jPhase User's Guide

Juan F. Pérez Germán Riaño

Contents

Introduction

Phase-Type (PH) distributions are a powerful tool in stochastic models of real systems. **jPhase** [19] is a Java-based framework that allows the representation of PH distributions through computational objects. The developed structure induces a formal representation of a Phase-type distribution and a set of properties that it should have. In this Manual, we illustrate the use of **jPhase** through several examples that are included in each section. This document is organized as follows: the first section gives an introductory background in PH distributions. Section 2 shows how the core module **jPhase** if structured and gives some examples. Sections 3 and 4 illustrate the structure of the packages **jPhaseFit** and **jPhaseGenerator**, and also give several examples of how to use the services provided by the package. More examples and applications for stochastic modeling with **jMarkov** can be found in the release that can be downloaded from `copa.uniandes.edu.co`.

1 Phase-Type Distributions

In this section, we review the definition and some properties of PH distributions. We follow the treatment presented in [14] and [11], and therefore, we do not include proofs in this section since the interested reader can find them in those books.

1.1 Continuous Phase-Type Distributions

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 equal 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, the distribution of a Continuous Phase-Type variable X is completely determined by the parameters $(\boldsymbol{\alpha}, A)$, and its cumulative distribution function (CDF) is

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

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 (PDF) of the continuous part 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

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

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

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

1.2 Discrete Phase-Type Distributions

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 transition probability matrix of that process may be defined as:

$$\mathbf{P} = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{a} & \mathbf{A} \end{bmatrix},$$

where the first row in the matrix represents the absorbing state. As the sum of the elements in every row of the matrix must equal one (in order to be a probability mass function), $\mathbf{a}$ is determined by:

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

Similarly, the initial probability distribution is defined as

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

where $\alpha_0 = 1 - \boldsymbol{\alpha}\mathbf{1}$ is the probability of starting the process in the absorbing state, i.e. the number of steps in that case would be equal to zero. As before, the distribution of a discrete Phase-Type variable X is completely determined by the parameters (α, A) and its probability mass function is defined as

$$P\{X = k\} = \begin{cases} \alpha_0 & , k = 0 \\ \boldsymbol{\alpha}\mathbf{A}^k\mathbf{a} & , k \geq 1 \end{cases}$$

This last definition makes natural the definition of the cumulative probability function of the discrete Phase-Type variable

$$P\{X \leq k\} = 1 - \boldsymbol{\alpha}\mathbf{A}^k\mathbf{1}, \quad k \geq 0.$$

Also, the generating function can be calculated as

$$\alpha_0 + z\boldsymbol{\alpha}(\mathbf{I} - z\mathbf{A})^{-1}\mathbf{a}, \quad |z| \leq 1.$$

from which, the factorial moments of the distribution can be computed

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

1.3 Closure Properties

An important issue of Phase-Type distributions is that they are closed under some operations, which can be useful in the analysis of some systems. The following closure properties are valid for both discrete and continuous distributions.

1. **Convolution of a finite number of Phase-Type distributions**
   If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$ and $Y \sim PH(\boldsymbol{\beta}, \mathbf{B})$ (independent of X), with n and m phases respectively, then the convolution is $PH(\boldsymbol{\gamma}, \mathbf{C})$ with $m+n$ phases and

   $$\boldsymbol{\gamma} = [\boldsymbol{\alpha}, \alpha_0\boldsymbol{\beta}] \qquad \text{and} \qquad \mathbf{C} = \begin{bmatrix} \mathbf{A} & \mathbf{a}\boldsymbol{\beta} \\ \mathbf{0} & \mathbf{B} \end{bmatrix}.$$

2. **Convex mixture of a finite number of Phase-Type distributions**
   If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$ and $Y \sim PH(\boldsymbol{\beta}, \mathbf{B})$ (independent of X), with n and m phases respectively, and distribution functions $F(\cdot)$ and $G(\cdot)$. Then, the convex mixture $\theta F(\cdot) + (1-\theta)G(\cdot)$, with $0 \leq \theta \leq 1$, has representation $PH(\boldsymbol{\gamma}, \mathbf{C})$ with $m+n$ phases, where

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

3. **Convolution of a discrete Phase-Type number of Phase-Type distributions**
   If X_i are i.i.d. continuous $PH(\boldsymbol{\alpha}, \mathbf{A})$ and N is a discrete $PH(\boldsymbol{\beta}, \mathbf{B})$, then $\sum_{k=0}^{N} X_i$ is $PH(\boldsymbol{\gamma}, \mathbf{C})$, with

   $$\boldsymbol{\gamma} = \boldsymbol{\alpha} \otimes \boldsymbol{\beta} \qquad \text{and} \qquad \mathbf{C} = \mathbf{A} \otimes \mathbf{I} + \mathbf{a}\boldsymbol{\alpha} \otimes \mathbf{B}.$$

   The function $\otimes$ denotes the Kronecker product and $\oplus$ the Kronecker sum. [1]

   As the geometric distribution is a particular case of Discrete Phase-Type distributions, this property also holds for the geometric case. If X_i are i.i.d. continuous $PH(\boldsymbol{\alpha}, \mathbf{A})$ and N follows a geometric distribution with parameter p, then $\sum_{k=0}^{N} X_i$ is $PH(\boldsymbol{\gamma}, \mathbf{C})$, with

   $$\boldsymbol{\gamma} = \boldsymbol{\alpha} \qquad \text{and} \qquad \mathbf{C} = \mathbf{A} + (1-p)\mathbf{a}\boldsymbol{\alpha}.$$

4. **The minimum of a set of Phase-Type distributions**
   If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$ and $Y \sim PH(\boldsymbol{\beta}, \mathbf{B})$ (independent of X), with n and m phases respectively, then $min(X, Y) \sim PH(\boldsymbol{\gamma}, \mathbf{C})$ with mn phases and

   $$\boldsymbol{\gamma} = \boldsymbol{\alpha} \otimes \boldsymbol{\beta}.$$

   In this case, the matrix C has a different definition if the process is discrete or continuous. In the discrete case, the resulting probability transition matrix is given by

   $$\mathbf{C} = \mathbf{A} \otimes \mathbf{B}.$$

[1] The Kronecker product of matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix}$$

And the Kronecker sum of matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as $\mathbf{A} \oplus \mathbf{B} = \mathbf{A} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{B}$.

For the continuous case, the generator matrix is given by

$$\mathbf{C} = \mathbf{A} \oplus \mathbf{B}.$$

5. **The maximum of a set of Phase-Type distributions**
If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$ and $Y \sim PH(\boldsymbol{\beta}, \mathbf{B})$ (independent of X), with n and m phases respectively, then $max(X, Y) \sim PH(\boldsymbol{\gamma}, \mathbf{C})$ with $mn + n + m$ phases and

$$\boldsymbol{\gamma} = [\boldsymbol{\alpha} \otimes \boldsymbol{\beta}, \beta_0 \boldsymbol{\alpha}, \alpha_0 \boldsymbol{\beta}]$$
$$\mathbf{C} = \begin{bmatrix} \mathbf{A} \oplus \mathbf{B} & \mathbf{I} \otimes \mathbf{b} & \mathbf{a} \otimes \mathbf{I} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{B} \end{bmatrix}.$$

1.4 Further Closure Properties for Continuous Distributions

There are some other important closure properties that only apply for the case of Continuous Phase-Type distributions, which are listed below.

1. **Waiting time in a $M/PH/1$ queue**
If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$ is the service time distribution in a $M/G/1$ queue, then the distribution of the waiting time $W(\cdot)$ is $PH(\boldsymbol{\gamma}, \mathbf{C})$, with

$$\boldsymbol{\gamma} = (1 - \rho)\boldsymbol{\pi} \qquad \text{and} \qquad \mathbf{C} = \mathbf{A} + \rho \mathbf{a}\boldsymbol{\pi},$$

where $\rho = \lambda m$ is the traffic coefficient, λ is the arrival rate and m is the expected value of the service time. $\boldsymbol{\pi}$ is the stationary probability vector of $\mathbf{A} + \mathbf{a}\boldsymbol{\alpha}$, i.e. $\boldsymbol{\pi} = (\boldsymbol{\alpha}\mathbf{A}^{-1}\mathbf{1})\boldsymbol{\alpha}\mathbf{A}^{-1}$.

2. **Residual time distribution**
If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$, then the residual time distribution

$$G(x) = \mathrm{P}(X - \tau \leq x | X > \tau)$$

has representation $PH(\boldsymbol{\gamma}, \mathbf{C})$, with

$$\boldsymbol{\gamma} = \frac{1}{1 - F(\tau)} \boldsymbol{\alpha} e^{\mathbf{A}\tau}.$$

3. **Equilibrium Residual time distribution**
If $X \sim PH(\boldsymbol{\alpha}, \mathbf{A})$, then the equilibrium residual time distribution

$$G(x) = \frac{1}{E[X]} \int_0^x (1 - F(u)) du,$$

has representation $PH(\boldsymbol{\pi}, \mathbf{A})$, where $\boldsymbol{\pi}$ has the same meaning as stated above.

4. **Termination time of a Phase-Type process with Phase-Type failures [15]**
Consider a process where the service time is determined by Phase-Type distribution with m phases and representation $PH(\boldsymbol{\alpha}, \mathbf{A})$, and it is subject to failures that occur according to a Poisson process with rate λ. If the duration of the failure is $PH(\boldsymbol{\beta}, \mathbf{B})$ with n phases, then the total completion time has distribution $G(\cdot)$ with representation $PH(\boldsymbol{\gamma}, \mathbf{C})$. Two different cases must be differentiated: if the service must be restarted after the failure, or if the task

can begin from the point where it was left before the failure. In the first case, the resulting distribution has $m + n$ phases and

$$\boldsymbol{\gamma} = [\boldsymbol{\alpha}, \mathbf{0}] \qquad \text{and} \qquad \mathbf{C} = \begin{bmatrix} \mathbf{A} - \mu\mathbf{I} & \mu\mathbf{1}\boldsymbol{\beta} \\ \mathbf{b}\boldsymbol{\alpha} & \mathbf{B} \end{bmatrix}.$$

In the second case, the representation has $m + mn$ phases and

$$\boldsymbol{\gamma} = [\boldsymbol{\alpha}, \mathbf{0}] \qquad \text{and} \qquad \mathbf{C} = \begin{bmatrix} \mathbf{A} - \mu\mathbf{I} & \mu\mathbf{I} \otimes \boldsymbol{\beta} \\ \mathbf{I} \otimes \mathbf{b} & \mathbf{I} \otimes \mathbf{B} \end{bmatrix}.$$

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 [6]). 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 $\mathbf{A}$.

1.5 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 [13] developed two similar algorithms to generate random variates from discrete and continuous Phase-Type distributions. These algorithms are supported on the alias method [12] 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.

1.6 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 [10]. 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 Phase-Type set to a special subset. In section 4, those algorithms included in the computational package will be revisited and further explained.

2 jPhase: the object-oriented Framework

2.1 General Structure

The `jPhase` package is supported on a set of interfaces, abstract classes and implementing 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 PH 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 implementing classes extends the corresponding abstract class, and thus they make use of the already implemented methods. These methods are useful for any user

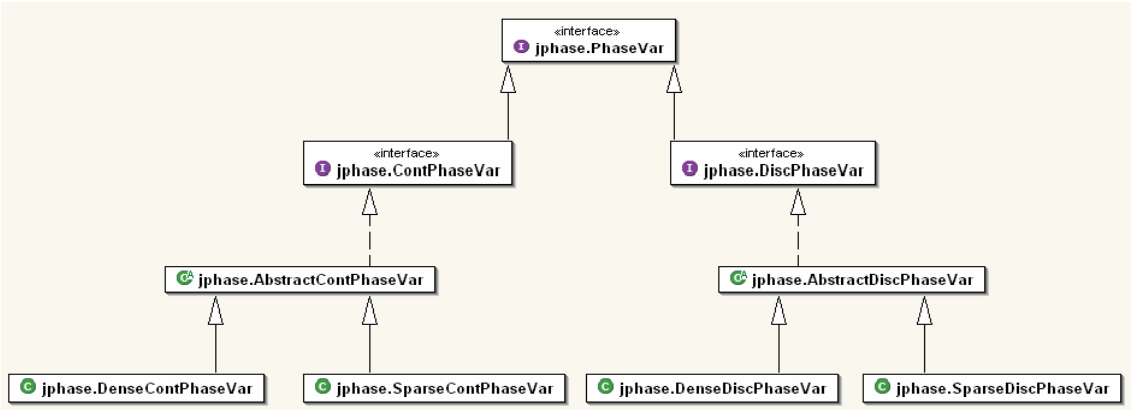

Figure 1: Simple **jPhase** Package Class Diagram

that wants to develop his own implementing class, because he does not 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 implementing classes will be explained.

2.2 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 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 PH variables, as those discussed in section 1. 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 PH distributions.

2.3 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 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[11]. 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 an implementing class are: `getMatrix` and `setMatrix`; `getVector` and `setVector`; `newVar` and `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. Since jPhase is built over the Matrix Toolkit for Java (MTJ) library [9], the user could develop his own representations over those offered by MTJ. Also the `newVar` and `copy` methods must return a variable that belongs to the same class of the original one. The implementing classes explained in the next section are themselves examples of classes that extend the abstract ones.

2.4 Implementing Classes

The developed implementing classes are those that a final user will utilize. They have been designed as general PH representations for the continuous and discrete cases, and with dense and sparse storage. The `DenseContPhaseVar` and `DenseDiscPhaseVar` classes represent continuous and discrete PH distributions, using the `DenseMatrix` and `DenseVector` classes defined by MTJ [9]. 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 `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. upper-diagonal matrices.

2.5 jPhase Examples

In this section, the use of **jPhase** is illustrated through relevant examples. All of these examples should be included in a class with a main class that calls them to be executed. In Figure 2, two PH variables: V_1 is a exponential variable with parameter $\lambda = 3$, and V_2 is an Erlang variable with parameters $\lambda = 1.5$ and $k = 2$. Once this is done, a third variable (V_3) is created as the maximum between V_1 and V_2. Finally, the probability that V_3 takes a value not greater than 2.0 is computed through the cumulative density function. This value is printed and is shown in Figure 3.

```
ContPhaseVar v1 = DenseContPhaseVar.expo(3);
ContPhaseVar v2 = DenseContPhaseVar.Erlang(1.5, 2);
ContPhaseVar v3 = v1.max(v2);
System.out.println("P(v3<=2.0):\t" +v3.cdf(2.0));
```

Figure 2: **jPhase**: Example 1

```
P(v3<=2.0):     0.7988666135682244
```

Figure 3: **jPhase**: Result for Example 1

```
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.toString());
```

Figure 4: **jPhase**: Example 2

```
v3:
-------------------------------------------------
Phase-Type Distribution
Number of Phases: 5
Vector:
        1.00     0.00     0.00     0.00     0.00
Matrix:
       -0.80     0.80     0.00     0.00     0.00
        0.00    -0.80     0.80     0.00     0.00
        0.00     0.00    -0.80     0.80     0.00
        0.00     0.00     0.00    -1.50     1.50
        0.00     0.00     0.00     0.00    -1.50
-------------------------------------------------
```

Figure 5: **jPhase**: Result for Example 2

Figure 4 shows again the creation of two particular PH variables: V_1 is an Erlang variable with parameters $\lambda = 0.8$ and $k = 3$, and V_2 is an Erlang variable with parameters $\lambda = 1.5$ and $k = 2$. Then, the variable V_3 is created as the sum of V_1 and V_2. This variable is printed, and the result is shown in Figure 5. The printed version of the variable includes the initial probability vector $\boldsymbol{\alpha}$ and the transition matrix $\mathbf{A}$, as explained in section 1.

```
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());
```

Figure 6: **jPhase**: Example 3

As shown in Figure 6, 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 1). Once the distributions are created, they can be manipulated through the use of closure properties, as shown in Figure 6, where the sum of the variables $v1$ and $v2$ is calculated. The resulting variable is shown next in Figure 7, where the calculated variable is printed.

Since **jPhase** is built over the Matrix Toolkit for Java (MTJ) library [9], it is also possible

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

Figure 7: **jPhase**: Result for Example 3

```
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());
```

Figure 8: **jPhase**: Example 4

```
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 9: **jPhase**: Result for Example 4

to construct PH distributions from matrices and vectors defined in that library. As can be seen in Figure 8, 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. In this example, the distribution of the waiting time in queue is computed taking the variable V_1 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 in Figure 9.

Another way to do the former calculations is through the use of the Graphic User Interface (GUI). This can be used to build PH 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 PH distribution.

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

3 jPhaseGenerator: the variates generator Module

This package was developed in order to define the behavior of any PH random variates generator. This behavior is specified by the abstract class `PhaseGenerator`, which is the core the package. As can be seen in Figure 10, this abstract class is extended by the implementing classes `NeutsContPHGenerator` and `NeutsDiscPHGenerator`. Those classes implement the algorithms proposed by Neuts and Pagano [13].

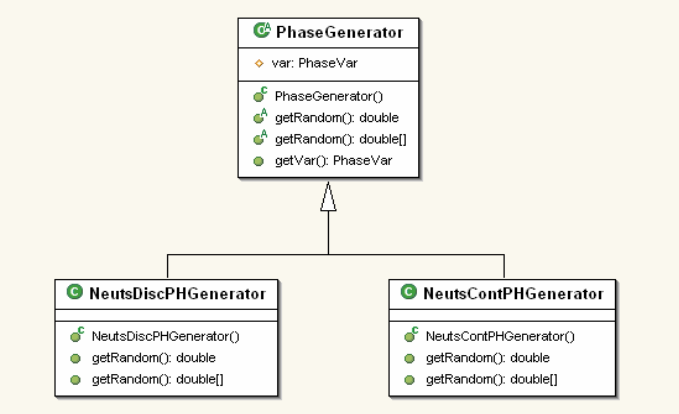

Figure 10: Simple jPhaseGenerator Package Class Diagram

3.1 `PhaseGenerator` Interface

This abstract class defines the basic methods that a PH 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 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 make use of 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 PH 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.

3.2 Implementing Classes

Currently, there are two implementing classes that extend the previously explained `PhaseGenerator` abstract class. These are `NeutsContPHGenerator` and `NeutsDiscPHGenerator`, which implement the methods proposed by Neuts and Pagano [13]. The first one implements the continuous case and the second the discrete one. The continuous algorithm has a first step, in which a discrete chain is biult over the continuous 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 PH random variate.

Two important issues of this algorithm must be emphasized. The first one is the use of discrete distributions to generate the variates, which can be done efficiently through the alias method [12]. 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 [13].

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 [12]. It also has an implementation of the polynomial-time algorithm proposed by Gonzalez et. al. [7] 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.

3.3 jPhaseGenerator Example

A simple example of the use **jPhaseGenerator** is shown in Figure 11. First, a new PH variable V_1 is created form its initial condition vector and the generator matrix. Then, a new generator (`gen`) is created using V_1 as parameter, since the variates to be generated must have this distribution. These variates are going to me stored in the `variates` array, that is declared. The next step is ask a set of ten independent random variates from the the generator. Finally, these variates are

printed and the result is shown in Figure 12.

```
DenseMatrix A = new DenseMatrix(new double[][] { {-4,2,1} ,
                                                 {1,-3,1} , {2, 1,-5} } );
DenseVector alpha = new DenseVector(new double[] {0.3, 0.3, 0.4});
DenseContPhaseVar v1 = new DenseContPhaseVar(alpha, A);

NeutsContPHGenerator gen = new NeutsContPHGenerator(v1);
double[] variates = new double[10];
variates = gen.getRandom(10);

for(int i = 0; i < 10; i++)
        System.out.println("var["+i+"]:"+variates[i]);
```

Figure 11: **jPhaseGenerator**: Example 1

```
var[0]: 0.8280372072645692
var[1]: 0.20203766144304366
var[2]: 0.8894691713002715
var[3]: 2.1955965497565346
var[4]: 0.21362619583588516
var[5]: 0.5985719384190227
var[6]: 0.05454947718195429
var[7]: 0.9746203239373876
var[8]: 2.1989174446014657
var[9]: 0.4107918940269194
```

Figure 12: **jPhaseGenerator**: Result for Example 1

4 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 PH distribution. As shown in Figure 13, 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 the fitting process.

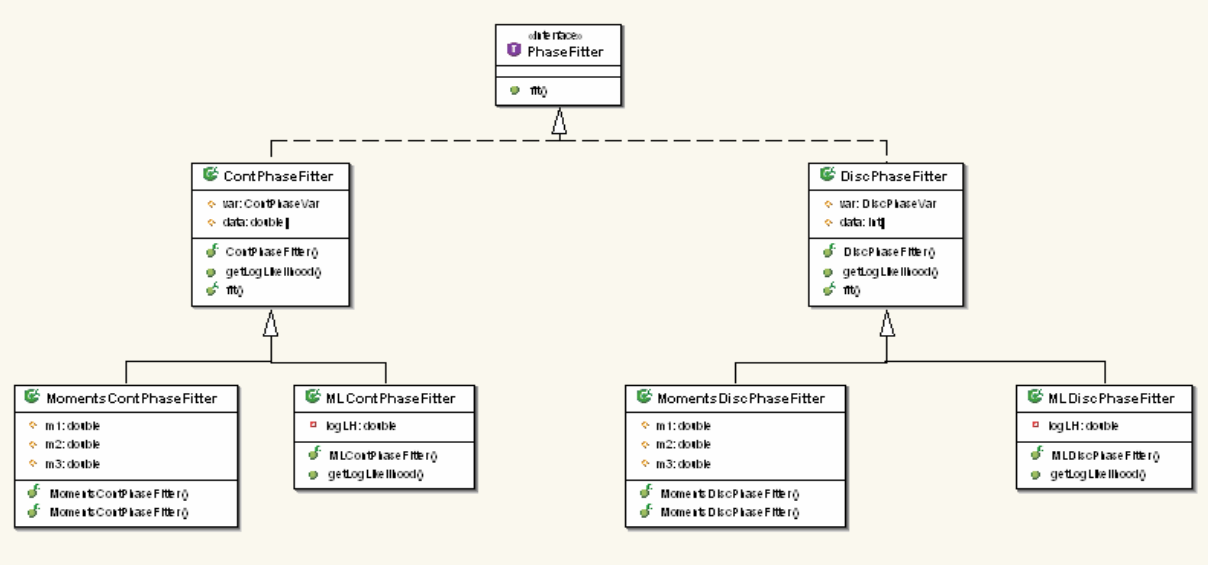

Figure 13: Simple **jPhaseFit** Package Class Diagram

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 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, these 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 be 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.

4.2 Maximum Likelihood Algorithms

The set of classes that implement maximum likelihood algorithms are almost all for Continuous PH 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 that executes the procedures to fit the parameters of a distribution.

4.2.1 General PH Distribution EM Algorithm

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 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. [4] to deal with the problem of incomplete data (for a review, see [8]). 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's easy to estimate the parameters of the distribution.

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 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. [5] 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.

4.2.3 Hyper-Erlang Distribution EM Algorithm

In 2005, Thümmler et. al. presented a method that fits the parameters of a hyper-Erlang distribution [21], which is a very interesting subset of the PH 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.

4.3 Moment Matching Algorithms

The distribution moments usually play an important role in the performance analysis of real systems [18]. 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.

4.3.1 Acyclic Continuous order-2 Distributions

In 2002 Telek and Heindl [20] proposed an algorithm to fit the parameters of an acyclic PH 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 [3] 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).

4.3.2 Erlang-Coxian Distributions

The next step in moment-matching techniques was given by Osogami and Harchol in a series of papers [17, 16, 18]. 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 doesn't need to try with different configurations. The resulting distributions are not large in the number of phases but are not strictly minimal.

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 [2], who present an algorithm to match a set of first three moments with acyclic PH 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.

4.4 jPhaseFit Examples

In order to illustrate the use of **jPhaseFit**, two examples are given in this section. The first one is shown in Figure 14, where the first step is the loading of a data set that comes from a text file. The

data set is stored in an array of doubles and it is used as the parameter to create the `fitter` object. This fitter is of the `EMHyperErlangfFit` type, that implements the method proposed in [21]. After the fitter initialization, a PH variable (V_1) is created as the result of fitting procedure applied over the data set, and the specification of the number of phases desired for the fitted distribution (4). Finally, the parameters of the fitted variable are printed, as well as the log-likelihood reached by the method in the last iteration. The result is shown in Figure 15.

```
double[] data = readTextFile("data/W2.txt");

EMHyperErlangFit fitter = new EMHyperErlangFit(data);

ContPhaseVar v1 = fitter.fit(4);

System.out.println("v1:\n"+v1.toString());

System.out.println("logLH:\t"+fitter.getLogLikelihood());
```

Figure 14: **jPhaseFit**: Example 1

```
v1:
-----------------------------------------------------
Phase-Type Distribution
Number of Phases: 4
Vector:
        0.220344        0.491673        0.205378        0.083606
Matrix:
        -0.152183       0.000000        0.000000        0.000000
        0.000000        -0.916394       0.000000        0.000000
        0.000000        0.000000        -9.177850       0.000000
        0.000000        0.000000        0.000000        -233.160991
-----------------------------------------------------

logLH:   -1180.4890525003095
```

Figure 15: **jPhaseFit**: Result for Example 1

The next example is shown in Figure 16. It shows the creation of a `fitter` of the `MomentsACPH-Fit` type, which implements the algorithm proposed in [2]. This `fitter` is created by specifying a set of moments that the fitted variable should have, in this case $m_1 = 2$, $m_2 = 6$ and $m_3 = 25$. As this information is enough for the method, it is not necessary to specify any other parameter and the variable V_1 is created as the fitted variable coming from running the method for the moments set. Finally, the variable is printed, as well as its moments, which naturally correspond to the asked ones. This result is shown in Figure 17.

```
MomentsACPHFit fitter = new MomentsACPHFit(2, 6, 25);

ContPhaseVar v1 = fitter.fit();

System.out.println("v1:\n"+v1.toString());

System.out.println("m1:\t"+v1.moment(1));

System.out.println("m2:\t"+v1.moment(2));

System.out.println("m3:\t"+v1.moment(3));
```

Figure 16: **jPhaseFit**: Example 2

```
v1:
-------------------------------------------------
Phase-Type Distribution
Number of Phases: 4
Vector:
        0.567595        0.432405        0.000000        0.000000
Matrix:
        -0.737313       0.737313        0.000000        0.000000
        0.000000        -2.438659       2.438659        0.000000
        0.000000        0.000000        -2.438659       2.438659
        0.000000        0.000000        0.000000        -2.438659
-------------------------------------------------

m1:     2.000000000000001
m2:     6.00000000000035
m3:     24.999999999983892
```

Figure 17: **jPhaseFit**: Result for Example 2

References

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

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

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

[4] 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 (1977), pp. 1–38.

[5] R. El Abdouni Khayari, R. Sadre, and B. R. Haverkort, *Fitting world-wide web request traces with the em-algorithm*, Performance Evaluation, 52 (2003), pp. 175–191.

[6] Y. Fang and I. Chlamtac, *Teletraffic analysis and mobility modeling for PCS networks*, IEEE Transactions on Communications, 47 (1999), pp. 1062–1072.

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

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

[9] B. Heimsund, *Matrix Toolkits for Java (MTJ)*, December 2005. Last modified: Monday, 05-Dec-2005 09:03:23 CET.

[10] 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.

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

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

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

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

[15] ——, *Two further closure properties of PH-distributions*, Asia-Pacific J. Oper. Res., 9 (1992), pp. 77–85.

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

[17] ——, *Necessary and sufficient conditions for representing general distributions by coxians*, in Proceedings of the TOOLS 2003, 2003.

[18] ——, *Closed form solutions for mapping general distributions to quasi-minimal PH distributions*, Performance Evaluation, 63 (2006), pp. 524–552.

[19] J. F. Pérez and G. Riaño, *jPhase: an object-oriented tool for modeling Phase-Type distributions*, in SMCtools '06: Proceedings from the 2006 Workshop on Tools for Solving Structured Markov Chains, New York, 2006, ACM Press.

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

[21] 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.