ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

# MEJORAS EN LOS PROYECTOS DE SIMULACIÓN DE FLUJO DE TRÁFICO EN TIEMPO ACELERADO.

# IMPROVEMENTS IN PROJECTS ABOUT ACCELERATED-TIME SIMULATION OF TRAFFIC FLOW.

Realizado por
**Ramírez López, Manuel**
Tutorizado por
**Aguilera Venegas, Gabriel**
**Galán García, José Luis**
Departamento
**Matemática Aplicada**

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Diciembre 2014

Fecha defensa:
El Secretario del Tribunal

**AGRADECIMIENTOS**

Gracias.

A Gabriel y José Luis. Agradecer su excelente forma de guiarnos en el proyecto y la confianza depositada en nosotros, pero sobre todo agradecer el trato recibido.

A mi familia y  amigos, en especial a Adrian y Abelardo. Se os echa mucho de menos.

A toda la clase trabajadora y  a todos los defensores de la educación pública de  este país, de vuestro sudor y vuestra valentía, hoy este hijo de obreros ha podido estudiar.

A mi madre. Siempre recuerdo cuando me obligabas a estudiar y decías que algún día te lo agradecería y yo decía que no. Pues gracias. Muchas gracias, por tu trabajo, dedicación, esfuerzo y todo lo enseñado.

A mi padre. Educar con el ejemplo no es una manera de educar, es la única. Y yo tengo el mejor ejemplo.

**Resumen:** Se han desarrollado cuatro mejoras en los proyectos de simulación de flujo de tráfico en tiempo acelerado. Los proyectos [1] y [2] realizan una simulación de flujo de tráfico en un CAS, Maxima, y usan Java, para realizar la GUI. Ambos usan Jacomax para realizar la comunicación Java-Maxima. La primera ha sido implementar un algoritmo Dijkstra difuso en [2] que simule (de forma más real que el algoritmo Dijkstra), el camino que sigue un vehículo entre un origen y un destino, dentro de un mapa (un grafo) que representa una zona de Málaga. Además, se ha personalizado el grafo inicial asociando uno ponderado a cada vehículo, en el cual, las aristas (las calles) tienen un peso calculado con una uniforme o una normal. Para ganar en rendimiento en [1] y [2], se ha permitido al usuario decidir cada cuantos pasos en Maxima se comunica con Java, eliminando así muchas comunicaciones que resultaban lentas. Además, se ha creado un programa con Java, el cual crea un paquete Maxima con las funciones de distribución, densidad, masa, variables aleatorias, que el usuario desee, dando la posibilidad de elegir entre las más usuales ya implementadas. Este paquete puede ser cargado en [1] y [2] permitiendo al usuario elegir la función de distribución que más se asemeje al fenómeno que se desea simular. La última ha sido conseguir que funcionen los proyectos [1] y [2] en una máquina Mac.

**Palabras claves:** Simulación Java Maxima CAS Jacomax Variables aleatorias Funciones de distribución Tiempo acelerado Algoritmo Dijkstra

**Abstract:** Four improvements projects were developed in the simulation of traffic flow in accelerated time. The projects [1] and [2] perform a simulation of traffic flow in a CAS, Maxima, and they use JAVA to make the GUI. Both use Jacomax for JAVA-Maxima communication. The first improvement has been the implementation of a diffuse Dijkstra's algorithm in [2] that simulates, more real way than the Dijkstra's algorithm, the path that follows a vehicle between an origin and a destination within a map (a graph) that represents an area of Malaga. In addition, it was customized the initial graph, associating one weighted for each vehicle, in which the edges (the streets) have a calculated weight with a uniform or normal distribution. On the other hand, in order to improve the performance in [1] and [2], the user is allowed to decide the number of steps between each Maxima communicates with Java, deleting so many communications that were slow. In addition, it was created a program with JAVA which creates a Maxima package with distribution functions, density, mass and random variables required por el usuario, giving the possibility to choose between the more usual already implemented. This package can be loaded in [1] and [2] allowing the user to choose the distribution function that most closely resembles the phenomenon that it was wanted to simulate. The last improvement has been getting the running of the projects [1] and [2] in a Mac machine.

**Keywords:** Simulation Java Maxima CAS Jacomax Aleatory variable Distribution function Accelerated-time Algorithm Dijkstra

# ÍNDICE


**1. INTRODUCCIÓN** .......... **11**

1.1. MOTIVACIÓN .......... 11
1.2. OBJETIVOS .......... 11
1.3. ESTRUCTURA DE LA MEMORIA .......... 12
1.4. TECNOLOGÍAS UTILIZADAS .......... 12
*1.4.1. JAVA .......... 13*
*1.4.2. MAXIMA .......... 13*
*1.4.3. JACOMAX .......... 13*
1.5. METODOLOGÍA .......... 14

**2. COMUNICAR JAVA CON MAXIMA** .......... **15**

**3. CONFIGURACIÓN EN OS X** .......... **17**

3.1. INTRODUCCIÓN Y MOTIVACIÓN .......... 17
3.2. TUTORIAL .......... 17

**4. ALGORITMO DIJKSTRA DIFUSO** .......... **21**

4.1. INTRODUCCIÓN Y MOTIVACIÓN .......... 21
4.2. ALGORITMO DIJKSTRA DIFUSO Y GRAFOS ASOCIADOS .......... 21
*4.2.1. ALGORITMO DIJKSTRA .......... 21*
4.2.1.1. DEFINICIÓN .......... 21
4.2.1.2. IMPLEMENTACIÓN .......... 22
4.2.2 CONVERTIR UN ALGORITMO DIJKSTRA EN UN DIJKSTRA DIFUSO .......... 23
*4.2.2.1 FALLO 1 .......... 23*
*4.2.2.2 FALLO 2 .......... 23*
*4.2.3 IMPLEMENTACIÓN .......... 23*
*4.2.4 EJEMPLO .......... 25*
*4.3 GRAFO ASOCIADO A UN VEHICULO .......... 26*
*4.3.2 OPCIÓN UNO: CON UNA UNIFORME .......... 26*
*4.3.3 OPCIÓN DOS: CON UNA NORMAL .......... 27*
*4.3.4 EJEMPLO .......... 27*
4.4 MANUAL DE USO .......... 27

**5 PAQUETE MAXIMA DE FUNCIONES DE DISTRIBUCIÓN** .......... **29**

5.2 INTRODUCCIÓN Y MOTIVACIÓN .......... 29
*5.2.2 MOTIVACIÓN .......... 29*
*5.2.3 OBJETIVO INICIAL Y RESULTADO FINAL .......... 29*
5.3 CONCEPTOS NECESARIOS .......... 30
*5.3.2 ESPACIO PROBABILÍSTICO .......... 30*
*5.3.3 VARIABLE ALEATORIA .......... 30*
5.3.3.1 CONCEPTO INTUITIVO .......... 30
5.3.3.2 DEFINICION FORMAL .......... 30
*5.3.4 DISTRIBUCIÓN DE PROBABILIDAD .......... 31*
*5.3.5 FUNCIÓN DE DISTRIBUCIÓN .......... 31*
*5.3.6 FUNCION DE DENSIDAD .......... 31*
*5.3.7 FUNCION DE MASA .......... 31*
*5.3.8 PROPIEDADES DE UNA FUNCIÓN DE DISTRIBUCIÓN .......... 31*
5.4 INTERFAZ DE USUARIO .......... 32
*5.4.2 VISTAS CREADAS Y MANUAL DE USO .......... 32*
*5.4.3 PROCESO PARA CREAR UNA VISTA .......... 37*
5.5 MODELO .......... 37

*5.5.2 CREAR UN PAQUETE ... 37*
*5.5.3 AÑADIR FUNCIONES ... 37*
5.5.3.1 PREDETERMINADAS ... 38
5.5.3.2 DISCRETAS ... 38
Puntos finitos: ... 38
Puntos infinitos: ... 39
5.5.3.3 CONTINUAS ... 39
5.6 CONTROLADOR ... 41
5.7 LIBRERÍA MAXIMA ... 41
*5.7.2 NOTACIÓN ... 41*
*5.7.3 FUNCIONES DEFINIDAS ... 41*
5.7.3.1 DISCRETAS ... 41
5.7.3.2 CONTINUAS ... 44
*5.7.4 ALGORITMOS ... 48*
5.6.3.1 CONTINUAS ... 48
5.6.3.2 DISCRETAS ... 49
*5.7.5 AUXILIARES ... 50*
5.8 PAQUETE MAXIMA PERSONALIZADO ... 51
*5.8.2 CONTENIDO ... 51*
*5.8.3 MANUAL PARA SU USO ... 51*
- Desde Maxima ... 51
- Desde Java ... 52
- Desde un proyecto de simulación usando Maxima ... 52

**6 NÚMERO DE PASOS QUE REALIZA MAXIMA ... 55**

6.1 INTRODUCCIÓN Y MOTIVACIÓN ... 55
6.2 IMPLEMENTACIÓN ... 56
*6.2.1 TRÁFICO DE MALETAS EN UN AEROPUERTO ... 56*
*6.2.2 TRÁFICO INTELIGENTE EN UNA ZONA DE MALAGA ... 57*
6.3 MANUAL DE USUARIO ... 57

**7 INTEGRACIÓN DEL PROGRAMA DEFINIDO EN 5, EN UN PROYECTO DE SIMULACIÓN DE FLUJO DE TRÁFICO EN TIEMPO ACELERADO ... 59**

**8 CONCLUSIONES ... 63**

8.1 OBJETIVO INICIAL Y RESULTADO FINAL ... 63
8.2 POSIBLES MEJORAS ... 63

**BIBLIOGRAFÍA ... 65**

# 1. INTRODUCCIÓN

## 1.1. MOTIVACIÓN

Podemos encontrar dos proyectos fin de carrera tutorizados por miembros del departamento de Matemática Aplicada de la UMA que realizan una simulación usando un CAS y mostrando dicha simulación con una GUI implementada en Java. Además de un trabajo fin de grado, con semejante estructura, que se está desarrollando simultáneamente a este proyecto. Los títulos de los cuales son:

- Simulación del recorrido del equipaje de un aeropuerto. [1]
- Simulación de tráfico con semáforos y señales inteligentes usando un CAS. [2]
- Simulación en tiempo acelerado de una red de autobuses metropolitana usando un CAS. [3]

Se detectaron cuatro mejoras a estos trabajos las cuales le dan mayor rendimiento, mayor portabilidad, un mayor realismo a la simulación (en el caso del trabajo que trata sobre el tráfico con semáforos y señales inteligentes) y se le ofrecerá la opción de realizar las simulaciones con las funciones de distribución que se desee, consiguiendo poder elegir las funciones que mejor se ajusten a la simulación.

## 1.2. OBJETIVOS

Los objetivos del proyecto son:

Para el PFC que simula el tráfico en una ciudad:

1) Con el fin de mejorar y conseguir una simulación más real, que se adapte de mejor forma a la verdadera ruta que seguiría un vehículo, se ha cambiado el algoritmo de Dijkstra por un algoritmo de Dijkstra difuso. Esto es debido a que en una situación real, el conductor no siempre conoce el camino óptimo. Además se cambiará el grafo asociado que dispone cada vehículo (grafo que se corresponde con el mapa, siendo los vértices cruces y las aristas calles), de esta manera cada vehículo no conocerá la distancia exactas de las calles sino una ponderación de estas, algo que se ajusta más a la realidad. Este objetivo se corresponde con el punto 4 de la memoria.

Para los dos PFC antes mencionados y el TFG en desarrollo [1], [2] y [3]:

2) Otra mejora que hemos realizado a los dos proyectos fin de carrera, es darle una mayor portabilidad, haciendo que funcionen en el sistema operativo Mac OS, en el que antes no funcionan. Punto 3 de la memoria.

3) Cuando se realiza una simulación, las funciones de distribución que van a modelar el comportamiento de la simulación son unas ya predefinidas, suelen ser las usuales en estadística (uniforme, normal, exponencial, Poisson,...), se ha creado un programa para poder utilizar en [1], [2] y [3] cualquier función de distribución. Este crea un paquete Maxima de funciones de distribución que también pueden ser usadas en otro programa Maxima o Java (usando Jacomax). Esto ayudará a que en posibles posteriores estudios sobre la simulación, se puedan utilizar funciones de distribución que se acerquen más a un tráfico real bien sea de maleta, de vehículos, de autobuses,… Podemos ver esta parte del proyecto en el punto 5.

4) Estos dos proyectos realizan una simulación con un CAS, en estos casos este es Maxima y proporcionan una interfaz de usuario en Java, calculando cada movimiento de la simulación en Maxima y enviando la información usando la librería (JACOMAX) a Java, esto puede resultar lento y muchas veces innecesario, para ello, así que, otra mejora es, poder decidir cuántos pasos queremos calcular en Maxima antes de comunicarnos con Java. De este modo, podemos eliminar la comunicación intermedia cuando esta no sea necesaria, aumentando así el rendimiento. Viene detallado en el punto 6 de la memoria.

## 1.3. ESTRUCTURA DE LA MEMORIA

Ya hemos visto una pequeña motivación y un breve resumen de los objetivos principales. Estos cuatro objetivos principales del proyecto serán los pilares fundamentales de la memoria. Cada uno de ellos tendrá una introducción y motivación propia. El otro punto en común de los cuatros objetivos será un manual de usuario. Los cuatro se compondrán de los apartados necesarios para su descripción detallada de funcionamiento y cómo han sido desarrollados. A continuación veremos los programas y tecnologías utilizadas y algunos detalles de estos, que pueden ser de interés en este proyecto. Veremos, también, la metodología de trabajo que hemos utilizado. Cómo instalar y configurar las tecnologías utilizadas y sobre todo cómo comunicarlas será otro punto. Este apartado lo veremos antes de comenzar a explicar los cuatro objetivos principales y será útil en cada uno de ellos. Antes de las conclusiones podremos ver un tutorial de cómo integrar el programa definido en el punto 5 en un proyecto de simulación, que en este caso es [3].

## 1.4. TECNOLOGÍAS UTILIZADAS

Son dos las principales que vamos a utilizar: Java y Maxima. Y una librería que cobra especial importancia ya que cumplirá la función de comunicar Java con Maxima. Veamos una breve descripción de cada una de ellas.

### 1.4.1. JAVA

Lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases, Java es uno de los lenguajes de programación más conocidos y utilizados. En nuestro caso será utilizado para crear la GUI. El entorno de desarrollo que utilizaremos parar programar en Java, principalmente, será Eclipse. Aunque también usaremos NetBeans por la facilidad que nos da para diseñar vistas simples de forma rápida y sencilla.

### 1.4.2. MAXIMA

Será la herramienta que realice la simulación. Maxima es un CAS, un sistema de algebra computacional, es decir, es un programa de ordenador o calculadora avanzada que facilita el cálculo simbólico. El objetivo de cualquier CAS es la realización de cálculos matemáticos, tanto simbólicos como numéricos, que es capaz de manipular expresiones algebraicas y matriciales, derivar e integrar funciones, realizar diversos tipos de gráficos, etc. Entre las ventajas que ofrece Maxima y lo diferencia del resto de CAS son:

- Es un software de código abierto.
- Programar en Maxima es más amigable que en la mayoría de CAS.
- Existen herramientas para la comunicación Java-Maxima.

### 1.4.3. JACOMAX

JACOMAX (Java connector for Maxima) es la librería de código abierto que hemos usado para realizar la comunicación entre Java y Maxima. La existencia de esta librería fue uno de los motivos principales para la elección de Maxima entre todos los CAS.

## 1.5. METODOLOGÍA

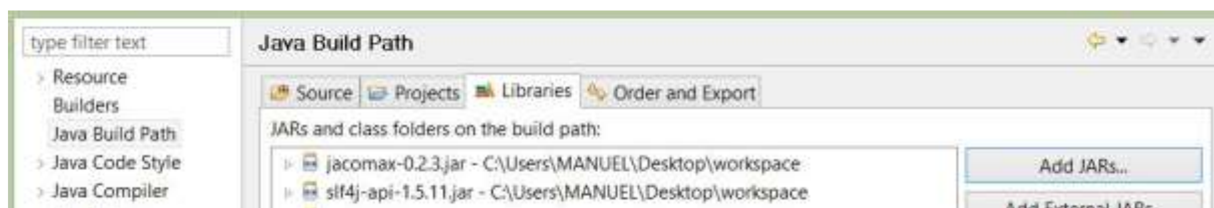
La metodología usada pare realizar el TFG ha sido prototipado simple o evolutiva. Tomando como los prototipos iniciales los PFCs ya hechos [1] y [2], a los que se les han añadido los objetivos ya descritos.

## 2. COMUNICAR JAVA CON MAXIMA

Dada la estructura que tienen los proyectos sobre los que vamos a trabajar, es importante no solo tener instalado Eclipse con Java y Maxima. Sino ser capaz de comunicarlos. Esto será lo que veamos a continuación.

Estos son los pasos que debemos seguir para comunicarlos en un ordenador con sistema operativo Windows. En el punto siguiente, veremos cómo se comunican en el sistema operativo OS X.

- En [7], nos podemos descargar la librería Jacomax.
- En el proyecto Java añadimos los .jar que nos acabamos de descargar. Estos son Jacomax-0.2.3.jar, slf4j-api-1.5.11.jar. Deben ser añadidos en Properties -> Java Build Path -> Libraries, tal y como se muestra en la imagen.




1. Ahora añadimos dentro de la carpeta del proyecto el archivo:“Jacomax.properties” el cual contiene las siguientes líneas:

```
Jacomax.Maxima.path=C\:\\Program Files (x86)\\Maxima-5.28.0-
2\\bin\\Maxima.bat

Jacomax.Maxima.charset=Cp1252
```

Nótese que el path debe de modificarse si se ha instalado Maxima en otra ubicación o se dispone de otra versión.

Para ejecutar una llamada a Maxima desde Java debemos de proceder a la siguiente forma:

2. Crear un String con la llamada. Por ejemplo:

```
String llamada = “2+2”;
```

3. Ejecutar la llamada a Maxima recibiendo la salida en otro String.

```
String salida = null;
```

```
MaximaConfigarion conf = JacomaxSimpleConfiguartor.configure();
MaximaProcessLauncher launcher = new MaximaProcessLauncher(conf);
MaximaInteractiveProcess process = launcher.launchInteractiveProcess();

try{
    salida = process.executeCall(llamada);
    process.terminate();

} catch (MatimaTimeoutException e){
    e.printStackTrace();
}
```

De esta forma, obtenemos en la cadena de salida la respuesta de Maxima. Es decir, hemos comunicado Java con Maxima realizando una llamada a Maxima y hemos recibido la respuesta de Maxima y almacenándola en una variable en Java. Para llamadas a Maxima más complejas deberemos de implementar métodos que interpreten la salida de Maxima.

# 3. CONFIGURACIÓN EN OS X

## 3.1. INTRODUCCIÓN Y MOTIVACIÓN

Se han realizado dos proyectos [1] y [2] en el departamento de Matemática Aplicada y se está realizando uno más [3], que siguen la misma filosofía: una simulación del tráfico de maletas en un aeropuerto, tráfico inteligente en una zona de Málaga y el tráfico una red de autobuses metropolitana en Maxima, mostrando el resultado en tiempo acelerado en Java y realizando la comunicación Java – Maxima usando la librería JACOMAX. Hasta ahora no se había conseguido el funcionamiento de estos proyectos en una máquina Mac. Con el fin de darle más portabilidad a estos proyectos y a sus futuras mejoras, se propuso como objetivo del proyecto su funcionamiento en OS X. A continuación, se verá un pequeño tutorial donde se ven las modificaciones necesarias para conseguir ejecutar los proyectos en el sistema operativo OS X.

## 3.2. TUTORIAL

En los dos proyectos anteriores [1] y [2], no se había conseguido su funcionamiento en un equipo Mac. Esto es debido, principalmente a cuatro factores:

1. Las direcciones en Mac son diferentes en formato a las direcciones en un PC Windows.
2. No es fácil conseguir compatibilidad entre las librerías que necesita el proyecto, Java y Maxima.
3. La configuración para establecer un MaximaInteractiveProcess (visto en el punto anterior) se realiza de forma diferente.
4. La salida que devuelve Maxima también es diferente a la salida que devuelve Windows así todos los métodos creados para procesar dicha salida deben de ser adaptados.

Veamos uno por uno los cuatro puntos en el proyecto [1] como ejemplo.

Estas son las direcciones que se han tenido que modificar para adaptar el proyecto a una máquina Mac:

- /Applications/f2.mac es donde se encuentra el código Maxima que se utiliza en el proyecto [1] para simular el tráfico de maletas en un aeropuerto.
- La dirección donde se encuentra Maxima no es necesario indicarla en este caso.

La versión de Maxima que he utilizado ha sido Maxima 5.30.0. La versión de eclipse ha sido Eclipse Helios y la versión de las librerías para la comunicación han sido Jacomax-samples-0.2.3.jar, Jacomax-0.2.3.jar y org.slf4.api_1.7.2.v20121108-1250.jar. Se puede afirmar que estas librerías, con la versiones de Eclipse y Maxima indicadas son compatibles sobre una máquina Mac con sistema operativo OS X con la versión 10.8.4. No es fácil que las versiones sean compatibles, ni se puede afirmar nada para nuevas versiones del OS X, ya que la librería Jacomax es un proyecto que no está siendo actualizado desde 2012.

El motivo por el que no es necesario indicar donde está ubicado Maxima es porque debemos instalar Maxima en /Applications y la configuración en Java pasa a realizarse de la siguiente manera:

```
MaximaConfigarion conf = JacomaxAutoConfigurator.configure();
MaximaProcessLauncher launcher = new MaximaProcessLauncher(conf);
MaximaInteractiveProcess process = launcher.launchInteractiveProcess();
```

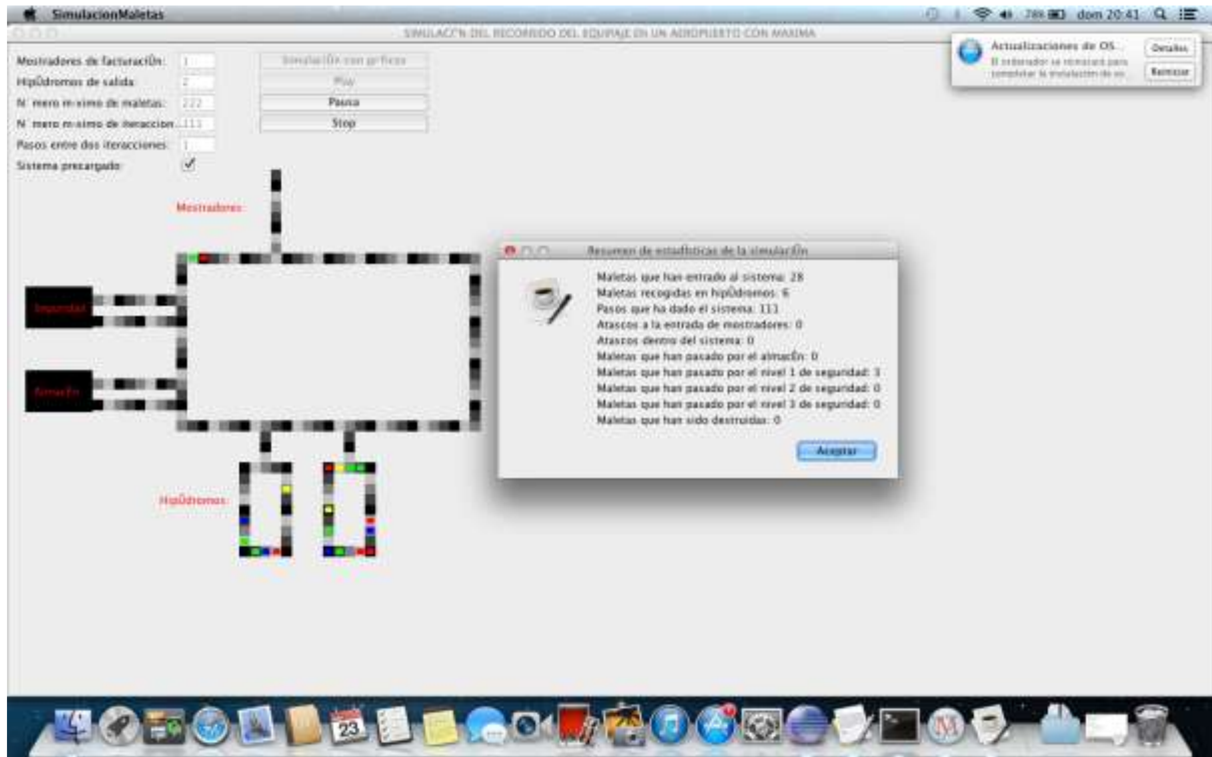
De este modo, no es necesario el archivo Jacomax.properties ya que la configuración se realiza automáticamente, y es por esto, que la localización de Maxima debe ser /Applications y no otra.

Para el correcto funcionamiento del proyecto se han de modificar los métodos MaximaTomatriz(), para interpretar correctamente la salida de Maxima, que es diferente en Windows y en Mac. Aquí se muestra un ejemplo de cómo deben de ser estos métodos:

```
public void MaximaToMatriz (String sal, int[][] m, int output){
	int pos=sal.lastIndexOf("(%o"+Integer.toString(output)+")");
	int pos2=sal.indexOf("(%i"+Integer.toString(output+1)+")",pos);
	String res=sal.substring(pos+6,pos2);
	res = res.trim();
	String[] items = res.replaceAll("\\[", "").replaceAll("E-4",
"").replaceAll("\\]", "").replaceAll(" ", "").replaceAll("\n",
"").split(",");
		int i= 0;
		for(int x=0;x<m.length;x++){
			for(int y=0;y<m[x].length;y++){
				if(i<items.length){
					m[x][y]=Integer.parseInt(items[i]);
					i++;
				}
			}
		}
```

```
            }
```

Realizando estas modificaciones el proyecto [1] se puede ejecutar correctamente en OS X.



# 4. ALGORITMO DIJKSTRA DIFUSO

## 4.1. INTRODUCCIÓN Y MOTIVACIÓN

En el proyecto [2], tenemos un mapa representado por un grafo dirigido en el cual, los cruces son los nodos o vértices y las calles son las aristas, las cuales están ponderadas según su longitud. La dirección de las aristas del grafo la determina la dirección de las calle del mapa, las cuales pueden cambiar.

Cada coche en el mapa tiene un nodo origen, al comienzo será su entrada y posteriormente (su posición) y un nodo destino (su salida). El camino entre el nodo origen y el nodo destino, lo determinamos usando el algoritmo Dijkstra o algoritmo de caminos mínimos. Este nos proporciona el camino más corto entre ambos nodos.

Pero sabemos que en el tráfico de una ciudad, los coches no siguen el camino más corto en sus desplazamientos. Es aquí, y con la finalidad de dotar de más realismo a la simulación, se ha implementado un algoritmo (Dijkstra difuso), el cual nos devuelve un camino pero no necesariamente el más corto. Además dependiendo de los parámetros que le indiquemos al algoritmo este devolverá caminos más o menos buenos. De esta forma, podemos ajustar los desplazamientos de los coches en la simulación a los desplazamientos que ocurren en un tráfico real.

En la misma dirección de conseguir más realismo en los desplazamientos, se ha añadido una nueva mejora. En una situación real, un conductor no conoce todas las distancias de las calles, de hecho, las distancias que si conoce son una estimación de la distancia real. Y bajo esta idea nace la siguiente mejora. Cada conductor tendrá una estimación sobre la distancia de las calles distinta, un grafo distinto, sobre el que se aplicará el algoritmo Dijkstra o Dijkstra difuso. La diferencia entre estos grafos será la ponderación de las aristas.

## 4.2. ALGORITMO DIJKSTRA DIFUSO Y GRAFOS ASOCIADOS

Para ver el comportamiento y la implementación del algoritmo de Dijkstra difuso, vamos primero ver el algoritmo Dijkstra, sobre el que se basa el nuestro.

### 4.2.1. ALGORITMO DIJKSTRA

#### 4.2.1.1. *DEFINICIÓN*

Algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista.

#### 4.2.1.2. *IMPLEMENTACIÓN*

```
public void encontrarRutaMinimaDijkstra(char inicio) {
          Queue<Nodo> cola = new PriorityQueue<Nodo>();
          Nodo ni = new Nodo(inicio);
          listos = new LinkedList<Nodo>();
          cola.add(ni);
          while (!cola.isEmpty()) {
               Nodo tmp = cola.poll();
               listos.add(tmp);
               int p = posicionNodo(tmp.id);
               for (int j = 0; j < grafo[p].length; j++) {
                    if (grafo[p][j] == 0)
                         continue;
                    if (estaTerminado(j))
                         continue;
               Nodo nod = new Nodo(nodos[j], tmp.distancia +
grafo[p][j], tmp);
                    if (!cola.contains(nod)) {
                         cola.add(nod);
                         continue;
                    }
                    for (Nodo x : cola) {
                         if (x.id == nod.id && x.distancia >
nod.distancia) {
                              cola.remove(x);
                              cola.add(nod);
                              break;
                         }
                    }
               }
          }
     }
```

### 4.2.2 CONVERTIR UN ALGORITMO DIJKSTRA EN UN DIJKSTRA DIFUSO

Sobre esta implementación hemos introducido dos fallos. Entendiendo como fallo que el algoritmo Dijkstra difuso, no toma la misma decisión que el algoritmo Dijkstra ocasionando que este no devuelva el camino más corto. Cada fallo se producirá con dos probabilidades. Sean param1, para el fallo 1 y param2, para el fallo 2.

#### 4.2.2.1 FALLO 1

Se produce entre "dos pasos del algoritmo", con probabilidad de error *param1* el algoritmo no elegirá la menor distancia para continuar, sino que de una lista ordenada con las opciones disponible: (list (lista de opciones con las que continuar)) y este elije con probabilidad 1-param1 la mejor opción y si no elije esta repite el proceso para el  resto de opciones. En el caso que solo quede una opción, elegimos esta.

#### 4.2.2.2 *FALLO* 2

Cuando el algoritmo encuentra un camino más corto entre dos nodos, este con probabilidad 1- *param2* elegirá la mejor opción y con probabilidad param2 la opción peor.

### 4.2.3 IMPLEMENTACIÓN

```
public void encontrarRutaDijkstraDifuso(char inicio, double param1,
                double param2) {
            List<Nodo> list = new ArrayList<Nodo>();
            Nodo ni = new Nodo(inicio);
            listos = new LinkedList<Nodo>();
            list.add(ni);
            while (!list.isEmpty()) {
                  Collections.sort(list);
                  Nodo tmp = null;
                  //Fallo 1
                  for (int i = 0; i < list.size(); i++) {
```

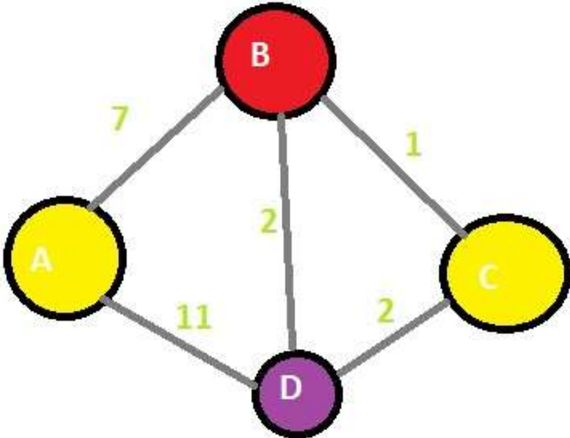
```
                    if (i == list.size() - 1) {
                          tmp = list.get(i);
                          list.remove(i);
                          break;
                    }
                    Random r = new Random();
                    double aux = r.nextDouble();
                    if (aux < 1.0 - param1) {
                          tmp = list.get(i);
                          list.remove(i);
                          break;
                    }
              }
              Collections.sort(list);
              listos.add(tmp);
              int p = posicionNodo(tmp.id);
              for (int j = 0; j < grafo[p].length; j++) {
                    if (grafo[p][j] == 0)
                          continue;
                    if (estaTerminado(j))
                          continue;
              Nodo nod = new Nodo(nodos[j], tmp.distancia +
grafo[p][j], tmp);
                    if (!list.contains(nod)) {
                          list.add(nod);
                          Collections.sort(list);
                          continue;
                    }
                    //Fallo 2
                    for (Nodo x : list) {
                          if (x.id == nod.id) {
                                if (x.distancia > nod.distancia) {
                                      Random r1 = new Random();
                                      double aux1 = r1.nextDouble();
                                      if (aux1 < 1.0 - param2) {
```

```
                                        list.remove(x);
                                        list.add(nod);
                                        Collections.sort(list);
                                    }
                                }
                                break;
                            }
                        }
                    }
                }
            }
```

Es claro que si al Dijkstra difuso le damos param1 y param2 con valor 0, este es un algoritmo Dijkstra.

### 4.2.4 EJEMPLO

Veamos el funcionamiento de estos dos algoritmos, con distintos parámetros para un ejemplo sencillo. Consideremos el siguiente grafo no dirigido (podría ser dirigido).




También es sencillo crear nuevos grafos para realizar distintas pruebas. En la siguiente tabla se muestra los resultados que nos devuelven los algoritmos Dijkstra y

Dijkstra difuso si le pasamos como nodo origen A y como nodo destino B. Para esta prueba el valor de param1 es 0.2 y de param2 es 0.3. En este caso vamos a realizar 9 pruebas.

| | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 |
|---|---|---|---|---|---|---|---|---|---|
| Dijkstra | abd | abd | abd | abd | abd | abd | abd | abd | abd |
| Dijkstra difuso | abcd | ad | ad | ad | abcd | abd | abd | ad | ad |
| Dijkstra difuso (fallo 1) | abcd | abcd | ad | ad | ad | abcd | abd | abd | abd |
| Dijkstra Difuso (fallo2) | abd | abcd | abd | abcd | abd | ad | ad | abcd | abd |

Podemos observar cómo todos los resultados son caminos posibles entre a y d. También podemos ver que el algoritmo Dijkstra nos devuelve la mejor opción en todas las pruebas. Y que el algoritmo difuso devuelve un camino pero no siempre el de menor coste. Con pruebas también podemos observar que si aumentamos la probabilidad de los fallos muy pocas veces vamos a obtener el mejor camino y más fácil será obtener un camino cada vez peor.

## 4.3 GRAFO ASOCIADO A UN VEHICULO

Si usamos esta opción, cuando el algoritmo Dijkstra o Dijkstra difuso determine el camino que siga un vehículo, este no usará la ponderación de las aristas reales. Las aristas estarán ponderadas realizando una estimación de la distancia. Y cada vehículo tendrá las aristas con una ponderación distinta.

Para ponderar las aristas, podemos usar dos opciones.

### 4.3.2 OPCIÓN UNO: CON UNA UNIFORME

Definida en Maxima: *uniformeG(a,b) := block( a + (b-a)*aleatorio(0.0,1.0))$*

uniformeG(a,b) genera el valor de una variable aleatoria uniforme entre a y b que será el nuevo valor de la arista. Donde, si v es el valor real y p el parámetro que le pasemos, el valor de la arista ponderada es uniformeG(v-p*v,v+p*v). p es un porcentaje y debe estar entre 0 y 1.

### 4.3.3 OPCIÓN DOS: CON UNA NORMAL

Definida en Maxima: *normalG(m,v) := block(m + (v^0.5)*(((-2*log(uniformeG(0.0,1.0)))^0.5)*cos(2*%pi*uniformeG(0.0,1.0))))$*

normalG genera un valor de la variable aleatoria normal con media m y varianza v, el cual será el nuevo valor de la arista.

La media será el valor real de la variable aleatoria. Y la varianza se pasará como parámetro. Es decir, el valor de la arista ponderada de esta forma es normalG(v,varianza). Donde la varianza se la debemos de dar nosotros. En ambas opciones, hemos impedido que se pueda ponderar el grafo con aristas de peso negativo. Para calcular los valores de las aristas y crear los grafos de cada uno de los vehículos, nos comunicaremos con Maxima usando Jacomax. Estas funciones están definidas en el archivo crearGrafico.mac

### 4.3.4 EJEMPLO

Veamos sobre el grafo anterior, dos grafos ponderados de estas dos formas.

| | Grafo real. | Grafo ponderado con una uniforme | Grafo ponderado con una normal |
|---|---|---|---|
| A-B | 7 | 7.788017224776772 | 6.413509606189065 |
| A-D | 11 | 8.274627643539191 | 11.85789735225634 |
| B-C | 1 | 0.98239052485162 | 1.968297182010995 |
| B-D | 2 | 2.827619199225792 | 3.16285530252506 |
| C-D | 2 | 1.390369235595578 | 0.60486633540265 |

Aquí es fácil ver que el camino más cor-to para el grafo real entre los nodos A y D es el ABD. Sin embargo, en el grafo ponderado con una uniforme (con parámetro 0.5), el camino más corto entre A y D pasa a ser AD. Y en el caso del grafo ponderado usando una normal (con varianza 1), el camino más corto es: ABCD.

## 4.4 MANUAL DE USO

Ejecutamos [2], y veremos la siguiente vista.

En ella, podemos introducir los dos parámetros necesarios para el algoritmo de Dijkstra difuso, además podemos decir si queremos ponderar los grafos asociados a cada vehículo y proporcionarle el parámetro necesario. El último dato que se pide en esta vista será explicado en el punto 6 de esta memoria.

# 5 PAQUETE MAXIMA DE FUNCIONES DE DISTRIBUCIÓN

## 5.2 INTRODUCCIÓN Y MOTIVACIÓN

### 5.2.2 MOTIVACIÓN

Cuando se quiere simular el comportamiento de una cola podemos usar una función de distribución exponencial para saber cada cuanto llegan elementos a la cola y si queremos saber cuántos elementos llegan usamos una Poisson. Una variable aleatoria normal puede estar asociada a caracteres morfológicos de individuos como la estatura, fisiológicos como el efecto de un fármaco, sociológico como el consumo de cierto producto, psicológico como el coeficiente intelectual,…

Son muchas las situaciones que se pueden simular con funciones de distribución ya conocidas y estudiadas, pero hay otras que se adaptarán a funciones de distribuciones específicas a la situación, si estas se ajustan mejor a la realidad.

Por lo que resulta útil e interesante tener una librería donde podamos encontrar un gran número de funciones de distribución conocidas, y que podamos añadir de forma fácil nuevas funciones de distribución. También que dada la función de distribución, en la librería se incluyan su función de densidad o de masa, una función que genere un valor o un vector de valores de la variable aleatoria dada por esa función de distribución.

### 5.2.3 OBJETIVO INICIAL Y RESULTADO FINAL

Se ha creado esta librería en Maxima. Se podrán crear librerías personalizadas las cuales contendrán esta librería y nuevas funciones que deseemos añadir. Gracias a esto los proyectos como [1], [2] y [3], se pueden simular con distintas funciones de distribución fácilmente. Así, conseguimos poder adaptarnos más a la realidad o probar distintas situaciones cambiando parámetros y funciones de distribución.

El objetivo al comenzar este proyecto era poder usar funciones de distribución cualesquiera en los proyectos [1] y [2]. Con esta librería e imponiendo la sintaxis de Maxima (al fin y al cabo debemos elegir una), hemos conseguido no solo que estos proyectos puedan tener una función dada por el usuario, sino poder comprobar que las funciones que se están usando son de distribución, dar la facilidad de elegir entre una lista de las funciones de distribución más utilizadas y poder cambiar entre el uso de una u otra librería fácilmente. Una vez creada la función de distribución por

el usuario, tendremos una función que genera valores de la variable aleatoria dada por dicha función de distribución. Y si esta es continua, también tendremos su función de densidad. Además, este paquete Maxima se puede usar en cualquier otro programa Maxima (como puede ser [3]), o ser llamado desde Java usando la librería Jacomax.

Así, se podrán crear de forma rápida y fácil un paquete con todas las funciones de distribución que necesitemos y poder usarla cómodamente.

## 5.3 CONCEPTOS NECESARIOS

Antes de ver cómo está implementado esta parte del proyecto, vamos a definir los conceptos matemáticos que vamos a utilizar.

### 5.3.2 ESPACIO PROBABILÍSTICO

Un espacio probabilístico asociado a un experimento es la terna $(\Omega, \mathcal{A}, \mathcal{P})$, donde $\Omega$ es su espacio muestral, $\mathcal{A}$ es la $\sigma$-álgebra de sucesos y $\mathcal{P}$ es una función de probabilidad definida sobre $\mathcal{A}$.

### 5.3.3 VARIABLE ALEATORIA

#### *5.3.3.1 CONCEPTO INTUITIVO*

Una variable aleatoria puede concebirse como un valor numérico que está afectado por el azar. Dada una variable aleatoria no es posible conocer con certeza el valor que tomará esta al ser medida o determinada, aunque sí se conoce que existe una distribución de probabilidad asociada al conjunto de valores posibles. Por ejemplo, en una epidemia de cólera, se sabe que una persona cualquiera puede enfermar o no (suceso), pero no se sabe cuál de los dos sucesos va a ocurrir. Solamente se puede decir que existe una probabilidad de que la persona enferme. Para trabajar de manera sólida con variables aleatorias, en general, es necesario considerar un gran número de experimentos aleatorios, para su tratamiento estadístico, cuantificar los resultados de modo que se asigne un número real a cada uno de los resultados posibles del experimento. De este modo, se establece una relación funcional entre elementos del espacio muestral asociado al experimento y números reales.

#### *5.3.3.2 DEFINICION FORMAL*

Una variable aleatoria sobre un espacio probabilístico $(\Omega, \mathcal{A}, \mathcal{P})$ es una función $\mathcal{X}: \Omega \rightarrow \mathbb{R}$ tal que $\mathcal{X}^{-1}(\mathcal{B}) \in \mathcal{A}$ para todo $B \in \mathcal{B}$.

### 5.3.4 DISTRIBUCIÓN DE PROBABILIDAD

La distribución de probabilidad de una variable aleatoria es una función que asigna a cada suceso definido sobre la variable aleatoria, la probabilidad de que dicho suceso ocurra. La distribución de probabilidad está definida sobre el conjunto de todos los sucesos, cada uno de los sucesos es el rango de valores de la variable aleatoria.

### 5.3.5 FUNCIÓN DE DISTRIBUCIÓN

La distribución de probabilidad está completamente especificada por la función de distribución, cuyo valor en cada *x* real es la probabilidad de que la variable aleatoria sea menor o igual que *x*, es decir, la función de distribución F de una función de probabilidad Ƥ definida en B(|R) es la que para cada real x, nos devuelve, F(x) = Ƥ([-inf,x]).

### 5.3.6 FUNCION DE DENSIDAD

La función de densidad de una variable aleatoria continua X permite trasladar la medida de probabilidad o "suerte" de realización de los sucesos de una experiencia aleatoria a la característica numérica que define la variable aleatoria.

Expresamos F, la función de distribución de probabilidad de X, en forma integral:

$$F(x) = p(X \leq x) = \int_{-\infty}^{x} f(x)dx$$ , cualquiera que sea el valor x,

donde $f : \mathbb{R} \to \mathbb{R}$ es una función no negativa e integrable.

### 5.3.7 FUNCION DE MASA

Es una función que asocia a cada punto de su espacio muestral X, la probabilidad de que ésta lo asuma. En concreto, si el espacio muestral E, de la variable aleatoria X consta de los puntos $x_1$, $x_2$,..., $x_k$, la función de probabilidad P asociada a X es:

$$P(x_i) = p_i ,$$

donde $p_i$ es la probabilidad del suceso $X = x_i$.

### 5.3.8 PROPIEDADES DE UNA FUNCIÓN DE DISTRIBUCIÓN

Estas propiedades las usaremos para comprobar que la función introducida por un usuario es una función de distribución. Son las siguientes:

1) $$0 \leq F(x) \leq 1$$

2) $$x_1 < x_2 \Rightarrow F(x_1) \leq F(x_2)$$

3) $$F(-\infty) = \lim_{x \to -\infty} F(x) = 0$$

4) $$F(+\infty) = \lim_{x \to +\infty} F(x) = 1$$

5) $$F(a^+) = \lim_{x \to a^+} F(x) = F(a)$$

6) $$\mathrm{P}(x = a) = F(a) - F(a^-)$$

7) $$\mathrm{P}(a < x \leq b) = F(b) - F(a) \qquad a, b \in \mathbb{R} \qquad a < b$$

## 5.4 INTERFAZ DE USUARIO

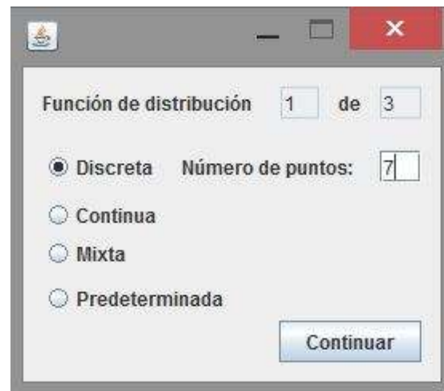
### 5.4.2 VISTAS CREADAS Y MANUAL DE USO

En la interfaz de usuario, se ha buscado que sea fácil e intuitivo crear una libraría de funciones de distribución nueva de Maxima. Se ha creado de forma que el usuario solo pueda realizar una acción o tomar una decisión por vista. Esta ha sido programada en Java usando el patrón de arquitectura modelo-vista-controlador.

Se han utilizado nueve vistas para este proceso. En la primera, llamada v1, podemos decidir el nombre de la librería y el número de funciones de distribución que vamos a añadir.

**Vista 1 o v1.**

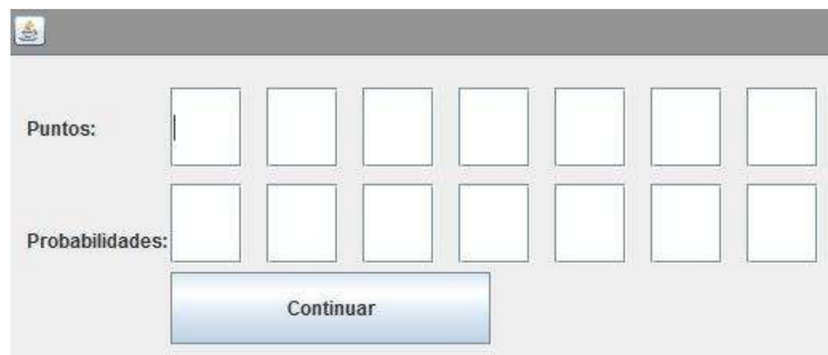
En la segunda vista, debemos decidir qué tipo de función de distribución queremos añadir entre tres opciones: discreta, continua o predeterminadas. En la parte de arriba tenemos una referencia para saber qué función de distribución estamos creando y cuantas tenemos que crear.




**Vista 2 o v2.**

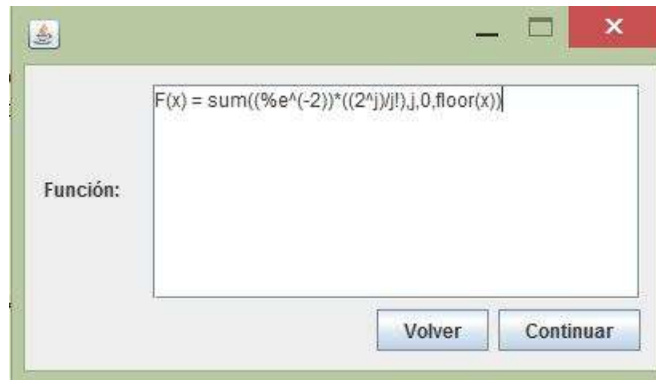
Si elegimos la opción Discreta, podemos elegir entre escribir la función de distribución con la sintaxis Maxima directamente, esta opción es obligatoria si es una discreta de infinitos puntos con probabilidad mayor que 0. Entonces debemos dejar los números de puntos en Inf (Infinito en sintaxis Maxima). Si, por otro lado, queremos que nuestra función discreta a añadir tenga un número finitos de puntos podemos indicarle dicho número y pasaremos a la vista 3.

Ahora debemos añadirles los puntos con probabilidad positiva en orden creciente y la probabilidad en cada punto. Si las probabilidades añadidas no suman 1, esta no será una función de distribución y así lo notificará el programa, aunque si dejará ser añadida bajo la responsabilidad del usuario.



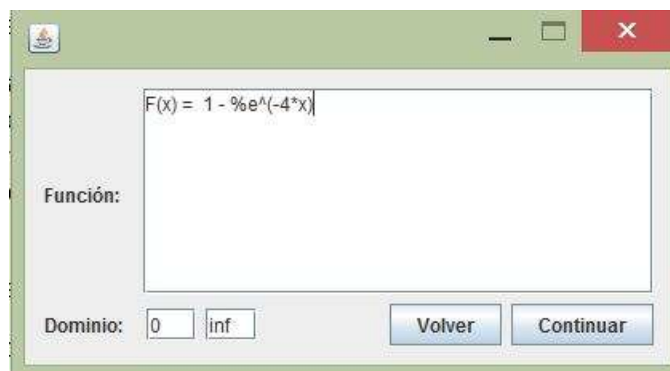

**Vista 3 o v3.**

Este es el caso que deseemos dar la función discreta con la sintaxis Maxima directamente, tan fácil como escribirla usando x como la variable.




**Vista 4 o v4.**

Volvamos a la vista 2, si pulsamos la opción Continuar. En este caso veremos la vista 5. En la cual, daremos en sintaxis Maxima la función de distribución donde la función es positiva. En la parte inferior, indicaremos el dominio de la función. Hemos visto que inf indica infinito en Maxima, y para indicar menos infinito en Maxima se usa minf.




**Vista 5 o v5.**

En el caso que deseemos usar una de las funciones ya definidas, entonces en la vista 2 elegimos la opción predeterminada. En esta vista, vemos todas las opciones que han sido implementadas. Así, dándole valores a los parámetros de una específica y asegurándonos tenerla seleccionada podemos pulsar en continuar. También podemos ver información sobre cada una de las funciones seleccionándola y pulsando en información (vista 6b).

DISCRETAS

Degenerada x0

Uniforme a b

Bernouille p

Rademacher

Binomial n p

Poisson lambda

Geometrica p

Binomial negativa r p

Hipergeometrica n m n1

CONTINUAS

Uniforme a b

Exponencial lambda

Normal media varianza

Lognormal media varianza

Weibul alpha beta

Gamma alpha beta

Beta alpha beta

Chi-Square k

Stundet's t k

F k1 k2

Z k1 k2

Pareto x0 alpha

Logistic alpha

Cauchy alpha beta

Irwin-Hall n

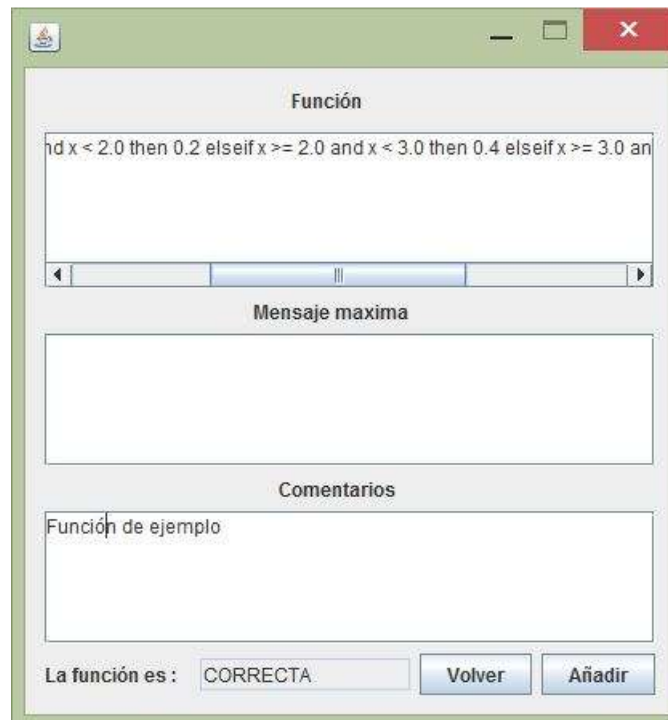
Información Volver Continuar

**Vista 6 o v6.**

Vista que aparece si seleccionamos una función en la vista anterior y pulsamos en información. Ésta sirve para saber la implementación de la función en Maxima y poder indicar el valor de los parámetros correctamente.

**Vista 6b o v6b.**

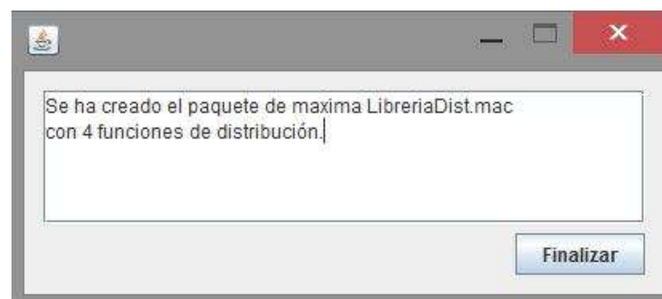
Una vez dada la función a añadir y pulsemos continuar iremos a la vista 7.




**Vista 7 o v7.**

En esta vista, vemos en sintaxis Maxima la función de distribución que vamos a añadir, además el mensaje Maxima que nos devuelve su algoritmo de comprobación. También nos permite añadir un comentario que será guardado en la librería.

Una vez añadida la función sea correcta o incorrecta la vista nos dirigirá a la vista 2 para añadir la siguiente función, salvo que está sea la última que iremos a la vista 8.



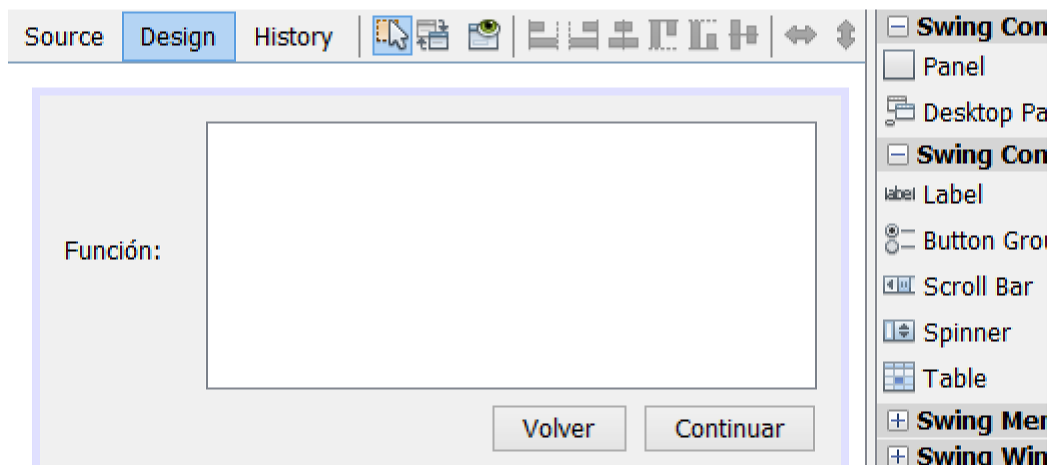

**Vista 8 o v8.**

Donde veremos un mensaje informativo y pulsando finalizar cerraremos el programa con la librería Maxima creada.

También existen los botones de volver en la mayoría de las vistas, el cual nos devolverla a la vista anterior.

### 5.4.3 PROCESO PARA CREAR UNA VISTA

Para crear las vistas anteriormente descritas, se ha usado la biblioteca de interfaces gráficas de usuario para Java, Swing, la cual está incluida en el entorno de desarrollo de Java (JDK). La cual extiende a la librería AWT. Muchas de estas vistas, por simplicidad, han sido creadas usando la opción gráfica que nos proporciona NetBeans.




La vista 3 se crea dinámicamente, dependiendo de los puntos con probabilidad que tenga la función de distribución discreta. Por ella esta ha sido implementada directamente con código.

## 5.5 MODELO

Como es de esperar, aquí tendremos la lógica de negocio. Encontraremos los métodos para crear el paquete de Maxima y añadirle cada una de las funciones que el usuario vaya introduciendo.

### 5.5.2 CREAR UN PAQUETE

La creación de un paquete Maxima es tan simple como crear un fichero de texto con el nombre dado por el usuario y con extensión ".mac". La dirección donde se crearan los paquetes será c://paquetesMaxima.

### 5.5.3 AÑADIR FUNCIONES

Vamos a dividir esta parte atendiendo a la naturaleza de la función de distribución a añadir.

#### 5.5.3.1 PREDETERMINADAS

Estas ya están implementadas en la librería y basta con escribirlas pasándoles los parámetros que el usuario ha decidido darles. Así que tan solo se debe escribir en el paquete, teniendo cuidado si es una función de ningún, uno, dos o tres parámetros. Un ejemplo de las líneas que se escribirían en maxima sería:

```
fDist1(x) := cauchyDist(x,2,2)$
fDens1(x) := cauchyDens(x,2,2)$
fG1() := cauchyG(2,2)$
vector_fG1(n) := vector_cauchyG(2,2,n)$
```

Nótese que estas funciones no necesitan comprobación.

#### 5.5.3.2 DISCRETAS

Una vez más, dividimos las funciones de distribución discretas en dos casos:

Puntos finitos: Si introducimos la función de esta forma tendremos dos arrays uno de los puntos con probabilidad y otro de la probabilidad de cada uno de los puntos. Es fácil crear la función de distribución y de masa dados estos valores. Sería de la siguiente forma:

```
public String discreta(List<Double> puntos, List<Double> prob) {
		String resultado = "";
		double suma = 0;
		for (int i = 0; i < prob.size(); i++) {
			suma = suma + prob.get(i);
		}
		if (suma != 1.0) {
			return "error";
		} else {
			suma = 0;
			resultado = "f(x) := if x < " + puntos.get(0) + " then 0 ";
			for (int j = 1; j < puntos.size(); j++) {
				suma = suma + prob.get(j - 1);
				resultado += "elseif x >= " + puntos.get(j - 1) + " and 
x < "
						+ puntos.get(j) + " then " + suma + " ";
			}
			resultado += "else 1;";
			return resultado;
		}
```

Para crear la función que genere valores de la variable aleatoria llamaremos a un algoritmo de la librería creado para este fin. En la sección algoritmos de la sección Maxima, podéis ver la implementación así cómo su explicación.

La comprobación de que los datos son correctos es simple, basta con sumar las probabilidades y comprobar que suman 1. Lo podemos hacer en Java, sin necesidad de comunicarnos con Maxima. El resto de propiedades se tendrán por construcción.

Puntos infinitos: para introducir esta función de distribución el usuario debe pasárnosla en sintaxis Maxima y así la escribiremos en el paquete. En esta parte es interesante ver como se comprueba que realmente sea una función de distribución. Para ello conviene recordar las propiedades suficientes y necesarias que cumple una función de distribución, las cuales ya han sido definidas. Para comprobar estas propiedades aprovecharemos las posibilidades que nos ofrece Maxima. La comunicación en Maxima es explicada en el apartado 2 de la memoria;

Asín, comprobamos:

Que la función es creciente lo podemos ver de la siguiente forma:

```
assume(a <= b); is (f(a) <= f(b));
```

Que tiene límite 1 cuando x tiende a infinito:

```
is (limit(f(x),x,inf) = 1);
```

Que tiene límite 0 cuando x tiende a menos infinito:

```
is (limit(f(x),x,minf) = 0);
```

Que el límite derecho en un punto c es el valor de la función en c:

```
is (limit(f(x),x,x,plus) = f(c));
```

Muchas veces, Maxima no sabe resolver lo que le preguntamos devolviéndonos un mensaje “unknow”. Esto puedo ocurrir cuando Maxima no puede comprobar si una función es creciente o calcular su límite. En la vista 7 en la parte de “Mensaje de Maxima”, podemos ver la respuesta de Maxima a nuestras preguntas de comprobación. De este modo, aunque nuestro algoritmo de comprobación devolviera que la función es incorrecta, será el usuario quien tenga la última palabra sobre si la función se debe o no añadir a la librería.

#### *5.5.3.3 CONTINUAS*

En este caso, cuando el usuario desea añadir una función de distribución continua, éste introducirá en sintaxis Maxima la función y además nos dará su dominio. Y la función de introducción que nosotros añadiremos a la librería será:

```
"fDist" + num + "(x) := block (" + aux0 + ")$";
```

La función de densidad será:

```
"fDen" + num + "(x) := block ( diff(" + aux0 + " , x))$"
```

La que genera valores de la variable aleatoria continua es:

```
"fG" + num + "() := block ( funcionContinuaG(" + aux0 + ","+ ini + "))$";
```

Y si lo que deseamos es un vector de n componentes:

```
"vector_fG" + num + "(n) := block ( vector_funcionContinuaG(" + aux0 + "," + ini
+ ",n))$";
```

Siendo num, el número de función que estamos añadidos y la cadena aux0 es la función que hemos añadido con la vista 5. Y funcionContinuaG y vector_funcionContinuaG los algoritmos implementados en Maxima y explicados en el punto 5.6.3 de la memoria.

Para comprobar que lo que el usuario nos ha introducido es una función de distribución, podemos usar el mismo algoritmo genérico que usamos con las funciones discretas de infinitos puntos. A pesar de ello, no lo haremos de esta forma y aprovecharemos que estamos en el caso continuo. Así, bastara comprobar que:

1. $f(x) \geq$ 0 para toda $x$.
2. $\int_{-\infty}^{\infty} f(x)\, dx = 1$

Donde f es la función de densidad. Las llamadas que realizamos a Maxima para comprobar estas propiedades son las siguientes:

```
assume(a >= 0); is (f(a) >= 0);
integrate(diff(f(x),x,1),x," + infer + "," + sup + ");
```

Donde infer y sup determinan el dominio de la función y sus valores son indicados en la vista 5.

## 5.6 CONTROLADOR

Nada destacable de esta clase. Se limita a cumplir la función que le da nombre en el modelo-vista-controlador.

## 5.7 LIBRERÍA MAXIMA

### 5.7.2 NOTACIÓN

Es importante conocer y seguir correctamente la notación, sobre todo en dos casos, para el correcto funcionamiento del proyecto. Estos son:

**Caso 1:** Como se ha mencionado anteriormente, hemos elegido la sintaxis que usa Maxima para que el usuario introduzca las funciones de distribución. Esta elección nos ha simplificado mucho el trabajo de función ad-hoc que teníamos establecido en el anteproyecto realizando Maxima el mayor trabajo. Así que la elección (obligada) de una sintaxis para las funciones ha sido determinante para el resultado final de dicho proyecto.

También es una decisión lógica elegir esta y no otra sintaxis. No solo por ahorrarnos todo el proceso de ad-hoc en Java. Por un lado, tenemos que elegir una. Por otro, este programa lo que crea es un paquete en Maxima, para ser usada o bien en Maxima (el usuario conoce la sintaxis Maxima) o bien usando Jacomax para un programa en Java (también usuario debe conocer la sintaxis Maxima).

**Caso 2:** Esta parte es la notación que usa nuestra librería. Así que, la necesitamos saber para poder usarla. Más adelante, vamos a ver como son las llamadas a las funciones que vienen definidas en esta librería. Así que, veamos la definición de las funciones que añadimos.

### 5.7.3 FUNCIONES DEFINIDAS

#### *5.7.3.1 DISCRETAS*

Éstas son las funciones de distribución discretas que pueden ser usadas con nuestro paquete Maxima.

```
/* Degenerada  */

degeneradaDist(x,x0) := block( if x>= x0 then 1 else 0)$
degeneradaMasa(x,x0) := block(if x = x0 then 1 else 0)$
degeneradaG(x0) := block(x0)$
vector_degeneradaG(x0,n) := block( for i:1 thru n do(
      vec[i] : x0
),
```

```
vec

)$

/*Uniforme x=a,a+1,a+2, ... ,b-1,b*/

uniformeDDist(x,a,b) := block( if x < a then 0 elseif x > b else (1/(b-
a+1))*(floor(x+1)-a))$
uniformeDMasa(x,a,b) := block(if x >= a and x <=b and esEnetero(x) = 1
then 1/(b-a+1) else 0)$
uniformeDG(a,b) := block(funcionDiscretaGeneradora(1/(b-a+1),a))$
vector_uniformeDG(a,b,n) := block(vector_funcionDiscretaGeneradora(1/(b-
a+1),a,n) )$

/*Bernouille x = 0,1*/

bernouilleDist(x,p) := block( if x < 0 then 0 elseif x>= 0 and x < 1 then
1 - p else 1 )$
bernouilleMasa(x,p) := block( if x = 0 then 1-p elseif x = 1 then p else
0)$
bernouilleG(p) := block(funcionDiscretaGeneradora((p^x)*((1-p)^(1-x)),0))$
vector_bernouilleG(p,n) :=
block(vector_funcionDiscretaGeneradora((p^x)*((1-p)^(1-x)),0,n) )$

/*Rademacher x = -1,1*/

rademacherDist(x) := block( if x < -1 then 0 elseif x>= -1 and x < 1 then
1/2 else 1 )$
rademacherMasa(x) := block( if x = -1 then 1/2 elseif x = 1 then 1/2 else
0)$
/*rademacherG()  := block(funcionDiscretaGeneradora())$
vector_rademacherG() := block(vector_funcionDiscretaGeneradora(,n) )$ */

/*Binomial x=0,1,2,...,n*/

binomialDist(x,n,p) := block(if x < 0 then 0 elseif x >= n then 1 else
sum(binomial(n,j)*p^j*((1-p)^(n-j)),j,0,floor(x)))$
binomialMasa(x,n,p) := block( if x >= 0 and x <=n and esEnetero(x) = 1
then binomial(n,x)*(p^x)*((1-p)^(n-x)) else 0)$
binomialG(n,p) :=
block(funcionDiscretaGeneradora(n*binomial(n,x)*((p^x)*((1-p)^(n-x))),0))$
vector_binomial(n,p,n1) :=
block(vector_funcionDiscretaGeneradora(n*binomial(n,x)*((p^x)*((1-p)^(n-
x))),0,n1))$

/*Poisson x=0,1,2,3,...*/

poissonDist(x,lambda) := block( sum((%e^(-
lambda))*((lambda^j)/j!),j,0,floor(x)))$
poissonMasa(x,lambda) := block( if x >= 0 and esEnetero(x) = 1 then (%e^(-
lambda))*((lambda^x)/x!) else 0)$
poissonG(lambda) := block(
      [c,b,k,aux],
```

```
      c:(%e)^(-lambda),

      b:c,

      k:0,

      aux:uniformeG(0.0,0.1),

      while aux>b do(k:k+1,c:(lambda/k)*c,b:b+c),k
)$
vector_poissonG(lambda,n) := block(for i:1 thru n do(
      vec[i] : poissonG(lambda)
),
vec)$


/*Geometric x=0,1,2,...*/

geometricDist(x,p) := block( if x >= 0 then sum((p*(1-p)^j),j,0,floor(x))
else 0)$
geometricMasa(x,p) := block( if x >= 0 and esEnetero(x) = 1 then p*(1-p)^x
else 0 )$
geometricG(p,b) := block(funcionDiscretaGeneradora(p*((1-p)^(x)),0))$
vector_geometricG(p,n) := block(vector_funcionDiscretaGeneradora(p*((1-
p)^(x)),0,n) )$

/*Negative Binomial x=0,1,2,...*/

negative_BinomialDist(x,r,p) := block(if x >= 0 then sum(((binomial(r+j-
1,j))*(p^r)*((1-p)^(j))),j,0,floor(x)) else 0)$
negative_BinomialMasa(x,r,p) := block(if x >= 0 and esEnetero(x) = 1 then
(binomial(r+x-1,x))*(p^r)*((1-p)^(x)) else 0)$
negative_BinomialG(r,p) :=
block(funcionDiscretaGeneradora(binomial(r+x,x)*(p^r)*((p)^(r))*((1-
p)^x),0))$
vector_negative_BinomialG(r,p,n) :=
block(vector_funcionDiscretaGeneradora(binomial(r+x,x)*(p^r)*((p)^(r))*((1
-p)^x),0),n) )$

/*Hypergeometric x=max(0,n1+m-n),...,min(n1,m)*/

hypergeometricDist(x,n,m,n1) := block(if x < max(0,n1+m-n) then 0 elseif x
> min(n1,m)then 1 else sum((((binomial(n1,j))*(binomial(n-n1,m-
j)))/(binomial(n,m))),j,max(0,n1+m-n),floor(x)))$

hypergeometricMasa(x,n,m,n1) := block(if x >= max(0,n1+m-n) and x <=
min(n1,m) and esEntero(x) = 1 then ((binomial(n1,x))*(binomial(n-n1,m-
x)))/(binomial(n,m)) else 0)$

hypergeometricG(n,m,n1) :=
block(funcionDiscretaGeneradora(((binomial(n1,x))*(binomial(n-n1,m-
x)))/(binomial(n,m)) ,max(0,n1+m-n)))$

 vector_hypergeometricG(n,m,n1,n2) :=
```

```
block(vector_funcionDiscretaGeneradora(((binomial(n1,x))*(binomial(n-n1,m-x)))/(binomial(n,m)) ,max(0,n1+m-n),n2) )$
```

#### *5.7.3.2 CONTINUAS*

Éstas son las funciones de distribución continuas que pueden ser usadas con nuestro paquete Maxima.

```
/*Uniforme*/

uniformeDist(x,a,b) := block( (x -a)/(b-a))$                    /* FUNCIÓN DE DISTRIBUCIÓN */
uniformeDens(x,a,b) := block( 1/(b-a))$                    /* FUNCIÓN DE DENSIDAD */
uniformeG(a,b) := block( a + (b-a)*aleatorio(0.0,1.0))$    /* FUNCIÓN QUE GENERA UN VALOR DE LA VARIABLE ALEATORIA */
vector_uniformeG(a,b,n) := block(
for i:1 thru n do(
        vec[i]:uniformeG(a,b)
),
vec
   )$  /*FUNCIÓN QUE GENERA UN VECTOR DE TAMAÑO n DE LA VARIABLE ALEATORIA*/

/*Exponencial*/

exponencialDist(x,lambda) := block( 1 - %e^(-lambda*x))$
exponencialDens(x,lambda) := block( lambda*%e^(-lambda*x))$
exponencialG(lambda):=block((-1/lambda)*log(uniformeG(0.0,1.0)))$
vector_exponencialG(lambda,n) := block(
for i:1 thru n do(
        vec[i] : exponencialG(lambda)
),
vec
)$

/*Normal*/

normalDist(x,media,varianza):= block(integrate(((1/(varianza*2*%pi)^(0.5)*(%e^(0.5*(((t-media)^2)/(varianza)))))),t,minf,x))$
normalDens(x,media,varianza):= block( (1/(varianza*2*%pi)^(0.5)*(%e^(0.5*(((t-media)^2)/(varianza))))) )$
normalG(media,varianza) := block(media + (varianza^0.5)*(((-2*log(uniformeG(0.0,1.0)))^0.5)*cos(2*%pi*uniformeG(0.0,1.0))))$
normalG1(media, varianza):= block(media + (varianza^0.5)*(((-2*log(uniformeG(0.0,1.0)))^0.5)*sin(2*%pi*uniformeG(0.0,1.0))))$
vector_normalG(media,varianza,n) := block(
for i:1 thru n do(
        vec[i] : normalG(media,varianza)
),
vec
)$
```

```
/*Log-normal distribution */

lognormalDist(x,media,varianza) :=
block(integrate(((1/(t*((varianza)^0.5)*((2*%pi)^0.5)))*(%e^(((-log(t)-
media)^2/(2*varianza))))),t,0,x) )$
lognormalDens(x,media,varianza) := block((1/(x*((varianza)^0.5)*((2*%pi)^0.5)))*(%e^(((-
log(x)-media)^2/(2*varianza)))))$
lognormalG(media,varianza) := block(%e^(normalG(media,varianza)))$
lognormalG1(media,varianza) := block(%e^(normalG1(media,varianza)))$
vector_lognormalG(media,varianza,n) := block(
for i:1 thru n do(
        vec[i] : lognormalG(media,varianza)
),
vec
)$

/*Weibull*/

weibullDist(x,alfa,beta) := block( 1 - %e^((-x/alfa)^beta)$
weibullDens(x,alfa,beta) := block( (beta/alfa)*((x/alfa)^(beta-1))*(%e^(-x/alfa)^(beta)))$
weibullG(alfa,beta) := block(beta*(exponencialG(1))^(1/alfa))$
vector_weibullG(alfa,beta,n) := block(
for i:1 thru n do(
        vec[i] : weibullG(alfa,beta)
),
vec
)$

/*Gamma*/

gammaDist(x,alfa,beta) := block(integrate(gammaDens(t,alfa,beta),t,0,x) )$
gammaDens(x,alfa,beta) := block(((x^(alfa -1))*(%e^((-
x)/(beta))))/((beta^alfa)*(auxGamma(alfa))))$
gammaG(alfa,beta) := block(
        if alfa = 1 then auxGamma1(alfa,beta) elseif alfa < 1 then auxGamma2(alfa,beta)
        elseif beta=1 then auxGamma3(alfa,beta) elseif alfa > 20 and beta>20 then
auxGamma4(alfa,beta) else
        auxGamma(alfa,beta)
)$
auxGamma(alfa,beta) := block([u,aux,ufinal],
        u:uniformeG(0,1),
        aux : product(uniformeG(0,1), k, 1, floor(alfa)),
        ufinal :uniformeG(0,1),
        if (alfa - floor(alfa)) <= u then aux:aux*ufinal,
        -beta*log(aux)
 )$
auxGamma1(alfa,beta) := block(exponencialG(beta))$
auxGamma2(alfa,beta) := block( betaG(alfa,1-alfa)*exponencialG(beta) )$
auxGamma3(alfa,beta) := block( [u1,u2,v,x,a,b,c],
        u1:uniformeG(0,1),
        u2:uniformeG(0,1),
        a:1/((2*alfa -1)^0.5),
        b: alfa-log(4),
        c=alfa+a,
        v:a*log(u1/(1-u1)),
```

```
        x:alfa*((%e)^v),
        if b+c*v -x >= log((u1^2)*u2) then x else auxGamma3(alfa,beta)
)$
auxGamma4(alfa,beta) := block( %e^(normalG(log((alfa/beta)) - (1/(2*alfa)),(1/alfa)^0.5 )) )$
vector_gammaG(alfa,beta,n) := block(
for i:1 thru n do(
        vec[i] : gammaG(alfa,beta)
),
vec
)$
```

/***Beta***/

```
betaDist(x,alfa,beta) := block(integrate(betaDens(t,alfa,beta),t,0,x) )$
betaDens(x,alfa,beta) := block( (auxGamma(alfa +
beta)/(auxGamma(alfa)*auxGamma(beta)))*((x^(alfa-1))*((1-x)^(beta-1))))$
betaG(alfa,beta) := block( if alfa < 4 or beta < 4 then auxBeta1(alfa,beta) else
auxBeta2(alfa,beta) )$
auxBeta1(alfa,beta) := block([y1,y2],
        y1 : gammaG(alfa,1),
        y2 : gammaG(1,beta),
        y1/(y1 + y2)
 )$
auxBeta2(alfa,beta) := block([z],
        z:normalG((log(alfa/beta) + (alfa - beta)/(2*alfa*beta)),(((alfa +
beta)/(alfa*beta))^(1/2))),
        (%e^(z))/(1 + %e^(z))
)$
auxBeta3(alfa,beta) := block()$
vector_betaG(alfa,beta,n) := block(
for i:1 thru n do(
        vec[i] : betaG(alfa,beta)
),
vec
)$
```

/***Chi-Square***/

```
chiDist(x,k) := block(integrate(chiDens(t,k),t,0,x))$
chiDens(x,k) := block((x^((k/2)-1))*(%e^(-x/2))/((2^(k/2))*(auxGamma(k/2))))$
chiG(k) := block( if k > 30 then auxChi1(k) else auxChi2(k) )$
auxChi1(k) := block([z],
z:normalG(0,1),
((z + (2*k -1)^(1/2))^2)/(2)
)$
auxChi2(k) := block( gammaG(k/2,2))$
vector_chiG(k,n) := block(
for i:1 thru n do(
        vec[i] : chiG(k)
),
vec
)$
```

/***Student's t***/

```
studentsTDist(x,k) := block(integrate(studentsTDens(t,k),t,minf,x))$
studentsTDens(x,k) := block( ((auxGamma(k+1/2))/(auxGamma(k/2)*((k*%pi)^(0.5))))*((1 +
(x^2)/(k))^((-k-1)/(2))))$
studentsTG(k) := block( normalG(0,1)/((chiG(k)/k)^0.5))$
vector_studentsTG(k,n) := block(
for i:1 thru n do(
        vec[i] : studentsTG(k)
),
vec
)$

/*F*/

fDist(x,k1,k2) := block(integrate(fDens(t,k1,k2),t,0,x))$
fDens(x,k1,k2) := block(((auxGamma((k1+k2)/2))*((k1/k2)^(k1/2))*((x)^((k1/2)-1))
)/((auxGamma(k1/2)*auxGamma(k2/2))*((1 + (k1*x)/(k2))^((k1 + k2)/2))))$
fG(k1,k2) := block( (chiG(k1)/k1)/(chiG(k2)/k2))$
vector_fG(k1,k2,n) := block(
for i:1 thru n do(
        vec[i] : fG(k1,k2)
),
vec
)$

/*Z*/

zDist(x,k1,k2) := block(integrate(zDens(t,k1,k2),t,minf,x))$
zDens(x, k1,k2) :=
block(((2*auxGamma((k1+k2)/2))*((k1/k2)^(k1/2))*((%e)^(k1*x)))/((auxGamma(k1/2))*(auxGa
mma(k2/2))*((1 + ((k1*(%e)^(2*x))/(k2)))^((k1 + k2)/(2)))))$
zG(k1,k2) := block( log(fG(k1,k2))/2 )$
vector_zG(k1,k2,n) := block(
for i:1 thru n do(
        vec[i] : zG(k1,k2)
),
vec
)$

/*Pareto*/

paretoDist(x,x0,alfa) := block(integrate(paretoDens(t,x0,alfa),t,x0,x))$
paretoDens(x, x0,alfa) := block((alfa/x0)*((x0/x)^(alfa+1)))$
paretoG(x0,alfa) := block( x0/((uniformeG(0,1))^(1/alfa))$
vector_paretoG(x0,alfa,n) := block(
for i:1 thru n do(
        vec[i] : paretoG(x0,alfa)
),
vec
)$

/*Logistic*/

logisticDist(x,alfa) := block( integrate(logisticDens(t,alfa),t,minf,x))$
logisticDens(x,alfa) := block( ((alfa)*((%e)^(-x)))/((1 + (%e)^(-x))^(alfa+1)))$
logisticG(alfa) := block( -log((1-(uniformeG(0,1))^(1/alfa))/((uniformeG(0,1))^(1/alfa))))$
```

```
vector_logisticG(alfa,n) := block(
for i:1 thru n do(
        vec[i] : logisticG(alfa)
),
vec
)$

/*Cauchy*/

cauchyDist(x,alfa,beta) := block(integrate(cauchyDens(t,alfa,beta),t,minf,x))$
cauchyDens(x, alfa,beta) := block((beta)/((%pi)*(beta^2 + (x - alfa)^2)) )$
cauchyG(alfa,beta) := block( alfa + beta*(tan(%pi*(uniformeG(0,1) - 0.5))) )$
vector_cauchyG(alfa,beta,n) := block(
for i:1 thru n do(
        vec[i] : cauchyG(alfa,beta)
),
vec
)$

/*Irwin-Hall*/

irwinhallDist(x,n) := block(integrate(irwinhallDens(t,n),t,minf,x))$
irwinhallDens(x,n) := block((1/(2*(n-1)!) )*(sum((((-1)^k)*(binomial(n,k))*((x-k)^(n-1))*(sgn(x-
k))) ,k,0,n)));
irwinhallG(n) := block( sum(uniformeG(0,1),j,0,n))$
vector_irwinhallG(n1,n) := block(   for i:1 thru n do(         vec[i] : irwinhallG(n1)   ),   vec   )$
```

### 5.7.4 ALGORITMOS

Aquí podemos ver los algoritmos implementados en Maxima. Son usados para generar valores de una variable aleatoria continua o discreta a partir de su función de distribución.

#### *5.6.3.1 CONTINUAS*

```
/*Continuas */
funcionContinuaGeneradora(f,ini):= block([v,ok,res],
      ok:1,
v:float(solve([uniformeG(0,1) = integrate(f,x,ini,x)],[x])),
            for i:1 thru length(v) do(
                  if esReal(v[i]) and ok=1 then
                      res:v[i],ok:0),
            if ok=1 then
                 res:v[1],
res
)

vector_funcionContinuaGeneradora(f,ini,n) := block(
for i:1 thru n do(
vec[i] : funcionContinuaGeneradora(f,ini)
```

```
),
vec
)$
```

Calculamos todas las soluciones de:

uniformeG(0,1)=integrate(f,x,ini,x)

donde uniformeG(0,1) es un número aleatorio entre 0 y 1 y integrate(f,x,ini,x) es la integral de la función de densidad f, entre inicio y x, siendo x la variable de la ecuación.

Todas las posibles soluciones la incluimos en un vector v. De este vector tomamos un valor real y si no existe ninguno el primero de ellos.

#### 5.6.3.2 DISCRETAS

```
/*Discretas*/

funcionDiscretaGeneradora(f,ini) := block(
[u,aux,k],
k:ini,
u:uniformeG(0.0,1.0),
aux:0,
while u>aux do( aux:aux+f(k),
                        k:k+1),
k
)$

funcionDiscretaGeneradora0(f) := block( funcionDiscretaGeneradora(f,0) $

funcionDiscretaGeneradora1(f,vec_puntos) := block(
[u,aux,k,ind],
k:vec_puntos[1],
u:uniformeG(0.0,1.0),
aux:0,
ind:1,
while u>aux do( aux:aux+f(k),
                        ind:ind+1,
                        k:vec_puntos[ind]
                        ),
k
)$

funcionDiscretaGeneradora2(vec_prob,vec_puntos) := block(
      [u,ind,prob],
      u:uniformeG(0.0,1.0),
      ind:1,
      prob:0,
      while u>prob do (
            prob:prob+vec_prob[ind],
```

```
            ind:ind+1,
      ),
      vec_puntos[ind]
)$

vector_funcionDiscretaGeneradora(f,ini,n) := block(
      for i:1 thru n do(
      vec[i] : funcionDiscretaGeneradora(f,ini)
),
vec
)$

vector_funcionDiscretaGeneradora0(f,n) := block(
vector_funcionDiscretaGeneradora(f,0,n) )$


vector_funcionDiscretaGeneradora1(f,vec_puntos,n) := block(
      for i:1 thru n do(
      vec[i] : funcionDiscretaGeneradora1(f,vec_puntos)
),
vec
)$


vector_funcionDiscretaGeneradora2(vec_prob,vec_puntos,n) := block(
      for i:1 thru n do(
      vec[i] : funcionDiscretaGeneradora2(vec_prob,vec_puntos)
),
vec
)$
```

Supondremos que la discreta comienza en 0 y aumenta de uno en uno, salvo que se indique lo contrario.

Lanzamos un número aleatorio entre 0 y 1 y desde el valor inicial de la variable, vamos recorriendo los puntos donde dicha variable tiene probabilidad uno a uno y si aux que es una variable iniciada a 0, le vamos a aumentando en cada punto de la variable aleatoria tanto como probabilidad tenga en ese punto, hasta que esta supere el valor del numero aleatorio antes calculado. Entonces dicho punto será lo que devuelva el algoritmo.

### 5.7.5 AUXILIARES

Algunas funciones auxiliares que se han necesitado

```
/*Función gamma*/

auxGamma(x) := block (integrate((t^(x-1))*(%e^(-t)), t, 0, inf));
```

```
/*Función signo*/

sgn(x ):= block( if x < 0 then -1 elseif x = 0 then 0 elseif x > 0 then 1
);

/* Función esEntero*/

esEnetero(x) := block( if floor(x) = ceiling(x) then 1 else 0)$


/*Función esReal */


esReal(x) := block( if imagpart(x) = 0 then 1 else 0)$


/* Función aleatoria */

aleatorio(a,b) := block( a + (b-a)*random(1.0))$
```

En este apartado, es importante realizar mención a la función aleatorio. Actualmente esta implementada usando la función random, pero puede ser interesante realizar otro tipo de implementación más avanzada. De este modo, todas las funciones de distribución que hemos implementados usarán la nueva implementación.

## 5.8 PAQUETE MAXIMA PERSONALIZADO

### 5.8.2 CONTENIDO

Éste es el paquete que crea nuestro programa de la forma que hemos visto en 5.3.1.

El contenido de este paquete se compone del paquete que hemos visto en el apartado anterior, esto es, todas las funciones continuas, discretas y auxiliares antes definidas. Los algoritmos antes vistos. Y todas las funciones que hayan sido añadidas por el usuario.

### 5.8.3 MANUAL PARA SU USO

Veamos las diferentes formas de usar nuestro programa.

- ***Desde Maxima***

Si usamos Maxima, bien sea para un cálculo inmediato, o creando un programa complejo, la forma de uso es cargar un paquete que hayamos creado

previamente, siguiendo los pasos del punto 5.3.1 y tan solo será necesario realizar las llamadas a todas las funciones definidas en el.

Por ejemplo, supongamos que hemos creado con nuestro programa un paquete de tres funciones de distribución, éstas serían unas de las posibles llamadas:

```
1. exponencialDens(2,3);
2. vector_weibullG(1,2,7);
3. fG2();
```

**Salida 1**. Obtendremos el valor de la función de densidad de una exponencial de lambda 3 evaluada en 2.

**Salida 2**. Un vector de tamaño 7  con los valores de una variable aleatoria Weibull de alpha 1 y beta 2.

**Salida 3**. Devolverá el valor de la variable aleatoria definida por la segunda función de distribución que introdujimos en el momento de crear el paquete.

- ***Desde Java***

Una vez que sabemos cómo se utiliza en Maxima, es fácil ver cómo podemos usarlo en Java. Tan sólo tenemos que realizar una comunicación con Maxima realizando una llamada en la que incluyamos el paquete cargado. Veámosla en un ejemplo:

```
"s1:make_random_state (true);
set_random_state(s1);
load(\"C:/paquetesMaxima/funcDist.mac\")$
fG2()"
```

En este caso, se realiza desde Java la llamada a Maxima 3, del ejemplo anterior. Una vez realizada la llamada, tan sólo es necesario saber interpretar la salida.

- ***Desde un proyecto de simulación usando Maxima***

Éste es uno de los puntos principales del proyecto. Y es dotar de la posibilidad de introducir cualquier función de distribución en los proyectos [1] y [2].

Con la elección de la sintaxis Maxima y la decisión de poder crear un paquete con funciones dadas por el usuario y funciones usuales, se ha dado un enfoque más general, con el cual, se puede utilizar en más casos.

Para adaptar los proyectos [1] y [2], y previsiblemente se adaptará [3], (ya que no es algo costoso de realizar), tenemos que seguir los siguientes pasos.

- Cada función de distribución, función de densidad, función de masa, variable aleatoria o vector de variable aleatoria que exista en el fichero Maxima del proyecto debe ser renombrado por la sintaxis de nuestra librería. Es decir, debe de llevar uno de estos nombres:

```
• fDist1(x);
• fDen1(x);
• fG1();
• vector_fG1(n);
```

- Cargar el paquete en Maxima que se desee usar. Se pueden crear tantos paquetes como se deseen y usar en cada momento el que más nos interese.

Por último, veamos cómo se carga un paquete. Es tan simple como llamar en Maxima a:

```
load("C:/Users/MANUEL/Desktop/workspace_proyecto/LibreriaVariablesAleato
rias.wxm")$
```

Donde C:/Users/MANUEL/Desktop/workspace_proyecto/LibreriaVariablesAleatorias.wxm es la ubicación del paquete Maxima en el ordenador.

Otra forma a través de la interfaz que nos proporciona Maxima es: Archivo -> Cargar Paquete y elegir archivo que se desee, (Ctrl + L con el atajo de teclado).

# 6 NÚMERO DE PASOS QUE REALIZA MAXIMA

## 6.1 INTRODUCCIÓN Y MOTIVACIÓN

Con este objetivo se pretende que los proyectos [1] y [2] mejoren en rendimiento. Como hemos visto, la simulación se realiza en Maxima y se comunica con Java para mostrar el resultado. Dicha comunicación, se realiza una sola vez por cada iteración que realiza Maxima, es decir, en [1], Maxima calcula un paso del tráfico de maletas en un aeropuerto, envía el nuevo estado a Java y este muestra al usuario el nuevo estado. En [2], ocurre algo similar, Maxima calcula un paso del tráfico de coches en el mapa y envía de nuevo todos los datos a Java que los transforma para entenderlos y poder mostrarlos en el dibujo del mapa en la interfaz de usuario.

Si dividimos este proceso en cuatro pasos:

- Cálculo de un paso en Maxima.
- Comunicación de Maxima con Java.
- Java pinta el nuevo estado para que el usuario pueda verlo.
- Java vuelve a llamar a Maxima para que calcule el siguiente paso.

Tenemos que los pasos en los que existe la comunicación entre Java y Maxima son los más costosos y más tiempo consumen.

Es interesante tener una interfaz de usuario en Java, ya que, nada ayuda más a entender lo que está pasando que obtener la información de forma visual, así que poder ver la simulación en detalle, paso a paso, es muy útil. Pero también, puede ser realmente útil generar las estadísticas directamente sin esperar a ver la simulación paso a paso, y de este modo poder obtener una gran cantidad de estadísticas para posteriormente ser analizadas, en menos tiempo. Con este fin, se ha modificado el proceso de los proyectos, pidiéndole al usuario un valor que indique cada cuántos pasos de simulación, Maxima se comunicará con Java, reduciendo así la comunicación entre los dos programas, la cual era el elemento que más tiempo consumía.

Ahora el proceso queda de la siguiente manera:

- El usuario decide cada cuantos pasos se realizará la comunicación. Sea n dicho valor.

- Maxima calcula n pasos.
- Maxima se comunica con Java devolviendo el último estado calculado.
- Java pinta el nuevo estado para que el usuario pueda verlo.
- Java vuelve a llamar a Maxima para que calcule los siguientes n pasos.

## 6.2 IMPLEMENTACIÓN

Se ha realizado la implementación en los dos proyectos con diferentes resultados. Veámoslos por separado.

### 6.2.1 TRÁFICO DE MALETAS EN UN AEROPUERTO

En este caso, se ha realizado la implementación correctamente y esto ha llevado a que los objetivos de rendimiento hayan sido realizados con éxito. Primero, pedimos al usuario el valor que indica cada cuántos pasos desea que se actualice la interfaz. Sea n este número. La nueva llamada que realizamos a Maxima es:

```
llamada = "s1:make_random_state
(true);set_random_state(s1);load(\"C:/SimulacionMaletas/f2.mac\")$"+

      matrizToMaxima(mostradores,"mos")+matrizToMaxima(anillo,"ani")+matrizToMaxim
a(hipodromos,"hip")+

      matrizToMaxima(cjtoMaletas,"cjt")+matrizToMaxima(almacen,"alm")+matrizToMaxi
ma(seguridad,"seg")

      +matrizToMaxima(contadores,"cont")+matrizToMaxima(cuentas,"cola")+matrizToMa
xima(probabilidadesMaletas,"prob")
            +matrizToMaxima(parametros,"para")+"tramo:"+tramo+"$"

      +"avanzarPasos(mos,ani,hip,cjt,alm,seg,cont,cola,prob,para);mos;ani;hip;cjt;
alm;seg;cont;cola;prob;";
```

Donde en avanzarPasos, “para” vale n;

Y en código Maxima realizamos lo siguiente:

```
avanzarPasos(mostrador, anillo, hipodromo, maletas, almacen, seguridad,
contmaletas, cola,prob, para) :=block(
      for z:1 thru para do (
            avanza(mostrador, anillo, hipodromo, maletas, almacen,
seguridad, contmaletas, cola,prob, para)
      )
)$
```

Donde avanza es el método que calcula un solo paso.

### 6.2.2 TRÁFICO INTELIGENTE EN UNA ZONA DE MALAGA

En este caso, aunque podemos ver la interfaz actualizarse cada cuántos pasos indique el usuario, el rendimiento no habrá sido mejorado tanto como en el caso anterior. Veamos a qué se debe.

La diferencia que se produce entre este proyecto y el anterior es una mala separación entre la parte que le corresponde las funciones de computación (Maxima) y la parte encargada de mostrar la interfaz (Java). En este caso, Java realiza un cálculo necesario para pasar de un estado a otro. En concreto, en Java está calculado el algoritmo Dijkstra que indica hacia donde se tiene que dirigir un coche cada vez que llega a un cruce o se modifica un sentido en el mapa. Este problema conlleva que Maxima es dependiente de Java para realizar la computación entre un estado y otro, es decir, es necesaria la comunicación.

La implementación, en este caso, ha sido más tediosa, ya que en todo momento se tenía que comprobar si algún coche estaba en un cruce, para comunicarnos con Java y pedirle la información que calculaba el algoritmo Dijkstra, así como, llevar un control de cuántos pasos se llevaban calculados en cada momento para saber si además de llamar el Dijkstra se debe de pintar el nuevo estado del mapa.

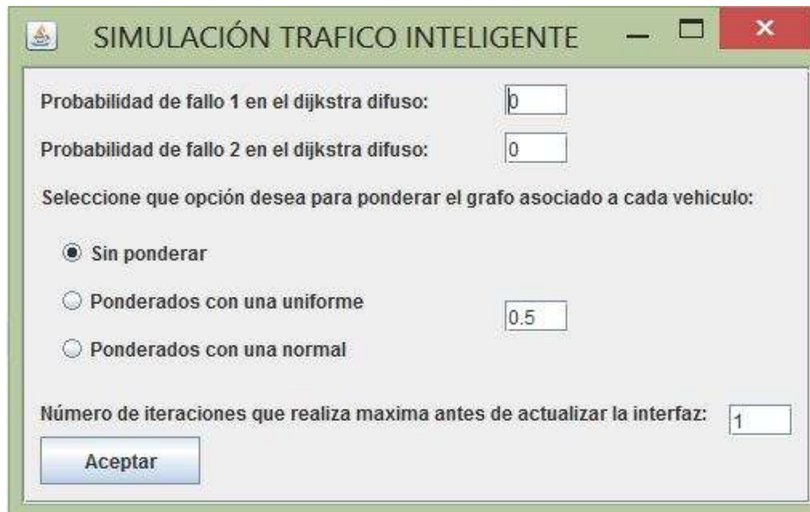
Respecto al rendimiento, la mejora ha sido mucho menos productiva que en el primer caso, pues la comunicación, se realiza menos veces, ya que, Maxima puede calcular sin comunicarse con Java tantos pasos como tarde un coche en llegar a un cruce, siempre serán menos comunicaciones que comunicándonos en cada iteración, aunque en una simulación con un número de coches considerables, será muy pequeña.

Puede ser interesante en posteriores versiones de este proyecto, implementar el algoritmo Dijkstra en Maxima, que es a quien le corresponde dicho cálculo.

## 6.3 MANUAL DE USUARIO

Tan sólo tenemos que añadir el valor que indica cada cuántos pasos se deben de comunicar Java con Maxima.

En [2] lo introducimos en esta vista:




Y en [1]:

# 7 INTEGRACIÓN DEL PROGRAMA DEFINIDO EN 5, EN UN PROYECTO DE SIMULACIÓN DE FLUJO DE TRÁFICO EN TIEMPO ACELERADO

Veamos un ejemplo de uso del programa que crea un paquete Maxima de funciones de distribución en un proyecto de simulación de flujo de tráfico en tiempo acelerado, en concreto en el proyecto [3].

Primero, hemos localizado cuales son las variables aleatorias en las que deseamos que sea el usuario el que las defina. Éstas son:

- La variable aleatoria que define cada cuánto tiempo llegan personas a una parada.
- La variable aleatoria que define cuántas personas llegan a una parada.
- La variable aleatoria que determina la frecuencia en la que se produce la avería de un autobús.

A continuación, se crean siete paquetes Maxima, y se cargan en el fichero Maxima que realiza la simulación. Cada uno de ellos contendrá una de las variables aleatorias antes definidas. Se diferenciará los paquetes en función del tráfico. Estos paquetes son:

- frecuenciapersonasintenso.mac
- frecuenciapersonasmoderado.mac
- frecuenciapersonasligero.mac
- cantidadpersonasintenso.mac
- cantidadpersonasmoderado.mac
- cantidadpersonasligero.mac
- frecuenciaaverias.mac

Las variables aleatorias que tendrán por defecto cada uno de ellos en el mismo orden que han sido mostrados antes, son:

- exponencial de lambda 1.
- exponencial de lambda 0.1.
- exponencial de lambda 0.03.
- poisson de lambda 5.
- poisson de lambda 3.

- poisson de lambda 1.
- exponencial de lambda 0.003.

Se ha implementado en Java la posibilidad de que el usuario modifique cualquiera de estos paquetes, cambiando así la variable aleatoria que desee, para que la simulación se adapte a la situación que desee probar.

La siguiente es una vista que da acceso para modificar los paquetes anteriores.

Cada vez que pulsemos el botón Modificar, se ejecutará el programa visto en el punto 5, que se encarga de crear el paquete deseado.

En el fichero Maxima encargado de realizar la simulación del flujo de tráfico, se cargarán dichos paquetes, atendiendo a la simulación del tráfico. En código será así:

```
load("C:/paquetesMaxima/frecuenciaaverias.mac")$

 /* Dependiendo del tipo de tráfico establecido se carga un paquete
u otro */

if (ceiling(tráfico[5])=1)then(
   /* Tráfico ligero */
   load("C:/paquetesMaxima/frecuenciapersonasligero.mac"),
   load("C:/paquetesMaxima/cantidadpersonasligero.mac")
```

```
      )else if (ceiling(tráfico[5])=2)then(
         /* Tráfico moderado*/
        load("C:/paquetesMaxima/frecuenciapersonasmoderado.mac"),
        load("C:/paquetesMaxima/cantidadpersonasmoderado.mac")
      )else(
        load("C:/paquetesMaxima/frecuenciapersonasintenso.mac"),
        load("C:/paquetesMaxima/cantidadpersonasintenso.mac")
)$
```

Una vez realizados todos estos pasos, cada vez que en la simulación (Maxima) se necesite un valor de una de estas variables aleatorias bastará con realizar una de las siguientes llamadas:

- frecuenciapersonasfG()
- cantidadpersonasfG()
- frecuenciaaveriasfG()

Se han realizado pequeñas modificaciones en el código del programa del punto 5 de la memoria para que se adapte a la nomenclatura seguida en [3].

La decisión de crear siete paquetes Maxima y cargar unos u otros dependiendo del tráfico, en vez de crear un solo paquete, ha servido para que en cualquier momento se pueda cambiar la definición de una de las variables sin necesidad de modificar las otras o volverlas a introducir. Además el paquete se puede cambiar en cualquier momento, inclusive cuando el programa está en ejecución.

# 8 CONCLUSIONES

## 8.1 OBJETIVO INICIAL Y RESULTADO FINAL

Veamos los cuatro objetivos uno por uno.

1. **Conseguir que funcionen los proyectos [1] y [2] en una máquina Mac**. Se ha conseguido realizar, aunque el código no es completamente portable, ya que se tienen que modificar las direcciones, la configuración y los métodos que interpretan la salida de Maxima.

2. **Realizar la comunicación Java-Maxima cada n pasos.** También el usuario puede decidir cada cuántos pasos se realiza la comunicación Java-Maxima, aunque cómo se ha comentado en el proyecto [2], se realiza una llamada interna al algoritmo Dijkstra en Java, cuando un coche llega a un cruce y esto puede ocurrir antes de que se agoten los n pasos en Maxima.

3. **Algoritmo Dijkstra difuso.** Se ha realizado el Dijkstra difuso, con dos posibles implementaciones. Además, se ha añadido una nueva mejora con el mismo fin que tenía el implementar el algoritmo Dijkstra difuso (simular lo más realista posible el camino que elegimos para llegar de un origen a un destino). Esta es que cada vehículo tenga un grafo asociado propio (será uno ponderado del grafo comun).

4. **Las funciones de distribución de [1] y [2], sean las añadidas por el usuario.** Se ha conseguido el propósito. Además se ha creado un paquete Maxima el cual puede ser incluido en cualquier proyecto similar, en cualquier código Maxima, o llamado desde Java siempre que se desee.

## 8.2 POSIBLES MEJORAS

En [2] sería interesante realizar un mapa dinámico de forma similar a cómo se ha implementado [3], con esto se podrían realizar simulaciones en diferentes grafos, ampliando así las situaciones que simular y pudiendo crear así diferentes situaciones.

Otra mejora que sería interesante es realizar el cálculo que realiza el algoritmo Dijkstra en Maxima, siendo así la separación entre Maxima (cálculo del paso entre un estado y el siguiente) y Java (realiza la interfaz de usuario), completa.

Los coches pueden tener prioridad, así, ganaríamos en realismo, ya que podríamos simular ambulancias o camiones de bomberos.

Actualmente, los coches tienen asignadas una entrada y una salida en el mapa siendo ese su recorrido. Se podría implementar que su recorrido finalice en un punto cualesquiera del mapa.

Dentro del programa que crea un paquete de funciones de distribución, se puede considerar el caso de las variables aleatorias mixtas. El algoritmo que genera el valor de una variable aleatoria mixta deberá ser una combinación de los algoritmos creados para continuas y discretas.

# BIBLIOGRAFÍA

**[1]** ALMIRÓN GARCÍA, ADRIÁN. (2012). Simulación del recorrido del equipaje en un aeropuerto con MAXIMA. Proyecto Fin de Carrera dirigido por Gabriel Aguilera Venegas y José Luís Galán García.

**[2]** CAMPOS DIAZ, JOSÉ LUIS.(2014) Simulación de tráfico con semáforos y señales inteligentes usando un CAS. . Proyecto Fin de Carrera dirigido por Gabriel Aguilera Venegas y José Luís Galán García

**[3]** GAVILÁN MONCADA, JOSÉ MANUEL.(2014) Simulación en tiempo acelerado de una red de autobuses metropolitana usando un CAS. Trabajo Fin de Grado dirigido por Gabriel Aguilera Venegas y José Luís Galán García.

**[4]** GALÁN GARCÍA, JOSÉ LUIS. AGUILERA VENEGAS, GABRIEL. RODRIGUÉZ CIELOS, PEDRO. PADILLA DOMÍNGUEZ, YOLANDA. GALÁN GARCÍA, MARÍA ÁNGELES. Random_distributions.mth: Random samples from distributions with DERIVE.

**[5]** Máquina virtual de Java.
http://www.oracle.com/technetwork/Java/Javase/downloads/jdk8-downloads-2133151.html?ssSourceSiteId=otnes

**[6]** RODRÍGUEZ RIOTORTO, M. (2012). Manual de Maxima versión 5.28. [online] Disponible en: http://Maxima.sourceforge.net/docs/manual/es/Maxima.html

**[7]** MCKAIN, D., (2012). Jacomax (Java Connector for Maxima) [online] Disponible en: https://www.wiki.ed.ac.uk/display/Physics/Jacomax

**[8]** JULIA GARCÍA GALISTEO, MARÍA DEL CARMEN MORCILLO AIXELÁ, MANUEL RUIZ CAMACHO. CURSO DE PROBABILIDAD Y ESTADÍSTICA.