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

Juan F. Pérez Germán Riaño
Centro de Optimización y Probabilidad Aplicada (COPA)
Departamento de Ingeniería Industrial
Universidad de los Andes, Bogotá D.C., Colombia
{fern-per, griano}@uniandes.edu.co

ABSTRACT

Phase-Type distributions are a powerful tool in stochastic modeling of real systems. In this paper, we describe an object-oriented tool used to represent and manipulate these distributions as computational objects. It allows the computation of multiple closure properties that can be used when modeling large systems with multiple interactions. The tool also includes procedures for fitting the parameter of a distribution from a data set and capabilities for generating random numbers from a specified distribution. This framework is built in a flexible and expandable way, and, therefore, it is not limited to the algorithms provided.

1. INTRODUCTION

Phase-Type (PH) distributions are a general class of probability distributions that generalize the well known exponential distribution through the composition of exponential phases. They were first introduced by Marcel Neuts [21].

The interest in PH distributions is threefold: they are dense, they allow to model many systems using Continuous Time Markov Chain (CTMC), even if the inter-arrival and processing times are not stochastic, and they have useful closure properties. We now comment on each of these items.

- First, it can be shown that PH distributions are *dense*, in the sense that the distribution of any continuous (non-negative) random variable can be approximated by a PH variable to any arbitrary precision (for a proof see, e.g., [21]). This implies that, for practical purposes, limiting the analysis of a particular system to PH distributions does not limit the applicability of the resulting model. However, the number of phases required could be prohibitively high for practical computations. Also, the aforementioned proof does not give light as how best to proceed when a PH representation is required for a particular data set, or a given distribution. There are, however, methods to find an approximating PH, and this problem has been the subject of recent research. Some of these methods are reviewed in Section 5.

- The second reason that interests us is that using PH variables you can model many systems as a CTMC, even if the distributions involved are not exponential. Consider, for example, a single-server queueing system with PH service times and Poisson arrivals. If the state definition keeps track of the current processing phase, the resulting model is a CTMC. Naturally this increases the number of states compared to the equivalent model with exponential distributions. Applications of PH distributions in manufacturing systems are described in [6] and references therein. They are also employed to model stochastic lead times in inventory models [34]. They are used in [27] to model Bucket Brigades systems.

- Finally, the PH variables have interesting closure properties under various operations like convolutions and mixtures.

Common examples of PH are the exponential, the Erlang, and the hyper-exponential distributions. For a review of these and other interesting PH properties see Neuts [21] and Latouche and Ramaswami [17].

In this paper, we present an object-oriented tool (called jPhase) to model PH distributions in a computational framework, allowing the representation and manipulation of these distributions as computational objects. This tool is part of a large project called jMarkov, whose aim is to provide an object-oriented based framework to facilitate modeling of large complex stochastic systems [28]. The developed structure allows a computational representation of PH distributions, the properties that they should have and functions to operate on them. For example, it has functions for the computation of the probability density or mass function, the cumulative distribution function, and moments, among others. It also admits the computation of closure properties, that can be used to model complex relations in stochastic stochastic systems. This issue was also included in SMART, a software package developed by Ciardo and his collaborators [7].

The main contribution of this work is that, using jPhase, any person with a basic knowledge in object-oriented programming can use PH distributions in the analysis of complex real-life systems where processing and inter-arrival times might not necessarily be exponential, and easily manipulate the distributions to generate new ones. In order to do that, it is important to find an easy way to go from data sets to parameters of PH distributions. One of the packages provided with the tool (called jPhaseFit) implements algorithms that fit the parameters of a PH distribution from a data set, and defines a flexible framework to add other algorithms. This tool also includes another package called jPhaseGenerator, which implements the algorithms developed by Neuts and Pagano [20] for generating discrete and continuous PH random variates, and defines the basic structure for any other that could be added in the future. In Figure 1 we show a simple example of what can be done with this tool. There two PH variables are created, the first one represents an exponential distribution and the other an Erlang distribution. Then a third one is created by computing the maximum between the first two variables and its cumulative distribution function is evaluated at 2.0. With the previous example we obtain that $P(v_3 \leq 2.0) = 0.7988$.

```
ContPhaseVar v1 = DenseContPhaseVar.expo(3);
ContPhaseVar v2 = DenseContPhaseVar.Erlang(1.5, 2);

ContPhaseVar v3 = v1.max(v2);

System.out.println("P(v3<=2.0):" +v3.cdf(2.0));
```

Figure 1: jPhase: Simple Example

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 distribution function.

This document is organized as follows: in Section 2, we describe PH distributions and some of its relevant properties. In Sections 3, 4, and 5, we describe the different modules that compose jPhase: the main, the fitter, and the generator, respectively. In Section 6, we give some illustrative usage examples. Finally, we give some some concluding remarks in Section 7.

2. PHASE-TYPE DISTRIBUTIONS

In this section, we review the definition and some properties of PH distributions. We follow the treatment presented in [21] and [17], and therefore, the proofs in this section are not included since the interested reader can find them in those books.

A continuous PH distribution is defined as the time until absorption in a CTMC, with one absorbing state and all others transient. The generator matrix of such process with $m+1$ states can be written as

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

where $\boldsymbol{A}$ is a square matrix of size m, $\boldsymbol{a}$ is a column vector of size m and $\boldsymbol{0}$ is a row vector of zeros. Here, the 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, $\boldsymbol{a}$ is determined by

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

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

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

where α_0 is the probability that the process starts in the absorbing state 0. Since the sum of all the components in the initials conditions vector must be equal to 1, α_0 is determined by

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

The distribution of a continuous PH variable X is, therefore, completely determined by the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{A}$ given above, and we say that X has a representation $(\boldsymbol{\alpha}, \boldsymbol{A})$. The cumulative distribution function (CDF) of X can be shown to be

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

Notice that this has a clear similarity to the well-known exponential distribution. In fact, if there is just one transient phase with associated rate λ and it is selected at time 0 with probability one, then the distribution is the exponential. From the previous expression, the probability density function (PDF) of the continuous part can be computed as

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

The Laplace-Stieltjes transform of $F(\cdot)$ is given by

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

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

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

A Discrete PH distribution can be seen as a discrete analogous case to the continuous PH distribution. In this case, the distribution is defined as the number of steps until absorption in a Discrete Time Markov Chain (DTMC), with one absorbing state and all others transient. The properties for this case can be found in [17].

As stated above, a relevant property of PH distributions is that they are closed under various operations, such as convolution, order statistics, convex mixtures, among other. For example, the mixture of two independent PH variables with representations $(\boldsymbol{\alpha}, \boldsymbol{A})$ and $(\boldsymbol{\beta}, \boldsymbol{B})$ which are chosen with probabilities p and $(1-p)$, respectively, has a PH representation $(\boldsymbol{\gamma}, \boldsymbol{C})$, where

$$\boldsymbol{\gamma} = [p\boldsymbol{\alpha}, (1-p)\boldsymbol{\beta}] \text{ and } \boldsymbol{C} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{B} \end{bmatrix}$$

Note that this is analogous to the construction of a hyper-exponential distribution.

These closure properties can be exploited in modeling some systems, as done, for example, in [26]. Continuous PH 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 PH process with PH failures [22].

3. OBJECT-ORIENTED FRAMEWORK

We will now describe the components of the jPhase modeling framework. To the extent of our knowledge, there is no academic or commercial software that can offer the capabilities of representation nor manipulation of PH distribution in a unified fashion. With our tool, it is possible to create an object that represents a continuous PH distribution, calculate the value of its PDF or its 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, like the distribution of the waiting time in a $M/PH/1$ queue.

We will now discuss some of the most important issues about the computational structure in order to give a good understanding of the framework.

The computational architecture is divided in three different packages: jPhase,jPhaseFit and jPhaseGenerator. The first implements the computational representation of PH distributions and its closure properties, and will be explained in this Section. The second offers an implementation of various PH fitting algorithms and will be explained in detail in Section 4. The last one implements PH random variates generators and will be discussed in Section 5. The first package can be seen as the core of the framework and the others are supported on it. Before proceeding, we will give a brief explanation of Java language, and object-oriented programming.

3.1 Java and Object-Oriented Programming

Java is a programming language by Sun Microsystems [33]. The main characteristics that Sun intended to have in Java are: object-oriented, robust, secure, architecture neutral, portable, high performance, interpreted, threaded and dynamic.

Object-Oriented Programming (OOP) is not a new idea. However, it did not have an increased development until recently. OOP is based on four key principles: abstraction, encapsulation, inheritance, and polymorphism. An excellent explanation of OOP and the Java programming language can be found in [33].

The abstraction capability is the one that interests us most. Java allows us to define abstract types like `PhaseVar`, and program some of its basic operations in a way that is completely independent of the internal representation used to store the matrices. We also define abstract functions like `min` and `max` that implement the respective closure properties for PH random variables.

Each object in Java belongs to a particular class that describes its different components (called *fields* in Java) and the functions and procedures (called *methods* in Java) that operate on them. A class can *extend* another class, and thus inherits fields and methods from the parent class. A class can also *implement* an *interface*. The interfaces determine the characteristics that a class should have, but have no implementation of any method.

3.2 General Structure

The `jPhase` package is supported on a set of interfaces, abstract classes, and implementing classes. As can be seen in the simple Class Diagram of Figure 2, there are three interfaces in the jPhase package: `PhaseVar`, `ContPhaseVar`, and `DiscPhaseVar`. These interfaces determine the behavior of PH distributions for both, the continuous and the discrete cases.

In the next level, the abstract classes `AbstractContPhaseVar` and `AbstractDiscPhaseVar` implement the corresponding interface (discrete or continuous), which develop some of the methods prescribed by the interfaces. Finally, the implementing classes extend the corresponding abstract class, and thus they make use of the already implemented methods. These methods are useful if the advanced user wants to develop her own implementing class, because she does not need to worry about the whole set of distribution properties, but only needs to implement a smaller set of methods. In the next sections, the properties of these interfaces, abstract and implementing classes will be explained.

3.3 Interfaces

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

- `PhaseVar`: This interface defines the set of properties that are common to both discrete and continuous PH distributions. Since this is the core interface in the framework, it has the major quantity of methods. The methods that the interface forces to implement for any distribution can be divided in three groups: access, moment,s and distribution methods.

- `DiscPhaseVar` and `ContPhaseVar`: These interfaces define some of the methods that correspond to the closure properties valid for discrete and continuous PH variables, as those discussed in section 2. The methods defined by each of these interfaces can be partitioned in two groups: distribution and closure methods. Some of the methods defined by these interfaces are shown in Table 2. Finally, Table 3 shows some methods that apply only for continuous PH distributions.

3.4 Abstract Classes

As shown in Figure 2, 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 class depends on the representation of the matrices and vectors involved. This means that new classes can be built on particular sparse matrix representations without losing the inherited methods from the abstract classes. This class computes the probability density function, using the uniformization method for computing the exponential matrix $e^{\boldsymbol{A}t}$ (see, e.g., [17]). In an analogous way, the abstract class `AbstractDiscPhaseVar` implements the interface `DiscPhaseVar`. Thus, it already has implemented the main methods imposed by the interface, including distribution related, moments, and some access methods.

Some methods depend on the particular representation given to the matrices. Our framework relies on the capabilities of the Matrix Toolkit for Java (MTJ) library [12] to represent dense and disperse matrices, and depending on this internal representation there are different classes as explained in the next subsection.

3.5 Implementing Classes

The developed implementing classes are those that a final user will usually manipulate. They have been designed as general PH representations for the continuous and discrete cases, and with dense and sparse storage. The `DenseContPhaseVar` and `DenseDiscPhaseVar` are classes that represent continuous and discrete PH distributions, using the

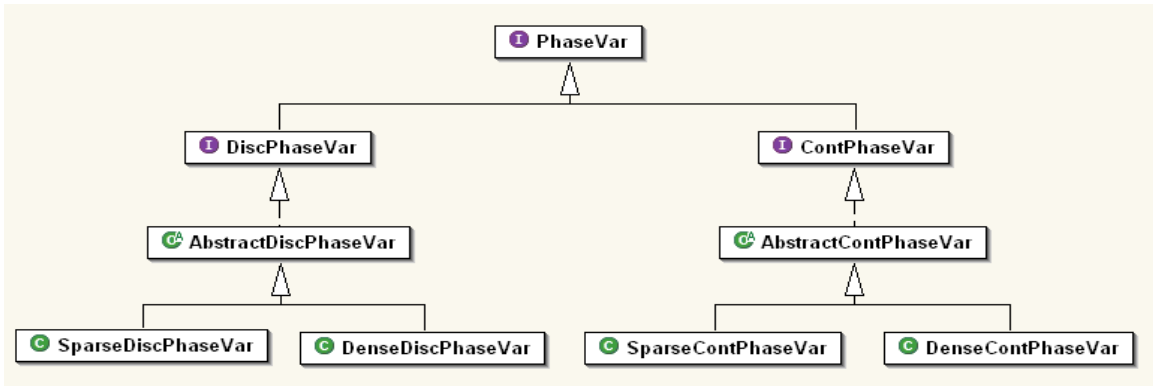

Figure 2: Class Diagram for jPhase Package

Table 1: Some methods for the `PhaseVar` interface

Type	Method	Result
Access	`getMatrix()`	Generator matrix $\boldsymbol{A}$
	`setMatrix(`A`)`	Set the value of the transition matrix equal to the parameter
	`getVector()`	Returns the initial probability distribution vector $\boldsymbol{\alpha}$
	`setVector(`α`)`	Set $\boldsymbol{\alpha}$ as the initial probability distribution
	`getNumPhases()`	Number of transient phases in the distribution
	`getVec0()`	Value of α_0
	`getMat0()`	Absorption rate vector $\mathbf{a} = -\mathbf{A1}$
	`copy()`	Deep copy of the distribution
Moments	`expectedValue()`	Expected value of the distribution
	`variance()`	Variance of the distribution
	`stdDeviation()`	Returns the standard deviation.
	`CV()`	The Squared coefficient of Variance.
	`moment(`k`)`	k-th non-central moment of the distribution.
Distribution	`cdf(`x`)`	CDF 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
	`quantile(`x`)`	Quantile x of the distribution
	`median()`	Median of the distribution

`DenseMatrix` and `DenseVector` classes defined by the Matrix Toolkit for Java (MTJ) library [12]. These 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 classes available in the MTJ package to represent sparse matrices and vectors.

4. FITTING MODULE

In the last twenty years, the problem of fitting the parameters of a PH distribution has received great attention from the applied probability community. There are different approaches that, as noted in [18], can be classified in two major groups: maximum likelihood methods and moment matching techniques. Nevertheless, almost all the algorithms designed for this task have an important characteristic in common: they reduce the set of distributions to be fitted from the whole PH set to a special subset.

We implemented the maximum likelihood algorithms by Asmussen et. al. [2], Khayari et. al. [16], and Thümmler et. al. [32], as well as the moment matching algorithms by Telek and Heindl [31], Osogami and Harchol [25], and Bobbio et. al. [5]. They were selected because they seem to be representative of the efforts done in both directions (maximum likelihood and moment matching) in the last ten years. Other algorithms can be found in references [3,4,13–15,29,30].

The jPhaseFit package defines the behavior of the classes that implement algorithms to fit the parameters of a PH 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 PH variable as the result of

Table 2: Methods for DiscPhaseVar and ContPhaseVar

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

Table 3: Closure methods for ContPhaseVar

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)	Waiting time distribution in a $M/PH/1$ queue with traffic coefficient equal to ρ

the fitting process. Note that each of the fitting algorithms receives different types of parameters. Therefore, when the user invokes `fit()`, those parameters should be specified at the constructor method of each implementing class .

4.1 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 specify the continuous or discrete nature of the variable to be fitted, as well as the procedures to compute the log-likelihood of the fitted distribution in relation to the data set.

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 separation is done for both the continuous and discrete cases. For the ML classes (`MLContPhaseFitter` and `MLDiscPhaseFitter`), there is an attribute called `logLH` that stores the log-likelihood value in order to make use of the usual computation of the log-likelihood in the fitting process. For the Moment-Matching related classes (`MomentsContPhaseFitter` and `MomentsDiscPhaseFitter`), the attributes `m1`, `m2`, and `m3` are defined, which correspond to the moments to be matched.

4.2 Maximum Likelihood Algorithms

The set of classes that implement maximum likelihood algorithms are almost all for continuous PH distributions, because most of the efforts have been done for these type of distributions. For each one of the following algorithms, there is an associated class that executes the procedures to fit the parameters of a PH distribution.

4.2.1 General PH Distribution EM Algorithm

In 1996 Asmussen, Nerman, and Olsson [2] presented a specialized version of the EM algorithm in order to fit the parameters of the whole set of continuous PH 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. [9] to deal with the problem of incomplete data (for a review, see [11]). The idea behind this algorithm is that a complete sample from PH 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 is easy to estimate the parameters of the distribution.

Nevertheless, the sample obtained from PH realizations is only the time that was required 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 vector and matrix, and the iterations include the computation of the likelihood (E-step) and its maximization to obtain a new set of parameters (M-step). In the case of PH fitting, 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. This algorithm does not compute the number of phases, and it must be entered as an initial parameter.

In this implementation, if the number of phases is not given, the program tries with distributions from 1 to 10 phases in order to find the one that shows the greatest log-

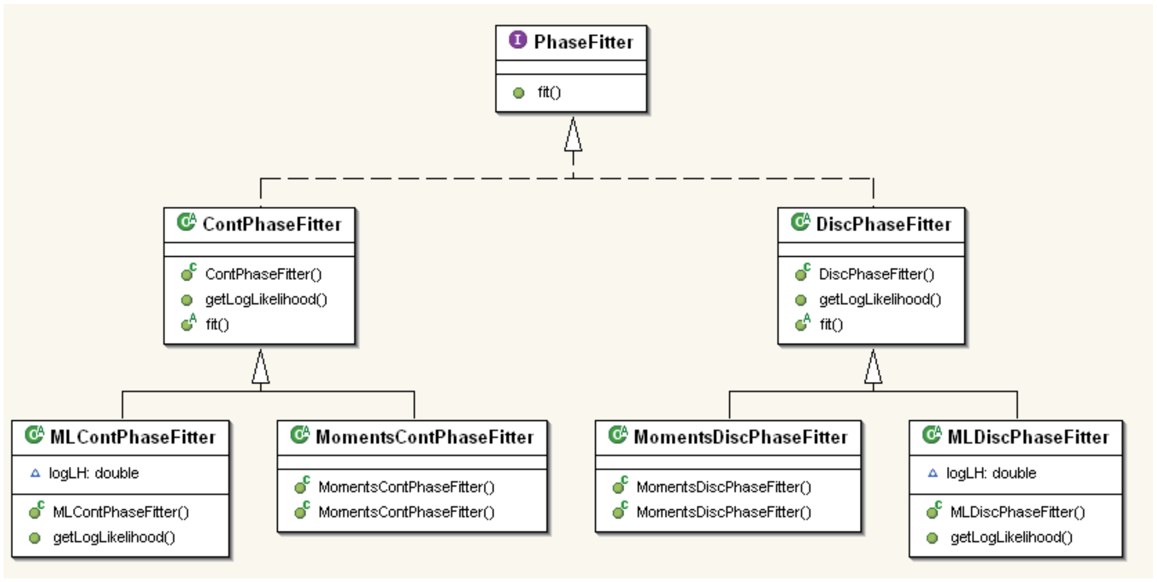

Figure 3: Class Diagram for jPhaseFit Package

likelihood. In every iteration, this method calls the E and M steps. The E-step uses an order-four Runge-Kutta procedure to solve the set of differential equations.

The user could also specify three important features for the algorithm performance: the precision for stopping the algorithm when the parameters show little change, the maximum number of iterations that the algorithm can execute; and the number of evaluation points for the Runge-Kutta method.

4.2.2 Hyper-Exponential Distribution EM Algorithm

The hyper-exponential distribution is a very special case of PH distributions, since the initial probability vector defines the probability of choosing the exponential phase to visit, and the generator matrix has diagonal representation with the rates of the i-th phase in the position (i, i). Thus, the number of parameters to fit a n-phase hyper-exponential distribution are $2n$. The algorithm proposed by Khayari et. al. [16] 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, which is taken as a given parameter.

In our implementation, if the number of phases is not given, several trials of configurations from one to ten phases are tried, and the distribution with greatest likelihood is selected. 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.

4.2.3 Hyper-Erlang Distribution EM Algorithm

In 2005, Thümmler et. al. [32] presented a method that fits the parameters of a hyper-Erlang distribution, which is a subset of the PH distributions that is 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 [2]. 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 method `fit()`makes a different work than just try distributions with one to ten phases. In the implementing class, a configuration is searched by means of the coefficient of variation of the data trace. When the coefficient is lower than one, then the program does not allow more than one branch since it has been shown that the PH variable with the least coefficient of variation is the n-Erlang($1/n$) [1]. When the coefficient of variation is greater than one, it enforces the creation of multiple branches as well as phases in each of them. For this method, the precision and number of iterations are relevant for its performance and the user can also specify them.

4.3 Moment Matching Algorithms

The distribution moments usually play an important role in the performance analysis of real systems [25]. 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 package, as will be explained in this section.

4.3.1 Acyclic Continuous Order-2 Distributions

In 2002, Telek and Heindl [31] proposed an algorithm to fit the parameters of an acyclic PH distributions with two phases. Acyclic distributions have been extensively studied since they have some important properties, as a canonic form developed by Cumani [8] and an 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).

In the implementing class, 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, that works in a similar fashion and is also included in the framework.

4.3.2 Erlang-Coxian Distributions

Osogami and Harchol in a series of papers [23–25] extended the previous method. This extension consists on the characterization of the bounds imposed over the first three moments representable by a PH distribution with 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 does not need to try different configurations. The resulting distributions are not large in the number of phases but are not strictly minimal.

The implementation of the algorithms is done in two classes: the first one implements 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 an exponential distribution in order to obtain the strictly positiveness.

4.3.3 Acyclic Continuous Distributions

One of the most recent effort done in this area was made in 2005 by Bobbio, Horvath, and Telek [5], who presented an algorithm to match a set of first three moments with acyclic PH distributions (APH). They described 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. This is done by determining the region representable by an APH with$n$ phases but not 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.

In the implementing class, 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.

5. RANDOM VARIATES MODULE

In many large applications, simulation is the appropriate tool to model the system because of the complex relations between different stochastic variables. Because of this, a random number generator might be a useful tool to model a wide range of non-deterministic systems. Neuts and Pagano [20] developed two similar algorithms to generate random variates from discrete and continuous PH distributions. These algorithms are supported on the alias method (see, e.g., [19]) 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.

The jPhaseGenerator package was developed to define the behavior of any PH random variates generator. This behavior is specified by the abstract class `PhaseGenerator`, which is the core of the package. As can be seen in Figure 4, this abstract class is extended by the classes `NeutsContPHGenerator` and `NeutsDiscPHGenerator`, that implement the algorithms proposed by Neuts and Pagano [20].

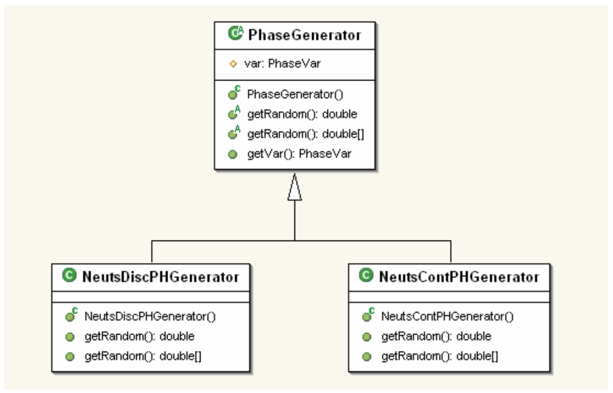

Figure 4: Simple jPhaseGenerator Package Class Diagram

5.1 Abstract Class

This abstract class defines the basic methods that a PH random variate generator should have. The class includes a `PhaseVar` attribute that represents the distribution from which the random variates will be generated. This attribute should be specified at the constructor level. This abstract class also forces the implementing subclasses to have procedures to return independent variates from the specified distribution.

5.2 Implementing Classes

Currently, two classes are provided with jPhase that extend the previously explained `PhaseGenerator` abstract class. These are `NeutsContPHGenerator` and `NeutsDiscPHGenerator`, which implement the method proposed by Neuts and Pagano [20]. 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 into a discrete one, using the well-known embedded chain. Thereafter, the main algorithm (for discrete distributions) can be used for both cases.

The algorithm strategy is to simulate the DTMC or CTMC associated with the corresponding PH distribution until absorption occurs. It first chooses 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; this process 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(n_i, λ_i) random variate must be generated. The sum of these variates over all the states is the value of the PH random variate.

Two important issues of this algorithm must be emphasized. The first one is the several uses of discrete distributions to generate the variates, which can be done efficiently through the alias method (see, e.g., [19]). 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^{-1}, that will be an Erlang variate with the required parameters [20].

The package `PhaseGenerator` also has an implementation of the polynomial-time algorithm proposed by Gonzalez et. al. [10] 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 PH distribution.

6. 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. 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 2), but there are also standard creators for distributions like exponential, Erlang, Cox, and hyper-exponential. Once the distributions are created, they can be manipulated through the use of closure properties. In Figure 5, the convolution between the distributions of two Erlangs with different parameters is calculated. Notice that we used a dense representation for them.

The resulting variable from the previous code has the usual representation, which includes the initial probability vector $\boldsymbol{\alpha}$ and the transition matrix $\boldsymbol{A}$, as explained in section 2. The result from the example is shown next, where the

```
ContPhaseVar v1 = DenseContPhaseVar.Erlang(0.8, 3);

ContPhaseVar v2 = DenseContPhaseVar.Erlang(1.5, 2);

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

Figure 5: jPhase: Example 1

representation of the calculated variable is printed. With a simple command, we could have also asked for the expected value or the variance of the obtained variable.

```
v3:
-----------------------------------------------
Phase-Type Distribution
Number of Phases: 5
Vector:
          1.0000   0.0000   0.0000   0.0000   0.0000
Matrix:
         -0.8000  0.8000   0.0000   0.0000   0.0000
          0.0000  -0.8000  0.8000   0.0000   0.0000
          0.0000  0.0000   -0.8000  0.8000   0.0000
          0.0000  0.0000   0.0000   -1.5000  1.5000
          0.0000  0.0000   0.0000   0.0000   -1.5000
-----------------------------------------------
```

Figure 6: jPhase: Result for Example 1

Since jPhase is built over MTJ [12], it is also possible to construct PH distributions from matrices and vectors defined in that library. As can be seen in the next example, the matrix and the vector of the PH distribution are first built as `DenseMatrix` and `DenseVector` (MTJ objects), and then the continuous PH 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.description());
```

Figure 7: 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.

Another way to do the former calculations is through the Graphic User Interface (GUI). This can be used to build PH variables from direct input, or from a data set to 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 PH distribution. A sample screen-shot of the developed GUI is shown in Figure 9.

As can be seen, the developed framework is an easy way to deal with PH 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 PH distributions are an important tool to do it. Thus,

```
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
--------------------------------------------------
```

Figure 8: jPhase: Result for Example 2

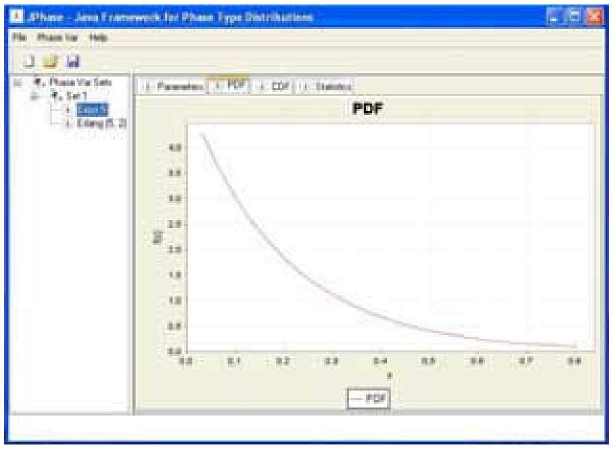

Figure 9: jPhase: Graphic User Interface

the researcher can focus on the modeling issue based on the computational representation developed in this work.

7. CONCLUSIONS

PH distributions have shown to be a powerful tool in computational probability since they can be used as input of Markov models, 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 allow 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 PH distribution for researches interested in stochastic modeling and performance evaluation of real systems.

The extensibility of the framework helps the advanced user to develop new classes that can have a different representation (special sparse structures), but still exploiting the implemented methods in abstract classes. Moreover, in the development of such extended classes, the interested user can just implement some simple methods for the specific representation, or can develop procedures for some or all the methods related to the distribution. In this way, the framework 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 fitting package offers a set of recently developed algorithms to fit the parameters of a PH distribution from a data trace. It is possible to use general settings 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 has all the distribution classes as well as other algorithms to support her development.

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

Acknowledgements

The class diagrams included in Figures 2, 3 and 4 were built with an academic license of Omondo software.

8. REFERENCES

[1] D. Aldous and L. Shepp. The least variable Phase-Type distribution is Erlang. *Stochastic Models*, 3:467–473, 1987.

[2] S. Asmussen, O. Nerman, and M. Olsson. Fitting Phase Type distributions via the EM algorithm. *Scandinavian Journal of Statistics*, 23:419,441, 1996.

[3] A. Bobbio and A. Cumani. ML estimation of the parameters of a PH distributions in triangular canonical form. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation*, pages 33–46. Elsevier Science Publishers, 1992.

[4] A. Bobbio, A. Horvath, M. Scarpa, and M. Telek. Acyclic discrete Phase Type distributions: properties and a parameter estimation algorithm. *Performance Evaluation*, 54:1–32, 2003.

[5] A. Bobbio, A. Horvath, and M. Telek. Matching three moments with minimal acyclic Phase Type distributions. *Stochastic Models*, 21:303–326, 2005.

[6] J. A. Buzacott and G. J. Shanthikumar. *Stochastic Models of Manufacturing Systems*. Prentice-Hall, 1993.

[7] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. SMART: Stochastic Model-checking Analyzer for Reliability and Timing. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, 2002.

[8] A. Cumani. On the canonical representation of homogeneous Markov processes modeling failure-time distributions. *Microelectronics and Reliability*, 22(3):583–602, 1982.

[9] 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*, 39:1–38, 1977.

[10] T. Gonzalez, S. Sahni, and W. Franta. An efficient algorithm for the Kolmogorov-Smirnov and Lilliefors tests. *ACM Transactions on Mathematical Software*, 3(1):60–64, 1977.

[11] C. Gourieroux and A. Monfort. *Statistics and Econometric Models*, volume 1, chapter 13 - Numerical Procedures, pages 443–491. Cambridge University Press, 1995.

[12] B. Heimsund. Matrix Toolkits for Java (MTJ). Webpage: `http://rs.cipr.uib.no/mtj/`, December 2005. Last checked on 05-Dec-2005.

[13] A. Horvath and M. Telek. Approximating heavy-tailed behaviour with Phase-Type distributions. In G. Latouche and P. Taylor, editors, *Advances in Algorithmic Methods for Stochastic Models*, pages 191–213. Notable Publications, Inc, 2000.

[14] M. A. Johnson and M. R. Taaffe. Matching moments to phase distributions: mixtures of Erlang distributions of common order. *Comm. Statist. Stochastic Models*, 5(4):711–743, 1989.

[15] M. A. Johnson and M. R. Taaffe. Matching moments to phase distributions: density function shapes. *Comm. Statist. Stochastic Models*, 6(2):283–306, 1990.

[16] R. Khayari, R. Sadre, and B. Haverkort. Fitting world-wide web request traces with the EM-algorithm. *Performance Evaluation*, 52:175–191, 2003.

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

[18] A. Lang and J. Arthur. Parameter approximation for Phase-Type distributions. In S. Chakravarty, editor, *Matrix Analytic methods in Stochastis Models*. Marcel Dekker, Inc., 1996.

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

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

[21] M. F. Neuts. *Matrix-geometrix solutions in stochastic models*. The John Hopkins University Press, 1981.

[22] M. F. Neuts. Two further closure properties of PH-distributions. *Asia-Pacific J. Oper. Res.*, 9(1):77–85, 1992.

[23] T. Osogami and M. Harchol. A closed form solution for mapping general distributions to minimal PH distributions. In *Proceedings of the TOOLS 2003*, 2003.

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

[25] T. Osogami and M. Harchol. Closed form solutions for mapping general distributions to quasi-minimal PH distributions. *Performance Evaluation*, 63:524–552, 2006.

[26] G. Riaño. *Transient behavior of stochastic networks: application to production planning with load dependent lead times*. PhD thesis, Georgia Institute of Technology, 2002.

[27] G. Riaño and J. P. Alvarado. Modeling Bucket Brigades with stochastic processing times. Working paper. Universidad de los Andes, 2006.

[28] G. Riaño and J. Góez. jMarkov: an object-oriented framework for modeling and analyzing Markov chains. working paper, Universidad de los Andes, 2006.

[29] A. Riska, V. Diev, and E. Smirni. Efficient fitting of long-tailed data sets into hyperexponential distributions. In *Proceedings of the IEEE Internet Performance Symposium (GlobeCom 2002)*, 2002.

[30] A. Riska, V. Diev, and E. Smirni. An EM-based technique for approximating long-tailed data sets with PH distributions. *Performance Evaluation*, 54:147–164, 2004.

[31] M. Telek and A. Heindl. Matching moments for acyclic discrete and continuous Phase-Type distributions of second order. *I.J. of Simulation*, 3(3–4):47–57, 2002.

[32] 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.

[33] P. van der Linden. *Just Java(TM) 2*. Prentice Hall, 6th edition, 2004.

[34] P. Zipkin. *Foundations of Inventory Management*. Mc Graw Hill, 2000.