Structured Markov chains solver: software tools

D. A. Bini, B. Meini, S. Steffé[*]
Dipartimento di Matematica
Università di Pisa, Pisa, Italy
bini, meini, steffe @dm.unipi.it

B. Van Houdt[†]
Department of Mathematics and Computer Science
University of Antwerp, Antwerpen, Belgium
benny.vanhoudt@ua.ac.be

ABSTRACT
The package SMC-Solver for solving structured Markov chains is presented. It contains the most advanced algorithms for solving QBD, M/G/1 and G/M/1 problems. The package is provided in two versions: a Matlab toolbox and a Fortran 95 version with a user-friendly graphical interface.

1. INTRODUCTION
The package Structured Markov Chains Solver (SMC-Solver) for solving the most important classes of Markov chains encountered in queueing models is presented. This package implements the most advanced available algorithms for QBD, M/G/1 and G/M/1 problems. The main features of these algorithms are reported in the paper [2] to which we refer the reader for more details.

The algorithms have been implemented in two different software tools: a matlab Toolbox where each method and a set of auxiliary algorithms have been implemented as Matlab functions, and a Fortran 95 package where a set of subroutines are given as a module. A user-friendly graphical interface which allows an easy use of the software tool is provided. The interface is written in C and relies on the GTK graphic libraries.

In this paper we describe the main features and the use of these two software tools, and report the results of a set of numerical experiments. The software tools are available upon request from the authors and will be made available online.

The main problems solved by our software tools are described below.

[*]This work has been partially supported by MIUR grant number 2004015437.
[†]B. Van Houdt is a postdoctoral fellow of the FWO-Flanders.

1.1 QBD problems
1.1.1 Computing G, R and U
Given nonnegative matrices A_{-1}, A_0, A_1 such that $A_{-1} + A_0 + A_1$ is stochastic and irreducible, compute the minimal nonnegative solutions G, R and U of the equations

$$\begin{aligned} G &= A_{-1} + A_0 G + A_1 G^2, \\ R &= A_1 + R A_0 + R^2 A_{-1}, \\ U &= A_0 + A_1 (I - U)^{-1} A_{-1}. \end{aligned} \tag{1}$$

1.1.2 Computing $\boldsymbol{\pi}$
Given in addition the nonnegative matrices B_0, B_{-1} such that $B_{-1} + A_0 + A_1$ and $B_0 + A_1$ are stochastic, if the QBD is positive recurrent, compute the steady state vector $\boldsymbol{\pi}$ of the QBD defined by A_{-1}, A_0, A_1 and B_{-1}, B_0, see [2, Section 2.1].

1.2 M/G/1-type Markov chains
1.2.1 Computing G
Given the nonnegative matrices A_i, $i = -1, 0, 1, \ldots$, such that $\sum_i A_i$ is stochastic and irreducible, compute the minimal nonnegative solution G of the equation

$$G = \sum_{i=-1}^{+\infty} A_i G^{i+1}. \tag{2}$$

In the applications, the infinite sequence of data A_i, $i = -1, 0, 1, \ldots$, is truncated to the finite size K such that $\sum_{i=-1}^{K} A_i$ is numerically stochastic. In this way, the equation (2) turns into

$$G = \sum_{i=-1}^{K} A_i G^{i+1}. \tag{3}$$

1.2.2 Computing $\boldsymbol{\pi}$
Given in addition the nonnegative matrices $B_0, B_1, \ldots$ such that $\sum_i B_i$ is stochastic, if the Markov chain is positive recurrent, compute the steady state vector $\boldsymbol{\pi}$ associated with A_i and B_i, see [2, Section 2.2].

1.3 G/M/1-type Markov chains
1.3.1 Computing R
Given the nonnegative matrices A_i, $i = -1, 0, 1, \ldots$, such that $\sum_i A_i$ is stochastic and irreducible, compute the minimal nonnegative solution R of the equation

$$R = \sum_{i=-1}^{+\infty} R^{i+1} A_i. \tag{4}$$

1.3.2 Computing $\boldsymbol{\pi}$

Given in addition the nonnegative matrices $B_0, B_1, \ldots$, if the Markov chain is positive recurrent, compute the steady state vector $\boldsymbol{\pi}$ associated with A_i and B_i, see [2, Section 2.3].

1.4 Non-Skip-Free Markov chains

Given the nonnegative matrices A_i, $i = -N, -N+1, \ldots$, such that $\sum_i A_i$ is stochastic and irreducible, compute the minimal nonnegative solution $\mathcal{G}$ of the equation

$$\mathcal{G} = \sum_{i=-1}^{+\infty} \mathcal{A}_i \mathcal{G}^{i+1}. \quad (5)$$

where $\mathcal{A}_i$ are the matrices defined in [2, Section 2.4].

2. THE MATLAB TOOL

The MATLAB implementation of the tool consists of a collection of MATLAB functions which can be executed from the command line (or called from within other functions or scripts). This collection can be partitioned in two subsets: (A) the functions related to Quasi Birth-and-Death (QBD) Markov chains and (B) functions used to solve M/G/1, G/M/1 and Non-Skip-Free type Markov Chains. Each function takes at least one *required* parameter as input and may support several *optional* parameters. A call to a function, called fname, uses the following syntax:

$$\mathit{output\_para} = \text{fname}(\mathit{required\_para}, \mathit{optional\_para})$$

If the function produces a single output parameter O1, one simply replaces *output_para* by O1. In case of multiple outputs, O1, O2, ..., Ok, one sets *output_para* equal to [O1,O2,...,Ok]. If the user is only interested in the first $l < k$ output variables, it suffices to shorten the list to Ol. As with any other MATLAB function, the *required_para* field holds the list of required parameters separated by commas: R1,R2,...,Rr.

The *optional_para* field contains the list of optional parameters. When a function call uses no options, one simple passes the *required_para* field to the function. Otherwise, a pair of inputs must be given for each optional parameter used: the parameter name (pname) and the parameter value (pvalue). The name is always placed between single quotes, the value is placed between quotes if it holds a string (and not a numeric value). The parameter name has to be followed by its value, the order of the optional parameters on the other hand is arbitrary. Hence, in case of t optional parameters we have

$$\mathit{optional\_para} = \text{'pname1',pvalue1,\ldots,'pname}t\text{',pvalue}t$$

The tool parses all optional parameters using the support function *ParseOptPara.m*. As with any inbuilt MATLAB function, help can be requested for a function part of this tool by typing ‘help fname’ on the command line. Before discussing the various functions of the MATLAB tool, we introduce some optional parameters that are supported by most functions.

2.1 General Optional Parameters

2.1.1 MaxNumIt and MaxNumComp

All functions that compute the G and/or R matrix are of an iterative nature, except for the Invariant Subspace approach that relies on a Schur decomposition. To avoid infinite loops, a default maximum number of iterations is present for all such functions. The default value is typically about 50 for the algorithms with quadratic convergence and 10000 for those with linear convergence. One can adapt this maximum through the optional parameter *MaxNumIt*. Its parameter value is simply the maximum number of iterations allowed. Whenever this maximum is reached before satisfying the stopping criteria, a warning is generated. For the functions that compute the components $(\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots)$ of some steady state vector $\boldsymbol{\pi}$, the number of components computed is determined dynamically such that the sum of the returned components is at least $1 - 10^{-9}$. To avoid loops, there is a default maximum of 500 components, which can be altered through the optional parameter *MaxNumComp*.

2.1.2 Verbose

To get an idea of the final accuracy and progress of a function call, one can set the *Verbose* option to 1. For the iterative algorithms, an intermediate check value is printed at the end of each iteration, together with the final residual error of the G and/or R matrix computed. As the number of iterations of the functional iteration (FI) approach is often quite high, one can also assign an integer parameter value $k > 1$ to the *Verbose* parameter. This will cause a printout of the check value every k iterations. The default value of this optional parameter is 0, indicating that no feedback is provided. For the functions that determine the steady state probabilities, setting *Verbose* to 1, causes MATLAB to print the accumulated probability mass of the components computed so far.

2.1.3 Mode

A single MATLAB function exist for each family of algorithms that computes the G and/or R matrix. Examples of families are Cyclic Reduction, Functional Iterations, Invariant Subspace approach, etc. Within each family one can still choose among different approaches/algorithms. The optional parameter *Mode* determines the family member used to perform the computations. A default member was selected among each family. The specific modes for each of the functions concerned is discussed further on.

2.2 Quasi Birth-and-Death functions: computing G, R and U

The tool contains five functions to compute the G (R and U) matrix of a QBD Markov chain. Each of these functions takes three *required* input parameters: the matrices A_{-1}, A_0, A_1 defined in Section 1.1. Here and hereafter, the three input matrices A_{-1}, A_0, A_1 are denoted by the variables A0, A1 and A2, respectively, so that A0 is the matrix appearing below the main diagonal and A2 above it. Thus, G is the solution to $G = \text{A0} + \text{A1}G + \text{A2}G^2$. The *output_para* field equals [G, R, U], meaning the user can either request the G, the G and R matrix or all three matrices as output variables. Each of these functions first parses the input and optional parameters by calling the *QBD_ParsePara.m* and *ParseOptPara.m* function. Afterward, a check is performed to see whether one of the matrices A0 or A2 is of rank one, allowing us to provide an explicit solution for G, this is realized by means of the function $QBD\_EG.m$, which we discuss in more detail further on. Provided that no explicit solution is available, the actual computation of G starts via the algorithm of choice.

2.2.1 QBD_CR.m and QBD_LR.m

The Cyclic and Logarithmic Reduction (CR and LR) converge quadratically and both support two modes of operation: the *Basic* and the *Shift* mode. The difference between both modes exists in the fact that the *Shift* mode performs some pre- and postprocessing. More specifically, with the shift we first compute the matrices Ã0, Ã1 and Ã2 by performing the shift operation on A0, A1 and A2. Next, we apply the CR/LR algorithm to Ã0, Ã1 and Ã2, instead of A0, A1 and A2. The postprocessing occurs only in the positive recurrent case and exists in performing a rank one correction to G. The *Shift* mode is the default mode as it may potentially speed-up the convergence of the CR/LR algorithm. The additional cost of pre- and postprocessing is minor compared to the actual CR/LR execution time. The shift also makes the convergence quadratic in the null recurrent case. The shift operation can be turned off by setting the *Mode* option to *Basic*. Both functions support the *MaxNumIt* and *Verbose* option.

2.2.2 QBD_FI.m

The function based on Functional Iterations (FI) achieves linear convergence and has six modes of operation. The first three are the *U-Based*, *Natural* and *Traditional* iteration. The *U-Based* mode is the default mode as it outperforms the other two in terms of the number of iterations. Three more modes—*ShiftU-Based*, *ShiftNatural* and *ShiftTraditional*—are obtained by combining each scheme with the shift operation, to possibly reduce the number of iterations. Apart from the *MaxNumIt* and *Verbose* option, this function also supports the *StartValue* option. This option allows the user to select another initial matrix G_0, used during the first iteration, where the default choice is $G_0 = 0$. A good choice for recurrent chains is to set $G_0 = I$, while for the transient case having $G_0 = \eta I$ might result in a significant reduction in the number of iterations. The scalar $\eta < 1$ is the largerst eigenvalue of G, it can be found as the Caudal characteristic of the Ramaswami dual of the QBD process of interest (for its computation see $QBD\_Caudal.m$).

2.2.3 QBD_IS.m

The Invariant Subspace (IS) approach function has three modes of operation: the *Schur*, *MSignStandard* and *MSignBalzer*. In the Schur mode, the left invariant subspace of the matrix Z is computed using a Schur decomposition. This method, which is set as the default mode, is a direct method that requires no successive iterations. It relies on the builtin MATLAB function ordschur, which is only supported as of MATLAB 7. If an earlier MATLAB release is used, the function automatically switches to the *MSignBalzer* mode. The other two modes compute the invariant subspace of Z via the Matrix Sign function sign(Z), which we evaluate via a Newton iteration $Z_{k+1} = Z_k/2 + Z_k^{-1}/2$ (guaranteeing quadratic convergence). The *MSignBalzer* implements the Balzer acceleration to improve convergence by adapting the weights $1/2$ and computes the determinant of Z_k for this purpose. We take the maximum of 10^{-3} and the determinant's value to avoid the algorithm from stopping prematurely. This function does not support the null recurrent case.

2.2.4 QBD_NI.m

The Newton Iterations (NI) support three modes of operation: *Sylvest*, *Estimat* and *DirectSum*. They differ in the manner in which they solve the generalized Sylvester matrix equation of the form $AXB + XC = D$ at each iteration. The *Estimat* mode estimates the solution by letting $X = D - AX_{old}B + X_{old}(I - C)$ and therefore no longer guarantees quadratic convergence. The *Sylvest* mode, which is the default mode, solves the equation via an Hessenberg algorithm, implemented by the *QBD_NI_Sylvest.m* support function. Finally, the *DirectSum* mode reduces the problem to a (large) linear system of equations by taking the direct sum of both sides, it is mostly effective for small systems.

2.2.5 QBD_EG.m

This function determines an explicit solution for G (R and U) in case A0 or A2 is of rank one. For the recurrent case, we have an immediate expression for G if A0 is rank one, otherwise if A2 is of rank one, R can be expressed in terms of its dominant eigenvalue η (which is computed via the $QBD\_Caudal.m$ function). The transient cases are solved by combining the previous observations with the Ramaswami dual.

2.3 Quasi Birth-and-Death: steady state and support functions

Apart from the functions to compute G, the tool also provides a function to compute the steady state probability vector and the Caudal characteristic. Details on these functions are given below. Figure 1 shows the dependency graph of all the MATLAB functions related to the QBD Markov chains. Support functions are presented as white ellipses. Functions that support multiple modes are represented as square boxes holding a single gray ellipse per mode of operation. The default mode has an additional white periphery. An arrow is draw between two functions if the execution of the first requires a call to the second function. If a support function is only needed in a specific mode, the arrow departs from its corresponding ellipse.

2.3.1 QBD_pi.m

The tool also contains a function $QDB\_pi.m$ to compute the steady state probability vector $\boldsymbol{\pi} = (\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots)$. This function has three *required* input parameters: the matrices R, B0 and B1. B0 characterizes the transitions from level one to level zero and B1 from level 0 to itself. In fact, the variables B0 and B1 represent the matrices B_{-1} and B_0, respectively, of Section 1.1.2. The function returns the first $n+1$ components $\boldsymbol{\pi}_0$ to $\boldsymbol{\pi}_n$ if $\sum_{m>n} \boldsymbol{\pi}_m \boldsymbol{e} < 10^{-9}$ (see also Section 2.1.1). More general boundary conditions can be dealt with through the *Boundary* option. With this option, transitions from level zero to level one can occur according to a matrix B2 (in the default usage we have B2 = A2). Notice, B0 and B2 do not need to be square in general. The parameter value of the *Boundary* option should be set to the matrix [B2; A1+ R A2].

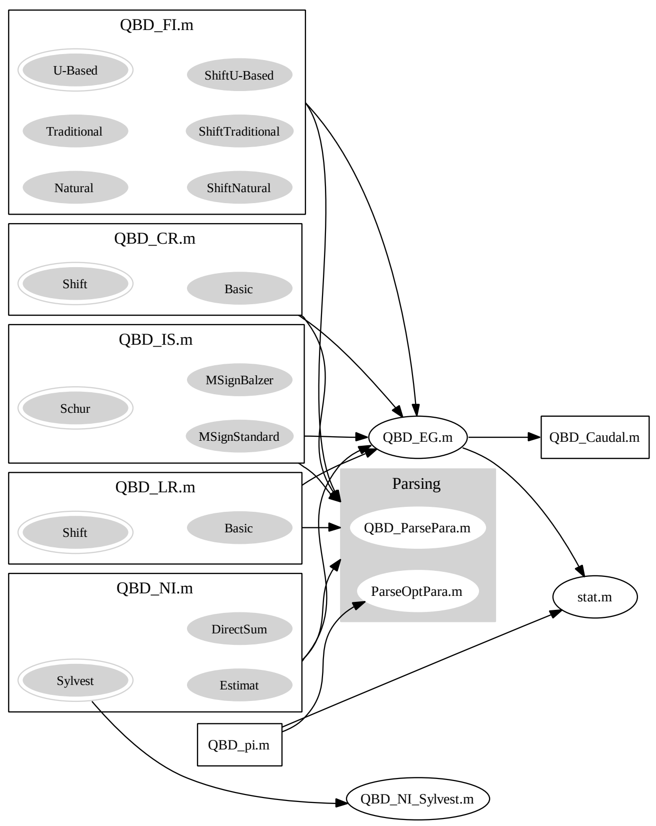


Figure 1: Dependency graph for the Quasi Birth-and-Death Functionality

2.3.2 *QBD_Caudal.m*

This function computes the dominant eigenvalue $0 < \eta < 1$ of the R matrix that solves the matrix equation $R =$ A2 $+ R$ A1 $+ R^2$ A0, where A0, A1 and A2 characterize a recurrent QBD Markov chain. This dominant eigenvalue is computed via a bisection algorithm and is also known as the Caudal characteristic. The function takes the matrices A0, A1 and A2 as *required* input parameters. Setting the option *Dual* to 1 causes *QBD_Caudal.m* to compute the dominant eigenvalue of the Ramaswami dual and must only be used for transient Markov chains. This option is useful for computing an alternate *StartValue* for the functional iteration (FI) algorithm.

2.4 M/G/1 and G/M/1 functions: computing G and R

All functions implementing an algorithm to compute the G or R of an M/G/1 or G/M/1 type Markov chain, respectively, demand a single *required* input parameter: the matrix A = [A0 A1 A2 ... Amax], where the variables A0, A1, ..., Amax denote the matrices $A_{-1}, A_0, A_1, \ldots, A_K$ of equation (3). For M/G/1 type Markov chains, A0 is the block appearing below the main diagonal, for the G/M/1 type, A0 appears above the main diagonal. Four functions are supported to compute the G matrix of the M/G/1 problem, while a single function, that relies on these four functions via the Ramaswami dual, is available for the G/M/1 setting. Each of these functions start, via the support function *MG1_EG.m*, by checking whether G can be determined explicitly, which is the case if A0 is of rank one. The dependency graph for the M/G/1 and G/M/1 functions is depicted in Figure 2

2.4.1 *MG1_CR.m*

The Cyclic Reduction (CR) algorithm achieves quadratic convergence and supports two modes of operation: the *ShiftPWCR* and the *PWCR* mode. As in the QBD case, the difference between the two modes exists in the pre- and postprocessing performed under the *ShiftPWCR* mode. The preprocessing is, in this case, taken care of by the support function *MG1_Shift.m*, which computes a new matrix Ã = [Ã0 Ã1 Ã2 ... Ãmax]. The *ShiftPWCR* mode is the default mode as it may potentially speed-up the convergence of the *CR* algorithm. The additional cost of pre- and postprocessing is limited compared to the actual *CR* execution time. The shift operation can be turned off by setting the *Mode* option to *PWCR*. The *PWCR* algorithm itself computes the matrix power series $A^{(n+1)}(z)$ and $\hat{A}^{(n+1)}(z)$ (see [2, Section 4.3]) through a point-wise evaluation. The point-wise evaluation is implemented by dynamically doubling the number of roots used in the FFT evaluation. A default maximum of 2048 roots is enforced for each iteration. One can adapt this number using the *MaxNumRoot* option. Whenever this maximum is attained, we generate a warning that the doubling process was interrupted prematurely. The *MG1_CR* function also supports the *MaxNumIt* and *Verbose* option.

2.4.2 *MG1_FI.m*

The Functional Iteration (FI) approach supports the same six modes of operations and optional parameters as its QBD counterpart (see Section 2.2.2 for details). An additional optional parameter for the M/G/1 setting is the *NonZeroBlocks* option. This option is mostly useful when the majority of the Ai blocks is equal to zero. Let $S = \{s1, \ldots, sn\}$ be the positive indices of the non-zero blocks of A. Then, setting the required parameter A=[A0 A$s1$ A$s2$ ... Asn] and the option value of *NonZeroBlocks* to [$s1$ $s2$ ... sn], may result in a substantial gain in the computation time. It is also useful to avoid unnecessary memory usage of the function call. The current implementation does not support the combination of the *NonZeroBlocks* option with either one of the *Shift* modes. If the user does provoke such a call, the shift operation is ignored.

2.4.3 *MG1_IS.m*

The Invariant Subspace (IS) function for M/G/1 type Markov chains has the same modes of operation and restrictions as the *QBD_IS.m* function (see Section 2.2.3). Except that the *MSignBalzer* is now the default mode, as it tends to be faster than the *Schur* mode. The current implementation only deals with the case where A$(z) = \sum_i$ Ai z^i is a matrix polynomial. The memory usage can be considerable as the dimension of the square companion matrix is a function of the degree of A(z). The IS approach is especially effective when A(z) is a rational power series (when the numerator and denominator have a fairly small degree). We plan to support such cases in a future implementation of the *MG1_IS.m* function.

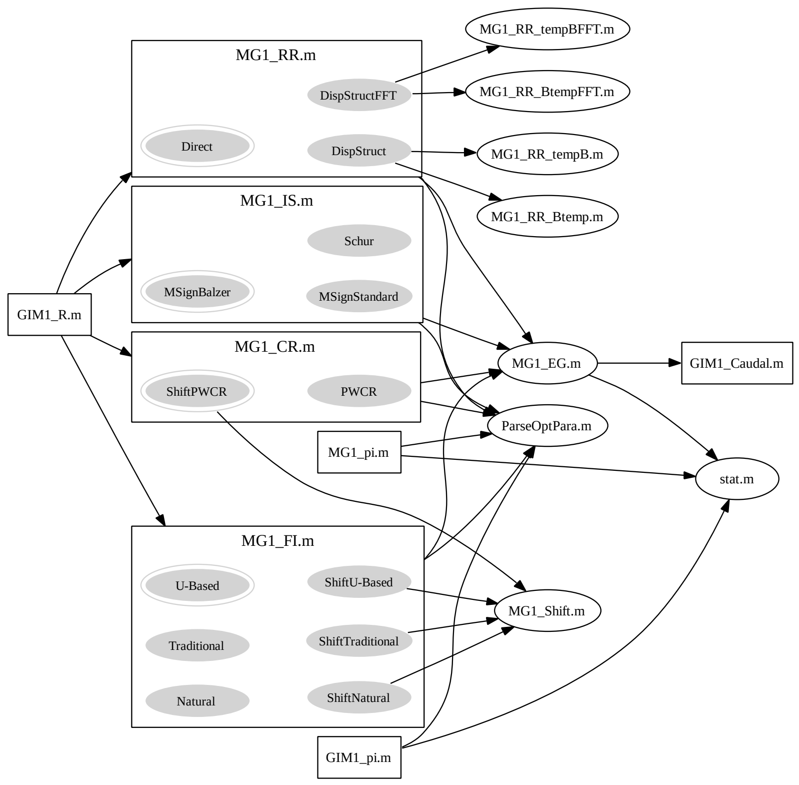


Figure 2: Dependency graph for the M/G/1 and G/M/1 Functionality

2.4.4 *MG1_RR.m*

The Ramaswami Reduction (RR) function converges quadratically and has three modes of operation: *Direct*, *DispStruct* and *DispStructFFT*. Let m be the block size of the original M/G/1 type Markov chain and N be the degree of the power series A(z). The *Direct* mode ignores the displacement structure of the matrices $A_1^{(n)}$ (see [2, Section 4.4]) and simply stores them entirely. The time and memory complexity per iteration therefore equals $O((mN)^3)$ and $O((mN)^2)$, respectively. Both other modes exploit the fact that $A_1^{(n)}$ can be expressed in terms of five block lower triangular block Toeplitz matrices (thereby reducing the memory usage to $O(m^2N)$). The *DispStructFFT* mode uses FFTs to compute all products between $A_1^{(n)}$ and a block column vector, whereas the *DispStruct* mode uses a standard approach. The resulting time complexity equals $O(m^2N^* \log N^* + m^3N^*)$, where N^* is the smallers power of 2 such that $N^* \geq N$, with the FFTs and $O(m^3N^2)$ without them. Both the *DispStruct* and *DispStructFFT* make use of two MATLAB support scripts (see Figure 2). For small systems, the *Direct* mode is superior and therefore set as the default mode. For larger systems the choice between the other two modes depends on the values of m and N.

2.4.5 *GIM1_R.m*

This function computes the R matrix of a G/M/1 type Markov chain. It first transforms the G/M/1 type problem characterized by the *required* input matrix A, into a M/G/1 problem by taking the Ramaswami dual À of A. Afterward, it passes À to one of the M/G/1 functions to compute the G matrix of the dual. The second *required* input parameter *Algor* specifies the algorithm of choice, it is one of the strings CR, FI, IS or RR. Any optional parameters, including the *NonZeroBlocks* option of the FI, for the M/G/1 call can be passes as optional parameters to the *GIM1_R.m* function. The R matrix of interest is retrieved as the dual of G.

2.5 M/G/1 and G/M/1: steady state and support functions

2.5.1 *MG1_pi.m*

By calling this function, one computes the steady state vector of a positive recurrent M/G/1 type Markov chain (see Section 2.1.1 for info on the number of components computed). There are three *required* input parameters: B = [B0 B1 B2 . . . B$bmax$], A = [A0 A1 A2 . . . A$amax$] and the matrix G. The matrix Bi characterizes the transitions from level 0 to level i. For the specific case where B = A, one simply assigns the empty matrix [ ] to B, i.e., *MG1_pi*([ ],A,G). In its default usage, transitions from level 1 to 0 are assumed to occur according to the matrix A0. The more general case, where C0 covers these transitions, can be treated via the *Boundary* option. The parameter value of this option must contain the matrix C0.

The components of $\boldsymbol{\pi}$ are computed via Ramaswami's formula. An implementation of the fast version of Ramaswami's formula, which is mainly effective if a very large number of components is required, is currently not supported.

2.5.2 *GIM1_pi.m*

The stationary vector of a positive recurrent G/M/1 type Markov chain is obtained through this function (see Section 2.1.1 for info on the number of components computed). It takes two *required* input parameters: B = [B1; B2; B3; ...; B$bmax$], where Bi governs the transitions from level $i-1$ to level 0 and the matrix R. For the more general case where the transitions from level 0 to 1 are described by a matrix B0 $\neq$ A0, one can evoke the *Boundary* option, the parameter value of which has to equal [B0; A1; A2; ...; A$amax$].

2.6 Non-Skip-Free Markov chain functions

Non-Skip-Free (NSF) type Markov chains can be solved in two manners. The first exists in reblocking the system to an M/G/1 type Markov chain and applying the standard algorithms for the M/G/1 type paradigm (i.e., the new block size equals Nm, where N was the number of blocks below the main diagonal and m the original block size). Alternatively, when computing G, one could also exploit the specific structure of the blocks as explained in [2, Section 4.1]. The tool contains two functions implementing the latter approach: *NSF_GHT.m* and *NSF_pi.m* which are discussed below.

2.6.1 NSF_GHT.m

This function computes the $mN \times mN$ size G matrix using the functional iterations developed by Gail, Hantler and Taylor (GHT), which converges linearly. It has one *required* input parameter A = [A0 ... Amax], where A0 appears N blocks below the main diagonal and AN on the main diagonal. The GHT algorithm first computes the first block row of G, which fully characterizes G, in an iterative manner and subsequently constructs the remaining block rows in a direct manner. Apart from the *Verbose* and *MaxNumIt* options, one can also set the *FirstBlockRow* option value to one. As a result only the first block row is returned as an output variable (which can useful to suppress the memory usage). The default value setting for *FirstBlockRow* is zero.

2.6.2 NSF_pi.m

The steady state vector of the NSF Markov chain can be computed through this function. It has three *required* input parameters: B, A and G. The B matrix holds the first N block rows of the transition matrix P, A equals [A0 ... Amax] as in the *NSF_GHT.m* function and G is the corresponding G matrix of size mN. Ramaswami's formula is used to generate the components of the $\boldsymbol{\pi}$ vector (the number of which can be affected by the *MaxNumComp* option). For NSF chains where each of the first N block rows is identical to A, one simply sets B equal to the empty matrix [ ]. In such particular cases we rely on a more efficient computation of the first N components of $\boldsymbol{\pi}$. When setting the *FirstBlockRow* option to one, it suffices to pass the first block row of G as input.

3. FORTRAN PACKAGE

Most part of the algorithms described in the paper [2] have been implemented in a Fortran 95 module. The subroutines can be called by the user inside a main program or, alternatively, can be used in an interactive way through a graphical interface written in C. The graphical interface relies on the GTK library and runs under the linux system. In this section we give an outline about the features and the use of the Fortran subroutines and of the graphical interface. The entire package, which is still work in progress, is available from the authors upon request.

3.1 The Fortran 95 subroutines

We report the main subroutines implemented in the package. The first two subroutines `crqbd` and `lrqbd` compute the minimal solutions G, R and U of the equations (1). They have the same syntax, i.e.,

```
subroutine crqbd(An1, A0, A1, doshift, dogth,&
     eps, maxiter, G, R, U, drift, err )
subroutine lrqbd(An1, A0, A1, doshift, dogth,&
     eps, maxiter, G, R, U, drift, err )
```

where `An1, A0` and `A1` are the matrix variables containing the matrices A_{-1}, A_0 and A_1 respectively; the logical variables `doshift` and `dogth`, if `.true.`, perform the shift acceleration of [2, Section 3] and the GTH trick of [5] for improving numerical stability. This trick cannot be applied if the shift acceleration is performed. The optional input variables `eps` and `maxiter` contain the error bound for the stop condition (by default `eps=1.e-12`) and the maximum number of allowed iterations (by default `maxiter=50`), respectively. The optional output variables `G, R` and `U` contain the solutions G, R and U, respectively; `drift` contains the value of the drift μ; while `error` is the residual error.

The subroutines `pwcr` and `spwcr` compute the solution of the equation (3) by means of the algorithm of point-wise cyclic reduction without the shift acceleration (`pwcr`) and with the shift acceleration (`spwcr`). Their syntax is the same, i.e.,

```
subroutine pwcr(A, eps, maxiter, &
     maxintp, G, drift, err)
subroutine spwcr(A, eps, maxiter, &
     maxintp, G, drift, err)
```

In both the subroutines, `A` is a three-way array where `A(:,:,k)` contains the block A_{k-2} for $k = 1, 2, \ldots, K+2$; the optional input variables `eps`, `maxiter` and `maxintp` contain an error bound used in the stop condition (by default `eps=1e-12`), the maximum number of iterations (by default `maxiter=50`) and the maximum number of interpolation points (by default `maxintp=256`), respectively. The output variables `G`, `drift` and `err` contain the solution G of the matrix equation (3), the drift μ and the residual error, respectively.

The subroutine `fi` computes G by means of functional iterations. The syntax is

```
subroutine fi(A, doshift, method, eps, &
     maxiter, x0, G, drift, err)
```

Here, `A` is a three-way array where `A(:,:,k)` contains the block A_{k-2} for $k = 1, 2, \ldots, K+2$; `doshift` is an optional logical variable, if `.true.` it performs the shift technique (default `doshift=.false.`); `method` is an optional integer variable which selects the functional iteration: 1$\rightarrow$Natural iteration, 2$\rightarrow$Traditional iteration, 3$\rightarrow$Method based on the matrix U (default `method=3`); the optional input variables `eps` and `maxiter` contain an error bound on the stop condition (by default `eps=1e-12`) and the maximum number of iterations (by default `maxiter=10000`); `X0` is an optional input variable which contains the initial approximation (by default `X0=0` if `doshift=.true.`, `X0=rI` if `doshift=.false.`, where `r`=$\rho(G)$). The output variables `G`, `drift` and `err` contain the solution G of the matrix equation (3), the drift μ and the residual error, respectively.

The subroutine `ni` computes G by means of Newton's iteration. The syntax is

```
subroutine ni(A, eps, maxiter, G, &
      drift, err)
```

Here, `A` is a three-way array where `A(:,:,k)` contains the block A_{k-2} for $k = 1, 2, \ldots, K+2$; the optional input variables `eps` and `maxiter` contain an error bound used in the stop condition (by default `eps=1e-12`) and the maximum number of iterations (by default `maxiter=50`). The output variables `G`, `drift` and `err` contain the solution G of the matrix equation (3), the drift μ and the residual error, respectively.

The subroutine `is` implements the invariant subspace method. Its syntax is

```
subroutine is(A, method, eps, maxiter, &
      G, drift, err)
```

Here, `A` is a three-way array where `A(:,:,k)` contains the block A_{k-2} for $k = 1, 2, \ldots, K+2$; `method` is an optional integer variable which selects the algorithm for computing the invariant subspace: $1 \rightarrow$ Matrix Sign Iteration; $2 \rightarrow$ Matrix Sign Iteration with the Balzer acceleration; $3 \rightarrow$ Schur Decomposition; by default `method=2`. The optional input variables `eps` and `maxiter` contain an error bound on the stop condition (by default `eps=1e-12`) and the maximum number of iterations (by default `maxiter=50`). The output variables `G`, `drift` and `err` contain the solution G of the matrix equation (3), the drift μ and the residual error, respectively.

3.2 The graphical interface

We have implemented a simple *Graphical Interface* in C relying on the **GTK+ 2.0** graphical libraries. The executable runs in Linux Boxes where the corresponding GTK-2 shared libraries are available. The Fortran routines are linked statically, so that the executable needs neither Fortran Compiler nor Fortran run-time libraries to run. A fully statically linked program is discouraged by GTK+ developers.

By means of menus in the main window, the user can easily choose the kind of problem (QBD, M/G/1, G/M/1), either by loading one of the examples provided in the package, or by loading the input data from some external ASCII files. In the first case, an example window allows the user to to set up the size of the problem and some parameters. Several formats are supported for the input files.

From the main menu the user can choose the desired algorithm among Cyclic Reduction, Logarithmic Reduction, Functional Iterations, Invariant Subspace. For each algorithm the user can set additional optional parameters like the shift acceleration, the diagonal adjustment, or can modify the values of the maximum number of iterations, the number of interpolations points in cyclic reduction, the error bound for the stop condition. For each algorithm, the user can select the goals, for instance, computing G, R, U or $\boldsymbol{\pi}$.

Warnings are flashed in the status line of the window for inconsistent choices of the user and also if the selected algorithms or options are not yet implemented.

The use of multithread support enables the user to start/stop a computation just by pressing a button.

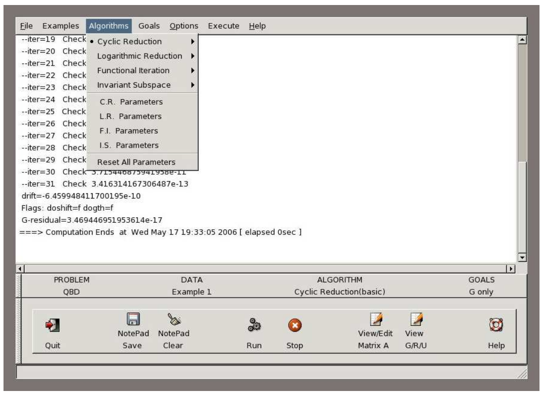
During the numerical computation, partial results, timings, and information on the evolution of the computation are displayed in real time on a scrollable viewport of the main window. The level of verbosity is selectable. The buffer with the log of the session can be saved in a file.

The computed solutions (matrices, vectors) can be saved in several ASCII formats. Morover a sparse matrix format is suppported for reading or writing data (the computations however do not use the sparse matrix format internally).

A viewing and editing graphical tool allows to display and edit input matrices and to view and compare output matrices. This is performed in a graphical way by representing each element of the matrices as a small gray (or colored) square, where the intensity varies according to the magnitude of the number. A good choice of the color scale allows an immediate and intuitive viewing of some structure properties of the matrix. The numerical values of the entries are displayed and edited in a separate small window as the user moves the cursor on the colored squares.

A simple online window help is available

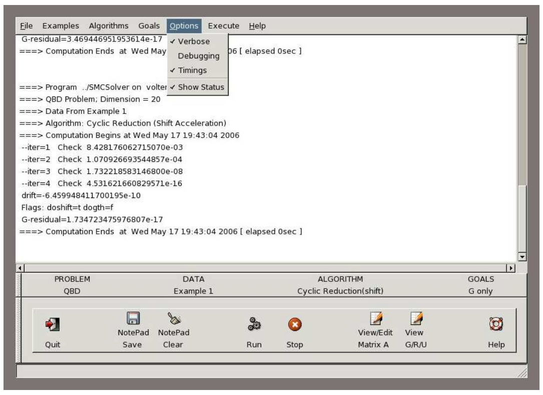
In Figures 3, 4, 5 we display some examples of the graphical interface.



```
 --iter=19   Check
 --iter=20   Check
 --iter=21   Check
 --iter=22   Check
 --iter=23   Check
 --iter=24   Check
 --iter=25   Check
 --iter=26   Check
 --iter=27   Check
 --iter=28   Check
 --iter=29   Check
 --iter=30   Check
 --iter=31   Check  3.416314167306487e-13
 drift=-6.459948411700195e-10
 Flags: doshift=f dogth=f
 G-residual=3.469446951953614e-17
===> Computation Ends  at  Wed May 17 19:33:05 2006 [ elapsed 0sec ]
```

PROBLEM	DATA	ALGORITHM	GOALS
QBD	Example 1	Cyclic Reduction(basic)	G only

Figure 3: Graphical User Interface: main window

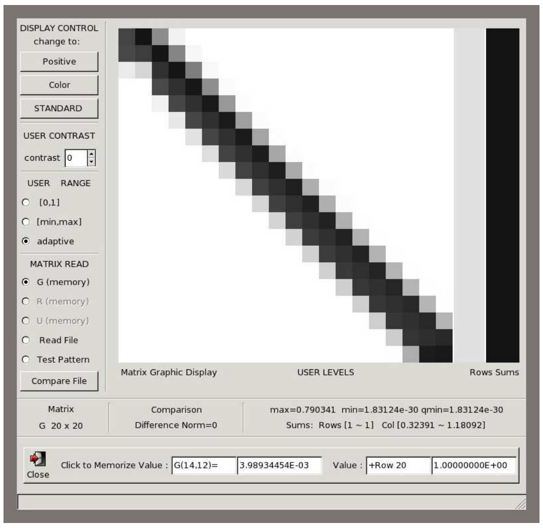


```
 G-residual=3.469446951953614e-17
===> Computation Ends  at  Wed May                06 [ elapsed 0sec ]


===> Program  ../SMCSolver on  volter
===> QBD Problem; Dimension = 20
===> Data From Example 1
===> Algorithm: Cyclic Reduction (Shift Acceleration)
===> Computation Begins at Wed May 17 19:43:04 2006
 --iter=1   Check  8.428176062715070e-03
 --iter=2   Check  1.070926693544857e-04
 --iter=3   Check  1.732218583146800e-08
 --iter=4   Check  4.531621660829571e-16
 drift=-6.459948411700195e-10
 Flags: doshift=t dogth=f
 G-residual=1.734723475976807e-17
===> Computation Ends  at  Wed May 17 19:43:04 2006 [ elapsed 0sec ]
```

PROBLEM	DATA	ALGORITHM	GOALS
QBD	Example 1	Cyclic Reduction(shift)	G only

Figure 4: Graphical User Interface: main window



contrast 0

[0,1]
[min,max]
[✓ adaptive]

MATRIX READ
[✓ G (memory)]
R (memory)
U (memory)
Read File
Test Pattern

Matrix G 20 x 20 Comparison Difference Norm=0 max=0.790341 min=1.83124e-30 qmin=1.83124e-30 Sums: Rows [1 ~ 1] Col [0.32391 ~ 1.18092]

Click to Memorize Value : G(14,12)= 3.98934454E-03 Value : +Row 20 1.00000000E+00

Figure 5: Graphical User Interface: displaying G

4. NUMERICAL EXAMPLES

Various numerical examples performed on some test problems are presented in this section. Since the Matlab and the Fortran versions implement the same algorithms, they perform similarly concerning the number of iterations and the residual error. For this reason, we report only the results of the Matlab version.

4.1 QBD problems

In this section we present some numerical examples to demonstrate the performance of the QBD MATLAB functions. This section also assists the user in making his/her choice between the different algorithms available. Of all tests conducted we report on four examples. It concerns (i) a QBD used to analyze a wireless MAC protocol named FS-ALOHA [4], (ii) the 32×32 QBD that was previously studied in [1] (with $\delta = 10^{-3}$), (iii) a chain with blocks of size 40 introduced in [6] to study the behavior of a video playout buffer and (iv) the QBD considered in [8] to analyze a discrete time tandem queue, it has blocks of size 388. We refer to these examples as *FS-ALOHA*, *DELTA*, *VIDEOPLAY* and *TANDEM*. The Markov chain for each of these examples is positive recurrent. Timing was performed by MATLAB on an Intel 2Ghz Pentium with 512 kB RAM in Realtime mode under Windows. As the MATLAB timing granularity on such a system is not very fine (in the millisecond range), most computations were averaged over several runs. Whenever the maximum number of iterations was reached, we only reported the computation time and residual error up to the maximum number of allowed iterations (which was lowered to 1000 for the *TANDEM* case, due to the large block size).

For the *FS-ALOHA* case, A_0 is of rank one, meaning that an explicit expression for G and R was available. All functions detected the rank one property through the *QBD_EG.m* function. Therefore, all algorithms required the same computation time without any iterations. Table 1 reports the results when computing the G, R and U matrix, in each of the remaining three examples. The values in the column labeled (R) hold the order of the maximum residual error (for the three matrices), e.g., having a residue of 2.5 10^{-12} implies that (R) equals -12. Let us briefly discuss a number of key observations. The *Shift* operation performed by the CR, LR and FI algorithm results in a strong decrease in the number of iterations in the *DELTA* example, whereas for the other two examples, there is either no or hardly any gain. This can be explained by the fact that the second largest eigenvalue η_2 of G is very close to one (above 0.99, which seems to be caused by the strong correlation in the arrival process of the corresponding queueing system) in the latter two examples, while for the *DELTA* case it is below 0.01. Whenever η_2 is not too close to one (in the positive recurrent case), some reduction may be expected by performing the shift. For the cases where the shift does not result in a reduction, only a limited amount of resources is waisted by performing it. Another observation related to the shift is that setting the *StartValue* equal to the identity matrix I seems to have a similar effect the shift operation when applied to the FI approach.

When comparing the CR and LR algorithm, we find that both perform very similar, except that the CR algorithm is slightly faster. We also observe that among the different IS modes, the *Schur* approach results in the shortest computation times in all examples. Although the NI algorithm often requires the lowest number of iterations, it tends to be considerably slower. Recall, the memory requirement of the *DirectSum* approach is $O(m^4)$, with m the block size; therefore, we did not apply it to the *TANDEM* example where $m = 388$.

4.2 M/G/1 type Markov chains

In this section we present a similar comparison, but for the M/G/1 type paradigm. Three examples where considered that each solve a D-BMAP/D/1 queue by means of an M/G/1 type Markov chain with a different arrival process, i.e., D-BMAP. In the first two examples the D-BMAP is a superposition of M ON/OFF sources as introduced in [3]. Both the ON and OFF period are geometric with a mean of $1/a$ slots. While on, a source generates a packet with probability $1/d$. In the first example, which we refer to as *VBR – 20*, we consider $M = 20$ sources, with a mean ON period of 100 slots and let $d = 10.5$, meaning the load of the queueing system equals .9524. In the second case, we obtain a transient chain by setting $M = 5$, $a = .0001$ and $d = 1.1$, meaning the load is very high and equals 4.5454. Finally, example three is the D-BMAP developed in [7] to approximate a superposition of various D-BMAPs, each used to model a variable bit rate source, by a circulant D-BMAP. In the first two examples both the block size m and the number of nonzero blocks N equals $M + 1$, while in the third we have $N = 25$ blocks each of size $m = 6$.

An overview of the results is shown in Table 2. We can see that the *Shift* is quite effective in the first and third example, as $\eta_2 = .9170$ and -0.3824, respectively. For the *VBR – 5* example we have $\eta_1 = .9996$ and $\eta_2 = .9946$. We also notice that even though the *Shift* might realize a reduction in the number of iterations of the CR algorithm, its computation time may be larger. This is caused by the higher number of roots required at each iteration to perform the point-wise evaluation. For all three examples, the fastest mode for the RR algorithm is the *Direct* mode. The other two modes

Example	*DELTA*			*VIDEOPLAY*			*TANDEM*		
	#It	Time	(R)	#It	Time	(R)	#It	Time	(R)
Cyclic Reduction (CR)									
Shift	3	.0084	-16	10	.0269	-15	12	13.16	-14
Basic	14	.0217	-16	10	.0242	-15	12	12.80	-15
Logarithmic Reduction (LR)									
Shift	3	.0092	-16	10	.0297	-15	12	16.96	-14
Basic	14	.0255	-16	10	.0283	-15	12	16.59	-15
Invariant Subspace (IS)									
Schur	1	.0344	-13	1	.0236	-15	1	24.08	-13
MSignBalzer	14	.0567	-12	13	.0708	-13	9	33.39	-12
MSignStandard	15	.0537	-11	14	.0758	-13	16	29.24	-14
Newton Iterarions (NI)									
Sylvest	14	.2364	-16	7	.2634	-15	9	278.6	-15
Estimat	6327	7.234	-13	450	.75	-13	1000+	899.1	-12
DirectSum	14	50.67	-16	7	84.37	-15	–	–	–
Functional Iterarions (FI)									
U-Based	6889	5.313	-16	526	.5541	-15	1000+	431.8	-7
ShiftU-Based	5	.0092	-16	481	.5106	-15	1000+	432.1	-8
U-Based + Start	5	.0078	-16	450	.5453	-15	1000+	431.5	-9
Natural	10000+	3.297	-10	1174	.5059	-14	1000+	215.9	-7
ShiftNatural	8	.0077	-16	1075	.4869	-14	1000+	216.2	-7
Natural + Start	9	.0072	-16	1004	.4666	-14	1000+	586.3	-8
Tradit.	10000+	4.14	-13	653	.4459	-15	1000+	404.6	-7
ShiftTradit.	5	.0080	-16	481	.3334	-15	1000+	405.0	-1
Tradit. + Start	6	.0078	-16	561	.3928	-15	1000+	815.8	-9

Table 1: Timing Results QBDs

Example	*VBR – 20*			*VBR – 5*			*CIRCULANT*		
	#It	Time	(R)	#It	Time	(R)	#It	Time	(R)
Cyclic Reduction (CR)									
Shift	11	5.242	-16	18	3.438	-16	7	.3562	-16
Basic	10	1.578	-15	17	3.928	-16	10	.1616	-14
Ramaswami Reduction (RR)									
Direct	11	1.922	-16	17	.0084	-16	12	.1256	-16
DispStruct	11	9.406	-16	17	.2000	-16	12	.7384	-16
DispStructFFT	12	26.85	-16	17	1.194	-16	12	2.155	-16
Invariant Subspace (IS)									
Schur	1	3.555	-7	1	.0094	-13	1	.1706	-5
Functional Iterarions (FI)									
U-Based	1147	1.101	-15	10000+	1.450	-7	1119	.3287	-15
ShiftU-Based	224	.2420	-15	10000+	1.462	-9	19	.0084	-16
U-Based + Start	206	.2190	-15	4503	.6594	-15	20	.0075	-16
Natural	2560	1.750	-15	10000+	.5844	-7	2270	.4612	-15
ShiftNatural	525	.3750	-15	10000+	.5906	-9	50	.0131	-15
Natural + Start	481	.3435	-15	5431	.3186	-15	51	.0125	-15
Tradit.	1635	1.640	-15	10000+	1.656	-7	1740	.5391	-15
ShiftTradit.	247	.2500	-15	10000+	1.637	-9	16	.0075	-15
Tradit. + Start	303	.3125	-15	4553	.7406	-15	36	.0134	-15

Table 2: Timing Results M/G/1 type

become more effective when N becomes large (recall, the memory complexity of the *Direct* mode is a square function of N). For the IS approach, we find that the residual error for the *Schur* mode is acceptable for the *VBR – 5* example, where there are only 6 nonzero blocks, which is no longer the case for the other examples. Additional experiments indicate that the IS method manages to produce a small residual error if the degree of A(z) is very limited. The other two IS modes of operation failed as, due to numerical errors, the dimension of the left invariant subspace did not match m (generating an error).

5. REFERENCES

[1] N. Akar and K. Sohraby. An invariant subspace approach in M/G/1 and GI/M/1 type Markov chains. *Stochastic Models*, 13(3):381–416, 1997.

[2] D. A. Bini, B. Meini, S. Steffé, and B. Van Houdt. Structured Markov chains solver: algorithms. *Proceedings of SMCTOOLS, Pisa 2006*, pages –, 2006.

[3] C. Blondia. A discrete-time batch Markovian arrival process as B-ISDN traffic model. *Belgian Journal of Operations Research, Statistics and Computer Science*, 32(3,4):3–23, 1993.

[4] D. V. Cortizo, J. García, C. Blondia, and B. Van Houdt. FIFO by sets ALOHA (FS-ALOHA): a collision resolution algorithm for the contention channel in wireless ATM systems. *Performance Evaluation*, 36-37:401–427, 1999.

[5] W. K. Grassmann, M. I. Taksar, and D. P. Heyman. Regenerative analysis and steady state distributions for Markov chains. *Oper. Res.*, 33(5):1107–1116, 1985.

[6] T. Hofkens, K. Spaey, and C. Blondia. Transient analysis of the D-BMAP/G/1 queue with an application to the dimensioning of a playout buffer for VBR traffic. In *Proc. of Networking 2004*, Athens, Greece, 2004.

[7] K. Spaey and C. Blondia. Circulant matching method for multiplexing ATM traffic applied to video sources. In *Proc. Perf. of Infor. and Comm. Systems (PICS)*, pages 234–245, Lund (Sweden), 1998.

[8] B. Van Houdt and A. Alfa. The response time in a discrete time tandem queue with blocking, markovian arrivals and phase-type services. *Operations Research Letters*, 33:4, 2005.