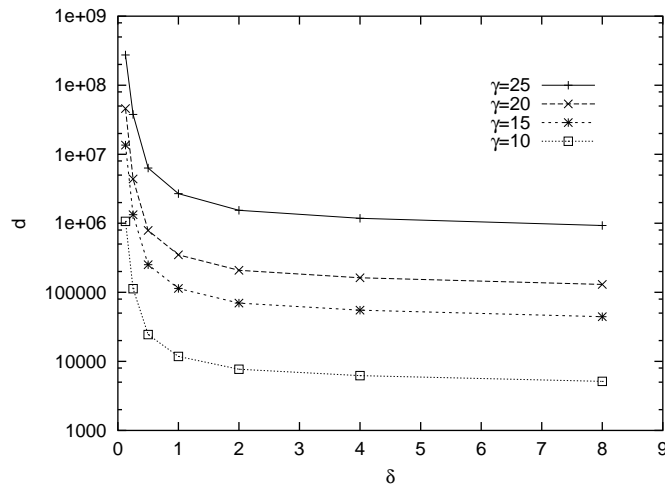


Figure 5: Number of points traversed vs. δ

References

- [1] T.H. Cormen, *et al.* *Introduction to Algorithms*, MIT Press, McGraw-Hill, MA, 2001.
- [2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [3] J. J. Gordon. The evaluation of normalizing constants in closed queueing networks, *Opns. Res.*, 38(1990), pp. 863–869.
- [4] P. G. Harrison. On normalizing constants in queueing networks. *Opns. Res.* 33 (1985), pp. 464–468.
- [5] K. W. Ross. *Multiserver Loss Models for Broadband Telecommunication Networks*, Springer-Verlag, London, 1995.
- [6] P. Lassila and J. Virtamo. Nearly Optimal Importance Sampling for Monte Carlo Simulation of Loss Systems. *ACM Trans. Modeling and Computer Simulation*, 10, pp. 326-347, 2000.
- [7] F. Reif. *Fundamentals of Statistical and Thermal Physics*. McGraw-Hill, New York, 1965.
- [8] D. McLachlan, Jr. *Statistical Mechanical Analogies*. Prentice-Hall Series in Materials Science, New Jersey, 1968.
- [9] A. E. Conway *et al.* RECAL: A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33, pp. 768–791, 1986.

- [10] S. S. Lam and Y. L. Lien. A tree convolution algorithm for the solution of queueing networks. *Commun. ACM*, 26, pp. 203–215, 1986.
- [11] J. McKenna and D. Mitra. Integral representations and asymptotic expansions for closed Markovian queueing networks: normal usage. *Bell System Tech. J.*, 61, pp. 661–683, 1982.
- [12] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41, pp. 480-488, 1992.
- [13] G. L. Choudhury, *et al.* Calculating Normalization Constants of Closed Queueing Networks by Numerically Inverting Their Generating Functions. *J. ACM*, 42, pp. 935–970, 1995.
- [14] G. L. Choudhury, *et al.*, Numerical transform inversion to analyze teletraffic models. *The Fundamental Role of Teletraffic in the Evolution of Telecommunication Networks, Proc. of the 14th Intl. Teletraffic Congress*, Elsevier, Amsterdam, 1b, pp. 1043-1052, 1994.
- [15] H. Dubner and J. Abate. Numerical Inversion of Laplace transforms by relating them to the finite Fourier cosine transform. *J. ACM*, 15, pp. 115-123, 1968.
- [16] E. Pinsky and A. Conway. Exact computation of blocking probabilities in state-dependent multi-facility blocking models. *Proc. of Performance of Distributed Systems and Integrated Communication Networks, IFIP Transactions*, North-Holland, pp. 383-392, 1992.
- [17] J. R. Ramos and V. Rego. Multidimensional Traversals for Normalization Constants. *Purdue CSD-TR, Department of Computer Sciences*, Purdue University, 2003.