jPhase: an Object-Oriented Tool for Modeling Phase-Type Distributions

Germán Riaño and Juan F. Pérez
Department of Industrial Engineering
Universidad de los Andes
Bogotá D.C., Colombia
Email: griano@uniandes.edu.co
fern-per@uniandes.edu.co

CONTENTS

Abstract— **Phase-Type distributions are a powerful tool in stochastic models of real systems. In this paper, we develop a tool to represent these distributions as computational objects. It allows the computation of multiple closure properties that can be used in the modeling of large systems with multiple interactions. The tool also includes capabilities for generate random numbers from a specified distribution and procedures for fitting the parameter of a distribution from a data set. This framework is built in an extensible way, such that it is not limited to the already provided algorithms.**

INTRODUCTION

Phase-Type distributions are a general class of probability distributions that generalize the well known exponential distribution through the composition of exponential phases [7]. They were first introduced by Marcel Neuts [8], and have the important property of a rational Laplace transform, which makes them a subset of the distributions presented by David R. Cox [9] (even though the distributions proposed by Cox admit complex probabilities, the Coxian distributions treated in this document are all related to real probabilities) . In this case, the extension is made as a generalization of the method of phases proposed by Erlang, and has the relevant feature of numerical tractability, as noted by Neuts [8].

A usual problem in building probabilistic models is the assumption of exponential behavior in the interrarival and processing times. Continuous Phase-Type distributions can be used to avoid such difficulty since they are dense in $[0, \infty)$, and thus they can represent any behavior defined over this support. This makes Phase-type distributions a crucial tool in extending markov models to non-markovian behavior.

In this paper we present an object-oriented tool (jPhase) to model Phase-Type distributions in an computational framework, allowing the manipulation of these distributions as computational objects. The developed structure induces a formal representation of a Phase-type distribution and a set of properties that it should have. These properties are related to the computation of the probability density or mass function, the cumulative distribution function and the moments, among others.

In recent years, some computational tools have been developed to model large complex systems in an object-oriented fashion [10]. From a practical point of view, an analyst can build a computational probabilistic model that represents a real system using a traditional markov model but using Phase-Type instead of exponential distributions. In order to do that, it is important to find a good way for going from data sets to parameters of Phase-Type distributions. One of the complementary packages of the tool (jPhaseFit) has a set of classes to fit the parameters of a Phase-Type distribution from a data set, through the implementation of some recently developed algorithms. These classes are included in a computational structure that allows the characterization of the desired input and output of any fitting algorithm, in terms of computational objects.

To represent the complex relations that may arise in a real system, the analyst can also take advance of another important issue of Phase-Type distributions: their closure properties. The framework includes the implementation of many of these properties that can be used extensively in the building of a model.

This tool includes another complementary package (jPhaseGenerator), which establishes the structure for any Phase-Type random variates generator, and implements the algorithms developed by Neuts and Pagano [11] for the discrete and the continuous cases.

With this structure, any person with a basic knowledge in object-oriented programming can use Phase-Type distributions in the analysis of a system, through the fitting of real data to a Phase-Type distribution and the computation of some performance measures with the help of the procedures and utilities implemented. A graphic user interface was also developed in order to allows the interaction with the tool through the familiar windows, buttons and menu bars. This interface allows an easier interaction with the user, and can be used to make a relevant analysis of a real system, including data fit, closure properties computation, and graphical presentation of the probability density function and cumulative probability function.

This document is organized as follows: in the first section some basic definitions of Phase-Type distributions are presented, as well as an overview of the parameter fitting methods and the random variates generation algorithms. In sections II, III and IV, the developed computational structure is presented: the main, the generator and the fitter modules. In section V some illustrative examples are given, and some conclusions are stated in the last section.

I. Phase-Type Distributions

In this section, the definition and basic properties of Phase-Type distributions are stated, according to the treatment presented in [8] y [12]. Therefore, the proofs of the definitions in this section are not included and the interested reader can find them in the given references.

A Continuous Phase-Type distribution can be defined as the time until absorption in a Continuous Markov Chain, with one absorbing state and all others transient. The generator matrix of that process can be written as:

$$\mathbf{Q} = \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{a} & \mathbf{A} \end{bmatrix},$$

where the first entry in the state space represents the absorbing state. As the sum of the elements on each row must be equal to zero, $\mathbf{a}$ is determined by

$$\mathbf{a} = -\mathbf{A}\mathbf{1},$$

where $\mathbf{1}$ is a column vector of ones. In order to completely determine the process, the initial probability distribution is defined and can be partitioned in the same way of the generator matrix

$$\begin{bmatrix} \alpha_0 & \boldsymbol{\alpha} \end{bmatrix},$$

where α_0 is the probability of starting the process in the state 0, and the sum of all the components in the vector must be equal to 1. Therefore, α_0 is determined by the following relationship

$$\alpha_0 = 1 - \boldsymbol{\alpha}\mathbf{1}.$$

In this way, a Continuous Phase-Type Distribution is completely determined by the parameters (α, T), and its probability distribution function is defined as

$$F(x) = 1 - \boldsymbol{\alpha} e^{\mathbf{A}x}\mathbf{1}, \quad x \geq 0,$$

which has a clear connection to the well known exponential distribution. Furthermore, if there is just one transient phase with associate rate λ and it is selected with probability one, then the distribution is exactly the exponential case. From the previous expression, the probability density function can be computed as

$$f(x) = \boldsymbol{\alpha} e^{\mathbf{A}x}\mathbf{a}, \quad x > 0.$$

And similarly, the Laplace-Stieltjes transform of $f(\cdot)$, is given by

$$f(s) = \alpha_0 + \boldsymbol{\alpha}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{a}, \quad Re(s) \geq 0,$$

from which, the non-centered moments can be calculated as

$$E[X^k] = k!\boldsymbol{\alpha}(-\mathbf{A}^{-1})^k\mathbf{1}, \quad k \geq 1.$$

A Discrete Phase-Type distribution can be seen as an analogous case to the continuous distribution. In this case, the distribution can be defined as the number of steps until absorption in a Discrete Markov Chain, with one absorbing state and all other transient. The properties of this case can be found in [12].

A relevant issue of Phase-Type distributions is that they are closed under a set of operations, such as convolution, order statistics, convex mixtures, among other. These closure properties can be exploited in the modeling process of real systems as done in [13]. Particularly, continuous Phase-Type distributions have some extra closure properties: the distribution of the waiting time in a $M/PH/1$ queue, the residual time, the equilibrium residual time, and the termination time of a Phase-Type process with Phase-Type failures [14].

The Continuous and Discrete Phase-Type distributions have the important property of being dense in $[0, \infty)$ and the non-negative integers, respectively (the proof of this property can be found in [15]). This implies that any distribution with support on those sets can be approximated by a Phase-Type distribution with the appropriate number of phases and parameters $\boldsymbol{\alpha}$ and $\mathbf{T}$. In this sense, any non-negative behavior could be represented through these distributions, but this is not completely true because the number of phases needed could be infinite, which is computationally nonviable.

A. Phase-Type Random Variates Generation

In many large applications, simulation is the appropriate tool to model the system because of the complex relations between different stochastic variables. This makes that a random number generator become an important tool to model a wide range of non-deterministic systems. Neuts and Pagano [11] developed two similar algorithms to generate random variates from discrete and continuous Phase-Type distributions. These algorithms are supported on the alias method [16] to generate variates from discrete distributions in order to simulate the process of selecting an initial state and then jump to the next one according to random vectors.

B. Fitting Algorithms

In the last twenty years, the problem of fitting the parameters of a Phase-Type distribution has received great attention from the applied probability community. These different approaches can be classified in two major groups: maximum likelihood methods and moment matching techniques, as noted in [17]. Nevertheless, almost all the algorithms designed for this task have an important characteristic in common: they reduce they set of distributions to be fitted, from the whole Phase-Type set to a special subset. In section IV, those algorithms included in the computational package will be revisited and further explained.

II. jPhase: the object-oriented Framework

One of the contributions of this work is the design and implementation of an object-oriented framework that allows the computational manipulation Phase-Type distributions. To date, there is no academic or commercial software that can offer the capabilities of representation nor manipulation of Phase-Type distribution in a unified fashion. For example, with the developed tool, it is possible to create an object that represents a continuous Phase-Type distribution, calculate the value of its probability density function (pdf) or its cumulative distribution function (cdf), as well as any power moment. It is also possible to compute the minimum or the maximum between two distributions, as well as other closure properties, e.g. the distribution of the waiting time in a $M/PH/1$ queue.

Now, some of the most important issues about the computational structure will be discussed in order to give a good understanding of the framework. It must be said that the computational architecture is divided in three gross packages: `jPhase`, `jPhaseGenerator` and `jPhaseFit`. The first is related to the computational representation of Phase-Type distributions and will be explained in this section. The second builds the structure to implement Phase-Type random variates generators and will be discussed in section III. The last one offers a computational representation of Phase-Type fitting algorithms and will be explained in detail in section IV. The first package can be seen as the heart of the whole framework and the others are supported on it.

A. General Structure

The `jPhase` package is supported on a set of interfaces, abstract classes and concrete classes. The interfaces determine the characteristics of an object and have no implementation of any method. As can be seen in the simple Class Diagram of Figure 1, there are three interfaces in the jPhase package: `PhaseVar`, `ContPhaseVar`, and `DiscPhaseVar`. These interfaces determine the behavior of a PhaseType distribution in both the continuous and discrete cases.

The abstract classes `AbstractContPhaseVar` and `AbstractDiscPhaseVar` implements the corresponding interface (discrete or continuous), in order to develop some of the methods determined by the interfaces. Finally, the concrete classes extends the corresponding abstract class, and thus they make use of the already implemented methods. These methods are useful for any user that wants to develop an own concrete class, because he or she doesn't need to get worried about the whole set of distribution properties, but only needs to implement a little set of simple methods. In the next sections, the properties of these interfaces, abstract and concrete classes will be explained.

B. Interfaces

As it was said above, the jPhase package consists of three interfaces, that determine the behavior of any Phase-Type distribution as shown next.

- `PhaseVar`
  This interface defines the set of properties that are common to both discrete and continuous Phase-type distributions. Since this is the core interface in the framework, it has the major quantity of methods and all other interfaces ans classes have fewer. The methods that the interface force to implement for any distribution can be divided in three groups: access, moments and distribution methods.
- `DiscPhaseVar` and `ContPhaseVar`
  This interfaces determine some of the closure properties valid for discrete and continuous Phase-Type variables, as those discussed in section I. The methods defined by each one of this interfaces can be partitioned in two groups: distribution and closure methods. The closure properties can only be defined at this level because each one of the discrete and continuous sets are closed under these properties, but not the whole set of Phase-Type distributions. Some of the methods defined by the interfaces are shown in Table II, where all but the distribution-related methods apply for both cases. Next, in Table III some other methods are shown, but they are only defined for the continuous class, as discussed in section I.

C. Abstract Classes

As shown in Figure 1, the `ContPhaseVar` interface is implemented by the abstract class `AbstractContPhaseVar`, which implements almost all the methods defined by `PhaseVar` and `ContPhaseVar`. In particular, none of the methods implemented by this

Fig. 1. Simple jPhase Package Class Diagram

TABLE I
SOME METHODS FOR THE PHASEVAR INTERFACE

Type	Method	Result
Access methods	`getMatrix()`	Generator matrix $\mathbf{A}$
	`setMatrix(`A`)`	Set the transition matrix equal to the parameter
	`getVector()`	Initial probability distribution vector $\boldsymbol{\alpha}$
	`setVector(`α`)`	Set the initial probability vector equal to the parameter
	`getNumPhases()`	Number of transient phases in the distribution
	`getVec0()`	Value of α_0
	`getMat0()`	Exit rate vector $\mathbf{a} = -\mathbf{A1}$
	`copy()`	Deep copy of the distribution
Moments methods	`expectedValue()`	Expected value of the distribution
	`variance()`	Variance of the distribution
	`stdDeviation()`	Returns the standard deviation.
	`CV()`	Squared coefficient of Variance.
	`moment(`k`)`	k-th non-central moment of the distribution.
Distribution methods	`cdf(`x`)`	Cumulative distribution function at x
	`prob(`a`, `b`)`	Probability that the variable takes a value between a and b
	`survival(`x`)`	Survival function at x
	`lossFunction1(`x`)`	Value of the order-one loss function evaluated at x
	`lossFunction2(`x`)`	Value of the order-two loss function evaluated at x
	`quantil(`x`)`	Quantil x of the distribution
	`median()`	Median of the distribution

TABLE II
SOME METHODS FOR THE DISCPHASEVAR AND CONTPHASEVAR INTERFACE

Type	Method	Result
Distribution methods	`pmf(`x`)` or `pdf(`x`)`	Value of the probability mass function at x (discrete case) or the probability density function (continuous case)
Closure methods	`sum(`Y`)`	Convolution between the original distribution and Y
	`sumGeom(`p`)`	Sum of a geometric number (with parameter p) of i.i.d. Phase-Type distributions as the original one
	`sumPH(`Y`)`	Convolution of a discrete Phase-Type (Y) number of i.i.d. Phase-Type distributions
	`mix(`p`, `Y`)`	Convex mixture between the original distribution (with weight p) and Y
	`min(`Y`)`	Minimum between the original distribution and Y
	`max(`Y`)`	Maximum between the original distribution and Y
Other methods	`newVar(`n`)`	New n phase variable with the same representation as the original
	`toString()`	Returns a string representation of the Phase-Type distribution (including its associated vector and the matrix)

TABLE III
SOME FURTHER CLOSURE METHODS FOR THE CONTPHASEVAR INTERFACE

Method	Result
`times(`k`)`	Distribution of the variable scaled by k
`residualTime(`x`)`	Distribution of the residual time at x
`eqResidualTime()`	Distribution of the equilibrium residual time
`waitingQ(`ρ`)`	Distribution of the waiting time in a $M/PH/1$ queue with traffic coefficient equal to ρ

class depends on the formal representation of the matrices and vectors involved. This means that all the operations are executed using solvers and preconditioners that apply for both sparse and dense representations of matrices and vectors. Moreover the probably most difficult routines are solved by this abstract class, such as the computation of the probability density function, that implies the use of uniformization methods for solve a set of differential equations[12]. The same arguments apply for the abstract class `AbstractDiscPhaseVar`, that implements the interface `DiscPhaseVar`.

This way, the only methods that the user must implement when developing a concrete class that extends `AbstractContPhaseVar` or `AbstractDiscPhaseVar` are:

- getMatrix and setMatrix
- getVector and setVector
- newVar
- copy

As can be seen, this methods depend on the particular representation of the distribution, e.g. if the matrix is represented by a particular sparse pattern, then the only one class of matrices that can be set must have the same pattern. Also the `newVar` and `copy` methods must return a variable that belongs to the same class of the original one. The concrete classes explained in the next section are themselves examples of classes that extend the abstract ones.

D. Concrete Classes

The developed concrete classes are those that are a final user will usually utilize. They have been designed as general Phase-Type representations for the continuous and discrete cases, and with dense and sparse storage. The `DenseContPhaseVar` and `DenseDiscPhaseVar` are classes that represent continuous and discrete Phase-Type distributions, using the `DenseMatrix` and `DenseVector` classes defined by MTJ. This classes are useful for many applications, where the number of phases is not large and the memory is not a problem. They also have constructors for many simple distributions such as exponential or Erlang in the continuous case, and geometric or negative binomial in the discrete case.

Nevertheless, the use of matrices with dense representation can be a problem because of the large number of phases. The `SparseContPhaseVar` and `SparseDiscPhaseVar` classes are built over the `FlexCompRowMatrix` and `SparseVector` MTJ classes, which give a good alternative when the number of phases is large but the number of entries is little relative to the total number of n^2 entries. It is important to note that the `FlexCompRowMatrix` allows a flexible sparse pattern stored by rows, that makes of this class a general sparse representation. Other specific representation could be developed by using a particular sparse pattern, e.g. upperdiagonal matrices.

III. jPhaseGenerator: the Variates Generator Module

This package was developed in order to define the behavior of any Phase-Type random variates generator. This behavior is specified by the abstract class `PhaseGenerator`, which is the core the package. As can be seen in Figure 2, this abstract class is extended by the concrete classes `NeutsContPHGenerator` and `NeutsDiscPHGenerator`, that implement the algorithms proposed by Neuts and Pagano [11].

Fig. 2. Simple jPhaseGenerator Package Class Diagram

A. `PhaseGenerator` *Interface*

This abstract class defines the basic methods that a Phase-Type random variate generator should have. The class includes an attribute, that belongs to the `PhaseVar` class, and is the distribution from which, the random variates will be generated. This distribution can only be specified in the constructor method, because the variable must be persistent in time for a particular PhaseGenerator object. This means that if the user wants to generate variates from another distribution, he or she must create a new PhaseGenerator.

In the constructor method, the variable is assigned and the `initialization()` method is called. It is expected that the user employs this method in order to effectively initialize the algorithm, and then a random variate can be generated after the construction of the PhaseGenerator. Another method defined by the abstract class is `getVar()`, which always returns the Phase-Type variable that remain under the PhaseGenerator and is already implemented.

The last two methods that a PhaseGenerator must implement are `getRandom()` and `getRandom(`k`)`. The first one must return a variate that follows the distribution specified at the construction, and the second must return k independent variates with the same characteristic.

B. *Concrete Classes*

Up to day, two concrete classes extend the previously explained `PhaseGenerator` abstract class. These are `NeutsContPHGenerator` and `NeutsDiscPHGenerator`, which implement the method proposed by Neuts and Pagano [11]. The first one implements the continuous case and the second the discrete one. The continuous algorithm has a first step, in which the continuous chain is transformed is a discrete one, using the well-known embedded chain. Thereafter, the main algorithm (for discrete distributions) can be used for both cases.

The algorithm simulates the whole process in the chain: it first choose an initial state from the distribution given by the initial probability vector; then it selects a next state to visit using the discrete distribution associated with the present state, given by the associated row in the transition matrix; the selection of the next state is repeated until the chosen state is the absorbing one. In the discrete case, the value of the random variate is the number of steps (selections) made until absorption. For the continuous case, the number of visits to each state is stored and an Erlang variate is generated for each state with non-zero number of visits. The parameters of the Erlang distributions are the associated rate of the state and the number of visits carried out. For example, if the state i was visited n_i times and has an associated rate of λ_i, an Erlang(λ_i,n_i) random variate must be generated. The sum of these variates over all the states is the value of the Phase-Type random variate.

Two important issues of this algorithm must be emphasized. The first one is the several use of discrete distributions to generate the variates, which can be done efficiently through the alias method [16]. The second issue is that for the continuous case, in addition to the discrete variates, only Erlang variates must be generated. In the case of many visits to the same state, these variates can also be efficiently generated by multiplying a gamma variate with parameters $(n_i, 1)$ times λ_i, that will be an Erlang variate with the required parameters [11].

The algorithms implemented in these classes are supported by the utilities class `GeneratorUtils`, that have several procedures useful for the generators. Particularly, it has a general implementation of the alias method used to generate variates from discrete distributions [16]. It also has an implementation of the polynomial-time algorithm proposed by Gonzalez et. al. [18] to perform a Kolmogorov-Smirnov test, that can be useful to test the goodness-of-fit of the generated numbers in relation to the theoretic Phase-Type distribution.

IV. jPhaseFit: the fitting module

This package contains the structure that defines the behavior of the classes that implement algorithms to fit the parameters of a Phase-Type distribution. As shown in Figure 3, the interface `PhaseFitter` is in the top of the package and defines the basic method that any PhaseFitter should have: `fit()`. This method has no parameters and must return a Phase-Type variable as the result of the fitting process.

A. Abstract Classes

In the next level, there are two abstract classes that implement the `PhaseFitter` interface: `ContPhaseFitter` and `DiscPhaseFitter`, for the continuous and discrete case, respectively. These classes have two additional issues: a constructor method from a data set in array format; and a method to compute the log-likelihood of the fitted distribution in relation to the data set (`getLoglikelihood()`). This is done because the log-likelihood is a usual way to compare the performance of fitting algorithms. In addition, this classes specify the continuous or discrete nature of the variable to be fitted in two different ways: the first one is the inclusion of the `var` attribute, where the fitted variable must be stored (a `ContPhaseVar` object for the continuous case or a `DiscPhaseVar` for the discrete case); the other way is the use of a data array as attribute, that in the continuous case is a double array, and in the discrete case is an integer array.

In the next level of abstract classes, a further division is done between classes that implement Maximum Likelihood (ML) algorithms and those related to Moment Matching techniques. This is done for both continuous and discrete cases. For the ML classes (`MLContPhaseFitter` and `MLDiscPhaseFitter`), there is a new attribute called `logLH`, that stores the log-likelihood value in order to take advance of the usual computation of the log-likelihood in the fitting process. For the Moment-Matching related classes (`MomentsContPhaseFitter` and `MomentsDiscPhaseFitter`), a new set of attributes is defined: `m1`, `m2`, and `m3`. These are the moments to me matched and are specified with a new constructor that receives only the three moments to be matched. An alternative way is the use of the redefined constructor that receives the data trace and calculates its moments. It must be said that there is not alternative to change the data, moments of log-likelihood attributes from outside the class, implying a safe fitting process.

B. Concrete Classes: Maximum Likelihood Algorithms

The set of classes that implement maximum likelihood algorithms are almost all for Continuous Phase-Type distributions, because the most of the efforts have been done in that direction. For each one of the following algorithms, there is an associated class the executes the procedures to fit the parameters of a distribution.

1) General Phase-Type Distribution EM Algorithm [1]: In 1996 Asmussen, Nerman and Olsson [1] presented a specialized version of the EM algorithm in order to fit the parameters of the whole set of continuous Phase-Type distributions, without reducing the target distribution to a restricted subset. The EM algorithm is a general statistical technique that was first introduced by Dempster et. al. [19] to deal with the problem of incomplete data (a good source to review it may be [20]). The idea behind this algorithm is that a complete sample from Phase-Type realizations should include the selected initial state, the whole path of states followed until absorption, and the time spent in each of these states. With this complete sample, it's easy to estimate the parameters of the distribution.

Nevertheless the sample obtained from Phase-Type realizations are only the time until absorption. In this way, the problem can be seen as the estimation of the parameters from an incomplete sample, which makes natural the use of the EM algorithm. The algorithm begins from an initial guess of the parameters and the iterations include the computation of the likelihood (E-step) and the its maximization to obtain a new set of parameters (M-step). In this case, the heavy work must be done in the E-step, where a set of $n(n+2)$ linear differential equations must be solved for a distribution of n phases. It must be noted that this algorithm does not select the number of phases, and it must be entered as an initial parameter. Even though this algorithm has been already evaluated in [17], it is included in the benchmark evaluation because it is the only one that fits the parameters of the whole Phase-Type class.

The concrete class that implements the algorithm is `EMPhaseFit`, that extends the abstract class `MLContPhaseFitter`. In this implementation, the method `fit()` doesn't need the specification of any parameter but it tries with distributions from 1 to 10 to find the one that shows the greatest log-likelihood. To do this, it calls the method `fit(`n`)`, that executes the proper algorithm to fit the parameters of a general Phase-Type distribution with n phases. In every iteration, this method calls the `eStep()` and `mStep` methods that executes the procedures for each of those steps in the EM algorithm. Particularly, the E-step uses an order-four Runge-Kutta procedure to solve the set of differential equations, which solution is expressed in the inner class `solution`.

The user could also make use of another constructor for this class in order to specify some features for the algorithm. With the method `EMPhaseFit(precision, iterations, evalPoints)`, three important features can be set: the *precision* for stopping the algorithm when the parameters show little change; the maximum number of *iterations* that the algorithm can execute; and the *evalPoints* parameter determines the factor to multiply the data trace size in order to obtain the number of evaluation points for the Runge-Kutta method.

Fig. 3. Simple jPhaseFit Package Class Diagram

2) Hyper-Exponential Distribution EM Algorithm [2]: The hyper-exponential distribution is a very special case of Phase-Type distributions, since the initial probability vector defines the probability of choosing the exponential phase to visit, and the generator matrix have diagonal representation with the rates of the i-th phase in the position (i, i). Thus the number of parameter to fit a n-phase hyper-exponential distribution are $2n$. The algorithm proposed by Khayari et. al. [2] is also an EM algorithm like the explained above. It begins with an initial guess of the parameters, that can be random or related to the properties of the trace (e.g. the expected value). The authors propose an easy way to select the initial parameters. Then a function to evaluate the quality of the parameters is calculated in the E-step through the probability density function of the data trace given the parameters. In the M-step, the new set of parameters is computed using estimators for the rates and the probabilities but not for the number of phases, that is taken as a given parameter.

The implementation of the algorithm was done in the `EMHyperExpoFit` class, where a method `fit(`n`)` is implemented in order to fit a distribution with n phases. As the method `fit()` must also be implemented in order to follow the parameters of the `PhaseFitter` interface, it executes several trials of configurations from one to ten phases, and selects the distribution with greatest likelihood. With the use of another constructor, the user can also specify the maximum number of iterations that the algorithm can execute and the precision level required to determine when the change in the estimated parameters is too little and the algorithm should stop.

3) Hyper-Erlang Distribution EM Algorithm [3]: In 2005, Thümmler et. al. presented a method that fits the parameters of a hyper-Erlang distribution [3], which is a very interesting subset of the Phase-Type distributions since they are also dense in $[0, \infty)$. In some results provided by them, the EM algorithm developed for this special class has a better behavior in terms of likelihood than the one designed for the complete Phase family [1]. The algorithm receives as a parameter the number of Erlang branches in the distribution as well as the total number of exponential phases in the distribution. With this information, the algorithm determines all the possible configurations of the Erlang branches and executes a version of the EM algorithm for each case. Finally, the configuration with the greatest likelihood is selected as the result of the algorithm.

As can be seen, this algorithm needs more information than the previous ones, and so the routine `fit()` makes a different work than just try distributions with one to ten phases. In the `EMHyperErlangFit` class, the method `fit()` guides the search of the configuration by means of the coefficient of variation of the data trace. When the coefficient is lower than one, then it doesn't allow more than one branch since it has been shown that the Phase-Type variable with the least coefficient of variation is the n-Erlang($\frac{1}{n}$) [21]. When the coefficient of variation is greater than one, it enforces the creation of multiple branches as well as phases in each of them. The method `fit(`n, m`)` executes the effective procedure proposed in the paper for a distributions with n phases and m branches. When all the possible configurations has been determined, this method calls `fit(`n, m, r`)` where the number of phases at the i-th branch is r_i. For this method, the parameters related to precision and number of iterations are important and the user can fix them with use of a special constructor.

C. Concrete Classes: Moment Matching Algorithms

The distribution moments usually play an important role in the performance analysis of real systems [5]. This has been an important motivation for the improvement of moment matching techniques, and the attention given by different research communities (Operations Research, Computer Science and Telecommunication Networks, among others). Some of the most recent advances have been implemented in the jPhaseFit module, as will be explained in this section.

1) Acyclic Continuous order-2 Distributions [4]: In 2002 Telek and Heindl [4] proposed an algorithm to fit the parameters of an acyclic Phase-Type distributions of second order (two phases). Acyclic distributions have been extensively studied since they have some important properties, as a canonic form developed by Cumani [22] and a upper triangular transition or generator matrix. In that paper, they establish bounds on the set of first three moments representable by acyclic distributions of second order, for the discrete and continuous cases. Over the characterization of these bounds, they build the algorithm that matches three moments with the three parameters of this distribution: the rates of each phase and the absorption probability after the first phase (the initial probability is all in the first phase as in the Coxian distribution).

This algorithm is implemented by the class `MomentsACPH2Fit`, that extends the abstract class `MomentsContPhaseFitter`. As it is constructed with the three moments to be matched (given explicitly or computed from a data trace), the algorithm begins with the computation of the bounds, in order to determine if the moment set is representable. If not, the moments are corrected to the nearest point in the representable region with a warning message about the correction for the user. When the moment set is representable, the parameters of the distribution are calculated according to the equations shown by the authors. Finally, the distribution is constructed with the parameters and returned to the user.

In the same paper, the authors present an analogous algorithm for the discrete case. It works in a similar fashion and is completely implemented by the class `MomentsADPH2Fit`, that extends the abstract class `MomentsDiscPhaseFitter`.

2) Erlang-Coxian Distributions[5]: The next step in moment-matching techniques was given by Osogami and Harchol in a series of papers [23] [24] [5]. This extension consists on the characterization of the bounds imposed over the first three moments representable by a Phase-Type distribution withn n phases. They also introduce Erlang-Coxian distributions, named because they can be represented as the convolution of an Erlang and a Coxian distribution of second order. They present an algorithm to fit the parameters of a Erlang-Coxian distribution with or without mass at zero, an important issue in constructing matrix-geometric models from phase type distributions. An important issue is that the algorithm itself determines the number of phases needed to represent the set of moments, making easier the use of the algorithm since the user doesn't need to try with different configurations. The resulting distributions are not large in the number of phases but are not strictly minimal.

The implementation of the algorithms is given by two classes: `MomentsECCompleteFit` and `MomentsECPositiveFit`. The first one is built over the "complete solution" proposed by the authors, where the moment set is representable by the convolution of Erlang and Coxian distribution but the resulting distribution can have a positive mass on zero. To avoid this, the second class implements the "positive solution", where all the resulting distributions have no mass at zero, but the Erlang-Coxian distribution must be extended through a convolution or a convex mixture with a exponential distribution in order to obtain the strictly positiveness. Whenever the complete solution returns a positive distribution, this will be used by the `MomentsECPositiveFit`, an issue that forces this this class to depend on the `MomentsECCompleteFit` class.

3) Acyclic Continuous Distributions [6] : One of the most recent effort done in this area was made in 2005 by Bobbio, Horvath and Telek [6], who present an algorithm to match a set of first three moments with acyclic Phase-Type distributions(APH). They show the possible sets that can be represented by an acyclic distribution of order n. Then they show how to match the first three moments in a minimal way, i.e. using the minimal number of phases needed to do it. It is done by determining the region representable by an APH of n phases but not with an APH with $n-1$. This region is then partitioned in five areas that represent different distribution configurations, such as the Erlang-Exp structure that represents and $n-1$ Erlang distribution with an additional exponential phase after it.

The algorithm proposed by the authors for the positive case is implemented in the `MomentsACPHFit` class. There the algorithm begins with the first three non-central moments and computes the first two normalized moments. With this information, the required number of phases is computed and the moment set is evaluated in order to find in which region it falls. When it is determined, the parameters are fitted according to the equations presented by the authors.

V. Examples

In order to give a closer understanding of jPhase, some examples will be given to clarify the construction and manipulation of the computational objects. As shown in Figure 4, the distributions can be created from arrays of doubles, that represent the initial probability vector and the generator matrix of the transient states (as specified in section I). Once the distributions are created, they can be manipulated through the use of closure properties, as shown in Figure 4, where the convolution between the variables $v1$ and $v2$ is calculated.

```java
double[][] A = new double[][] { {-2,2} , {2,-5} } ;
double[] alpha = new double[] {0.2,0.4};
DenseContPhaseVar v1 = new DenseContPhaseVar(alpha , A);

double[][] B = new double[][] {
                    {-4,2,1} , {1,-3,1} , {2, 1,-5} } ;
double[] beta = new double[] {0.1, 0.2, 0.2};
DenseContPhaseVar v2 = new DenseContPhaseVar(beta , B);

ContPhaseVar v3 = v1.sum(v2);
System.out.println("v3: "+v3.toString());
```

Fig. 4. jPhase: Example 1

The resulting variable from the precious code has the usual representation, which includes the initial probability vector $\boldsymbol{\alpha}$ and the transition matrix $\mathbf{T}$, as explained in section I. The result from the former example is shown next, where the calculated variable is printed.

```
v3:
---------------------------------------------------
Phase-Type Distribution
Number of Phases: 5
Vector:
         0.2000    0.4000    0.0400    0.0800    0.0800
Matrix:
        -2.0000    2.0000    0.0000    0.0000    0.0000
         2.0000   -5.0000    0.3000    0.6000    0.6000
         0.0000    0.0000   -4.0000    2.0000    1.0000
         0.0000    0.0000    1.0000   -3.0000    1.0000
         0.0000    0.0000    2.0000    1.0000   -5.0000
---------------------------------------------------
```

Fig. 5. jPhase: Result for Example 1

Since jPhase is built over the Matrix Toolkit for Java (MTJ) library [25], it is also possible to construct Phase-Type distributions from matrices and vectors defined in that library. As can be seen in the following example, the matrix and the vector of the Phase-Type distribution are first built as `DenseMatrix` and `DenseVector` (MTJ objects), and then the continuous Phase-Type distribution is constructed.

```
DenseMatrix A = new DenseMatrix(
                    new double[][] { {-4,2,1} ,
                        {1,-3,1} , {2, 1,-5} } );
DenseVector alpha = new DenseVector(new double[]
                        {0.1, 0.2, 0.2});

DenseContPhaseVar v1 = new DenseContPhaseVar(alpha, A);

double rho = 0.5;
PhaseVar v2 = v1.waitingQ(rho);
System.out.println("v2:\n"+v2.toString());
```

Fig. 6. jPhase: Example 2

In the previous example, the distribution of the waiting time in queue is computed taking the variable $v1$ as the service time distribution and assuming that the traffic coefficient of the $M/PH/1$ queue is equal to 0.5. The resulting distribution is then printed and the output is shown next.

```
v2:
------------------------------------------------
Phase-Type Distribution
Number of Phases: 3
Vector:
        0.1500   0.2250   0.1250
Matrix:
        -3.8500   2.2250   1.1250
         1.1500  -2.7750   1.1250
         2.3000   1.4500  -4.7500
------------------------------------------------
```

Fig. 7. jPhase: Result for Example 2

Another way to do the former calculations is through the use of the Graphic User Interface (GUI). This can be used to build Phase-Type variables from direct input, or from a data set that can be fit the parameters of the distribution. It also allows to compute closure properties and has the capabilities to show graphically the probability density function or the cumulative probability distribution of a specified Phase-Type distribution. A sample screenshot of the developed GUI is shown in Figure 8.

Fig. 8. jPhase: Graphic User Interface

As can be seen, the developed framework is an easy way to deal with Phase-Type distributions and can be used as a supporting tool in several practical researches, where the main point is to build a probabilistic model that describes the system, and the Phase-Type distributions are an important tool to do it. Thus, the researcher can focus on the modeling issue based on the computational representation developed in this work.

VI. Conclusions

Phase-Type distribution has shown to be a powerful tool in computational probability since they can be used as input of Markov chains, which allows the use of efficient algorithms to compute measures of performance of real systems. In this work a computational framework has been designed and developed in order to allow the computational representation and manipulation of these distributions. The computational objects allows the user to concentrate in the modeling issues and not in the computation of distributions, moments or closure properties. In this way, the developed tool makes more accessible the of Phase-Type distribution for researches interested in stochastic modeling and performance evaluation of real systems.

The extensibility of the framework helps the user to develop new classes that can have a different representation (special sparse structures), but still exploiting the implemented methods in abstract classes. Even more, in the development of such extended classes the user can just implement some simple methods for the specific representation, or can develop procedures for the some or all the methods related to the distribution. In this way, the structure is not restricted to the developed methods, e.g. the researcher could use a different solver to compute the density function of a particular class of distributions.

The framework also includes a module for Phase-Type variates generation, which can be used to model large systems using simulation models with Phase-Type distributions. The tool has itself some procedures to do that, but the user could also develop a new algorithm and implement it with the help of the utilities methods and the unified framework.

Finally, the fitting module offers a set of recently developed algorithms to fit the parameters of a Phase-Type distribution from a data trace. It is possible to use general setting for the algorithms, without specifying any parameter. But the user can also determine specific characteristics, as the number of phases in the distribution, or the precision for convergence criterion. There is also a framework that can help to design the implementation of new algorithms, since the user have all the distribution classes as well as other algorithms to support his or her development.

References

[1] S. Asmussen, O. Nerman, and M. Olsson, "Fitting phase type distributions via the em algorithm," *Scandinavian Journal of Statistics*, vol. 23, pp. 419,441, 1996.

[2] R. Khayari, R. Sadre, and B. Haverkort, "Fitting world-wide web request traces with the em-algorithm," *Performance Evaluation*, vol. 52, pp. 175–191, 2003.

[3] A. Thümmler, P. Buchholz, and M. Telek, "A novel approach for fitting probability distributions to real trace data with the em algorithm," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[4] M. Telek and A. Heindl, "Matching moments for acyclic discrete and continuous phase-type distribution of second order." *I.J. of Simulation*, vol. 3, no. 3–4, pp. 47–57, 2002.

[5] T. Osogami and M. Harchol, "Closed form solutions for mapping general distributions to quasi-minimal ph distributions," *Performance Evaluation*, 2006, to appear.

[6] A. Bobbio, A. Horvath, and M. Telek, "Matching three moments with minimal acyclic phase type distributions," *Stochastic Models*, vol. 21, pp. 303–326, 2005.

[7] J. Muppala, R. Fricks, and T. K., "Techniques for system dependability evaluation," in *Computational Probability*, W. Grassmann, Ed. Kluwer Academic Publishers, 2000.

[8] M. F. Neuts, *Matrix-Geometrix Solutions in Stochastic Models*. The John Hopkings University Press., 1981.

[9] D. Cox, "A use of complex probabilities in the theory of stochastic processes," *Proceedings of the Cambridge Philosophical Society*, vol. 51, pp. 313–319, 1955.

[10] G. Riaño, *JMarkov user's guide and reference manual*, Industrial Engineering, Universidad de los Andes, 2005.

[11] M. Neuts and M. Pagano, "Generating random variates from a distribution of Phase-Type," in *Proceedings of the Winter Simulation Conference*, 1981.

[12] G. Latouche and V. Ramaswami, *Introduction to matrix analytic methods in stochastic modeling*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1999.

[13] G. Riaño, "Transient behavior of stochastic networks: application to production planning with load dependent lead times," Ph.D. dissertation, Georgia Institute of Technology, 2002.

[14] M. F. Neuts, "Two further closure properties of PH-distributions," *Asia-Pacific J. Oper. Res.*, vol. 9, no. 1, pp. 77–85, 1992.

[15] Y. Fang and I. Chlamtac, "Teletraffic analysis and mobility modeling for pcs networks," *IEEE Transactions on Communications*, vol. 47, no. 7, pp. 1062–1072, 1999.

[16] A. Law and D. Kelton, *Simulation, Modeling and Analysis*. McGraw-Hill Higher Education, 2000.

[17] A. Lang and J. Arthur, "Parameter approximation for phase-type distributions," in *Matrix Analytic methods in Stochastis Models*, S. Chakravarty, Ed. Marcel Dekker, Inc., 1996.

[18] T. Gonzalez, S. Sahni, M. Neuts, and W. Franta, "An efficient algorithm for the kolmogorov-smirnov and lilliefors tests," *ACM transactions on Mathematical Software*, vol. 3, no. 1, pp. 60–64, 1977.

[19] A. Dempster, N. Laird, and R. D.B., "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society. Series B*, vol. 39, pp. 1–38, 1977.

[20] C. Gourieroux and A. Monfort., *Statistics and Econometric Models*. Cambridge University Press, 1995, vol. 1, ch. 13 - Numerical Procedures, pp. 443–491.

[21] D. Aldous and L. Shepp, "The least variable phase-type distribution is erlang," *Stochastic Models*, vol. 3, pp. 467–473, 1987.

[22] A. Cumani, "On the canonical representation of homogeneous markov processes modeling failure-time distributions," *Microelectronics and Reliability*, vol. 22, no. 3, pp. 583–602, 1982.

[23] T. Osogami and M. Harchol, "Necessary and sufficient conditions for representing general distributions by coxians," in *Proceedings of the TOOLS 2003*, 2003.

[24] ——, "A closed form solution for mapping general distributions to minimal ph distributions," in *Proceedings of the TOOLS 2003*, 2003.

[25] B. Heimsund, "MTJ: Matrix toolkit for java," http://rs.cipr.uib.no/mtj.