Implementación de un algoritmo de sincronización de semáforos usando Inteligencia Artificial

Escobar Zamora Carlos
Camacho Bello Darcy Michelle
Farrera Ramos Javier Antonio

4 de diciembre de 2018

Índice general

Códigos fuente

Introducción

El propósito del siguiente manual es documentar el código del proyecto para facilitar su posterior mantenimiento. Para ello, se incluye el código completo que ha sido previamente auto-documentado siguiendo el estilo de *doxygen*. El diagrama de clases de abajo, se anexa como apoyo visual para comprender mejor las clases que componen el sistema y sus relaciones.

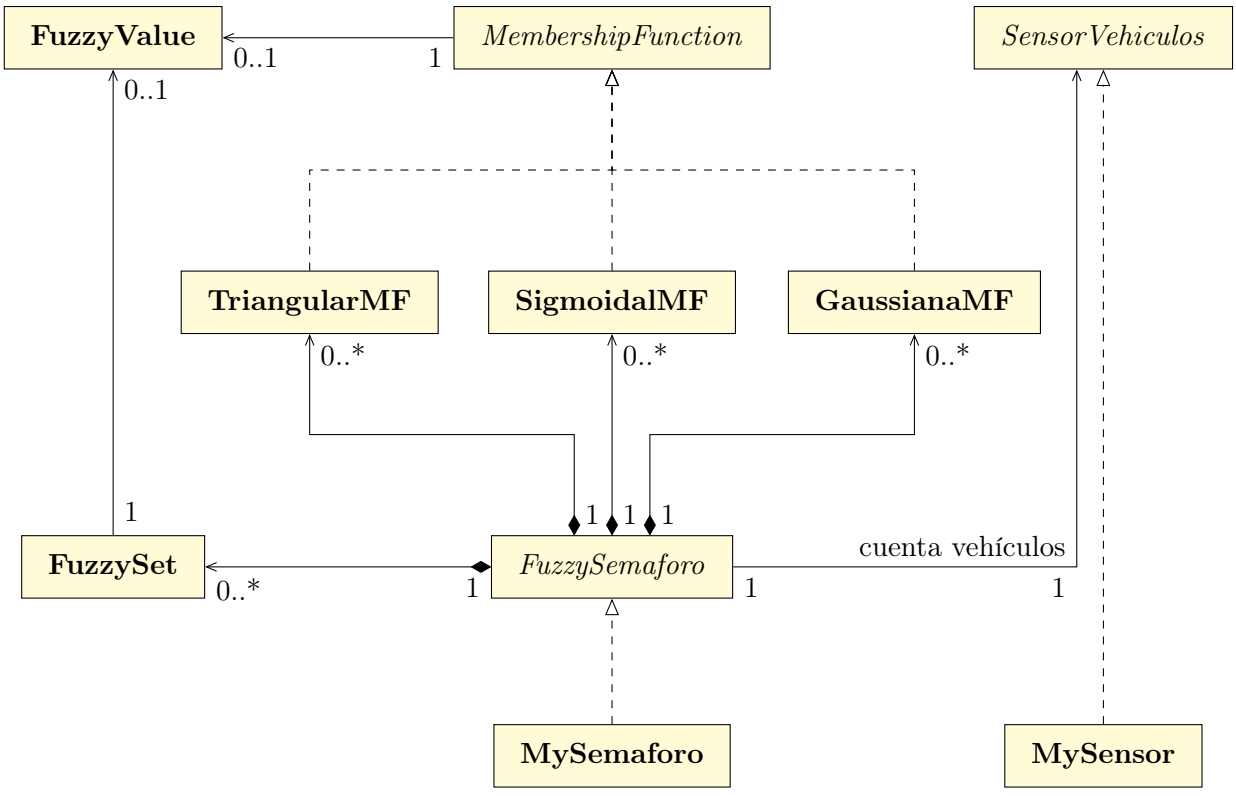

Figura 1: Diagrama de clases del sistema

Capítulo 1

Código Fuente

El proyecto fue construido en el lenguaje de programación **C++**, usando el estándar **C++11**, usando el paradigma **Orietado a Objetos**. Las clases desarrolladas, como es habitual en C++, fueron *separadas en Interfaz e Implementación* en archivos **hpp** y **cpp**, respectivamente (algo considerado como buena práctica en C++).

Debido a que el código se encuentra repartido a lo largo de varios archivos, para facilitar su compresión, se muestra organizado a lo largo de las siguientes secciones de manera conceptual. Además, en cada sección se hace referencia al archivo fuente donde se puede encontrar dicho código.

1.1. Archivo Main

El archivo Main, por convención, suele contener el código encargado de echar a andar todo el sistema, esta no es la excepción. A continuación se muestra el código encargado de configurar y ejecutar el sistema.

Este archivo es de suma importancia ya que es aquí donde se definen los detalles de las implementaciones finales tanto del *semaforo* como del *sensor de vehiculos*; también aquí es donde se configuran las *avenidas*, *fases* y *número de carriles* de la intersección. En pocas

palabras, este es el único archivo de todo el sistema que el usuario deberia modificar (no obstante, es libre de hacer mejoras o ajustes al código donde considere oportuno o necesario)

Debido a la importancia de este archivo y que se explicará de manera detallada. El resto de los archivos que se muestran están igualmente auto-documentados y se agregan algunas explicaciones donde se considera necesario.

Directivas del preprocesador en esta caso indican al pre-procesador que incluya las definiciones de funciones que serán útiles dentro de este archivo fuente.

```
8  // C STanDar Input Output, contiene la definición de las
9  // funciones para mostrar y recibir datos por las entradas
10 // y salidas estandar. Es la versión heredada del lenguage C
11 // pero con seguridad de tipos.
12 #include <cstdio>
13
14 // C Time, contiene funciones para realizar tomas de tiempo
15 // usada para relizar el Benchmark
16 #include <ctime>
17
18 // Vector, incluye la plantilla de clase Vector que permite
19 // usar vectores.
20 #include <vector>
21
22 // Declara que usará el vector del espacio de nombres estandar
23 using std::vector;
24
25
26 // Incluye la cabecera de FuzzySemaforo
27 #include "FuzzySemaforo.hpp"
28
29 // Incluye la cabeceta de SensorVehiculos
30 #include "SensorVehiculos.hpp"
31
32 // Incluye las funciones para graficar la interfaz de usuario
33 #include "semaforoUI.hpp"
```

Código fuente 1.1: Extracto de *Main.cpp*

Definición de la clase de pruebas *MySensor* Esta clase debe ser implementada adecuadamente para proporcionar al sistema el conteo de automóviles de la intersección. Sin embargo, para fines de prueba, se ha implementado de manera que le solicita al usuario que proporcione dichos números a través del teclado.

```
44 class MySensor : public SensorVehiculos
45 {
```

```
46 public:
47
48   /**
49   * Constructor que solo toma el numero de camaras
50   * i.e. el número de números que debe pedir
51   *
52   * @param int  camaras número de camaras a simular
53   */
54   MySensor(int camaras) : num_camaras(camaras)
55   {
56     // Constructor Vacio
57   }
58
59   /**
60   * Función que es llamada por el semaforo para conocer la
61   * cantidad de autos en las avenidas de la intersección. Es
62   * reponsabilidad del usuario final que se implemente
63   * correctamente.
64   *
65   * @return vector< double >
66   */
67   virtual vector<double> read() const
68   {
69     // Crea el vector en el que se guardarán los números
70     vector<double> conteo(num_camaras);
71
72     system("cls");
73
74     // Imprime el prompt para el usuario
75     printf("\nIngresa los valores de prueba para las %d camaras, e.g. <2 8
           8 10>\n>> ", num_camaras );
76
77     // Semáforos en rojo mientras espera al usuario
78     draw_semaforo(4,40, 'r' );
79     draw_semaforo(19,80, 'r');
80     draw_semaforo(34,40, 'r');
81     draw_semaforo(19,1, 'r');
82
83     // Itera el vector
84     for( double &num : conteo )
85     {
86       // Pide al usuario los números
87       scanf("%lf", &num );
88     }
89
90     // devuelve al semáforo el vector con los números
91     return conteo;
92   }
93
94 private:
95   int num_camaras; //! guarda el número de camaras
96
```

```
97 };
```

Código fuente 1.2: Extracto de *Main.cpp*

Definición de la clase de pruebas *MySemaforo* Los detalles de implementación de esta clase también dependen del usuario final. Al delegar a esta clase los detalles, se permite que el usuario implemente el algoritmo en un Arduino usando las funciones disponibles como: DigitalWrite y AnalogWrite; en una Raspberry mediante los puertos GPIO ó en algun PIC.

```
111 class MySemaforo : public FuzzySemaforo
112 {
113 public:
114
115   /**
116   * Constructor de la clase
117   *
118   * Se encarga de incializr la clase base FuzzySemaforo mediante
119   * la sintaxis de inicialización de miembros. Los datos requeridos son:
120   * la configuración del ciclo, el conteo de los carriles por avenida y,
121   * una referencia al sensor de vehiculos.
122   *
123   * @param vector<vector<int>> ciclo Configuración del ciclo del semáforo
124   * @param vector<int>         carriles Número de carriles por avenida
125   * @param SensorVehiculos     &sensor  Referencia al sensor de vehículos
126   */
127   MySemaforo(
128     vector< vector<int>> ciclo,
129     vector<int> carriles,
130     SensorVehiculos &sensor
131   ) : FuzzySemaforo( ciclo, carriles, sensor ), ciclo_semaforo(ciclo)
132   {
133     // Establece el tamaño de la consolas a
134     // 46 filas y 86 columnas
135     console_size(46,86);
136   }
137
138   // Función encargada de realizar las acciones pertinentes para
139   // realizar el cambio de fase, e.g Realizar la secuencia de fases:
140   // 1 Amarillo, 2 Todo rojo, 3 Poner en verde los semáforos de la
141   // fase indicada, 4 Esperar el tiempo indicado.
142   //
143   // Los detalles del como se implementa, e.g. mediante sonidos, leds,
144   // pantallas, etc... depeden del usuario. Para fines de prueba
145   // se ha optado por imprimir el tiempo inferido
146   virtual void set_lights( int fase, double tiempo )
147   {
148     // Imprime la información
149     printf("El tiempo inferido para la fase %d es de %02.02f segundos",
          fase, tiempo);
150     vector<int> config_fase = ciclo_semaforo[ fase ];
```

```
151
152     // Inicia la fase verde
153     draw_semaforo(4,40, (config_fase[0] == VERDE ? 'v' : 'r' ) );
154     draw_semaforo(19,80, (config_fase[1] == VERDE ? 'v' : 'r' ));
155     draw_semaforo(34,40, (config_fase[2] == VERDE ? 'v' : 'r' ));
156     draw_semaforo(19,1, (config_fase[3] == VERDE ? 'v' : 'r' ));
157
158     // Espera el tiempo inferido por el sistema, para fines
159     // de prueba, el tiempo es reducido a la mitad
160     Sleep(tiempo * 100 );
161
162     // Inicia el parpadeo de la fase amarilla
163     for( int i = 0; i < 3; i++ )
164     {
165       draw_semaforo(4,40, (config_fase[0] == VERDE ? 'a' : 'r' ) );
166       draw_semaforo(19,80, (config_fase[1] == VERDE ? 'a' : 'r' ));
167       draw_semaforo(34,40, (config_fase[2] == VERDE ? 'a' : 'r' ));
168       draw_semaforo(19,1, (config_fase[3] == VERDE ? 'a' : 'r' ));
169       Sleep(500);
170       draw_semaforo(4,40, (config_fase[0] == VERDE ? 'n' : 'r' ) );
171       draw_semaforo(19,80, (config_fase[1] == VERDE ? 'n' : 'r' ));
172       draw_semaforo(34,40, (config_fase[2] == VERDE ? 'n' : 'r' ));
173       draw_semaforo(19,1, (config_fase[3] == VERDE ? 'n' : 'r' ));
174       Sleep(500);
175     }
176
177     // Fase todo rojo
178     draw_semaforo(4,40, 'r' );
179     draw_semaforo(19,80, 'r');
180     draw_semaforo(34,40, 'r');
181     draw_semaforo(19,1, 'r');
182     Sleep(500);
183   }
184
185 };
```

Código fuente 1.3: Extracto de *Main.cpp*

Función *main* Los detalles de implementación de esta clase también dependen del usuario final. Al delegar a esta clase los detalles, se permite que el usuario implemente el algoritmo en un Arduino usando las funciones disponibles como: DigitalWrite y AnalogWrite; en una Raspberry mediante los puertos GPIO ó en algun PIC.

```
191 int main(int argc, char const *argv[])
192 {
193   // Instancia un sensor que simula 4 camaras
194   // 1 para cada avenida
195   MySensor mySensor(2);
196
197   // Declara la cantidad de carriles por avenida
```

```
198   // i.e. av 1 = 3 carriles, av 2 = 2 carriles
199   Carril carriles_avenidas = { 1, 1 };
200
201   // Declara la configuración del ciclo del semáforo
202   /*Ciclo ciclo_semaforo = {
203     // fase 1: avenida 1 en verde, el resto en rojo
204     { VERDE, ROJO, ROJO, ROJO  },
205
206     // fase 2: avenida 2 en verde, el resto en rojo
207     {  ROJO, VERDE, ROJO, ROJO },
208
209     // fase 3: avenida 3 en verde, el resto en rojo
210     {  ROJO, ROJO, VERDE, ROJO },
211
212     // fase 4: avenida 4 en verde, el resto en rojo
213     {  ROJO, ROJO, ROJO, VERDE }
214   };*/ Ciclo ciclo_semaforo = { {VERDE,ROJO},{ROJO,VERDE} };
215
216   // Inicializa el semaforo con la configuración previamente
217   // hecha.
218   MySemaforo mySemaforo( ciclo_semaforo, carriles_avenidas, mySensor );
219
220   // Inicia el bucle de cotrol
221   mySemaforo.run();
222
223   return 0;
224 }
```

Código fuente 1.4: Extracto de *Main.cpp*

1.2. Clase *FuzzySemaforo*

La clase FuzzySemaforo se encarga de integrar la inferencia difusa con el algoritmo de sincronización de semáforos. En otras palabras, se encarga de conectar todas las piezas: Sensor, Semáforo y Sistema de inferencia. Además delega los detalles de implementación final mediante la función `set_lights()`

1.2.1. Interfaz de la clase *FuzzySemaforo*:

```
1
2  // Defie la constante simbólica VERDE como 1
3  #define VERDE 1
4
5  // Defie la constante simbólica ROJO como 0
6  #define ROJO 0
7
8  // Define el alias Ciclo como un vector de vectores de tipo int
9  using Ciclo = vector< vector <int> >;
10
11 // Define el alias Carril como un vector de tipo int
12 using Carril = vector <int>;
13
14 /*
15 * Clase FuzzySemaforo
16 *
17 * Implementa el algoritmo de sincronización de semaforos dejando
18 * los detalles de implementacion del sensor y de los semáforos al
19 * usuario final
20 */
21 class FuzzySemaforo{
22 public:
23
24   // Constructor principal
25   FuzzySemaforo( Ciclo _ciclo, Carril _carriles, SensorVehiculos &_sensor
       );
26
27   // Función virtual que a la que se le delega el cambio de fase
28   virtual void set_lights(int, double) = 0;
29
30   // Echa a andar el bucle permanente que controlará los semáforos.
31   void run();
32
33 private:
34
35   /**
36   * Calcula el tiempo en verde.
37   *
```

```
38    * Esta función es una parte funamental de todo el sistema, es aqui
39    * donde se encuentra el Sistema de Inferencia Difusa.
40    */
41    double get_time( int, int );
42
43    // Calcula la media ponderada de los autos por carril
44    double get_media( int, int);
45
46    // Número de fases del ciclo
47    int num_fases;
48
49    // Número de carriles de la intersecćon
50    int num_carriles;
51
52    // Número de avenidas de la interseción
53    int num_avenidas;
54
55    // Fase actual
56    int fase;
57
58    // Tiempo asignado a la fase actual
59    double tiempo;
60
61    // Media ponderada Vehiculos
62    double vehiculos;
63
64    // Media ponderada Congestión
65    double congestion;
66
67    // Referencia al sensor de vehiculos
68    SensorVehiculos &sensor;
69
70    // Fases del ciclo
71    vector< vector<int>> ciclo;
72
73    // Carriles por avenida de la intersección
74    vector<int> carriles;
75
76    // Autos por avenida de la intersección
77    vector<double> autos;
78
79    // Pesos de ponderación por avenida
80    vector<double> pesos;
81 };
```

Código fuente 1.5: Extracto de *FuzzySemaforo.hpp*

1.2.2. Implementación de la clase *FuzzySemaforo*:

```
1
2  #include "FuzzySemaforo.hpp"
3  #include "FuzzySet.hpp"
```

```
4  #include "FuzzyValue.hpp"
5  #include "MembershipFunction.hpp"
6
7  /**
8  * Constructor de la clase
9  *
10 * Para crear un objeto de la clase es necesario proporcionar la
11 * confiuración de las fases del ciclo, la cantidad de carriles por avenida
12 * y una referencia al sensor de vehiculos.
13 *
14 * @param  vector< vector< int > >   _ciclo    matriz de enteros donde
15 * 1 es verde y 0 es rojo, además, cada fila es una fase y cada columna un
16 * semáforo. e.g. una intersección de dos avenidas y dos fases:
17 * ciclo = {
18 * { VERDE, ROJO }, // fase 1, verde en la avenida 1, rojo en la 2
19 * { ROJO, VERDE } // fase 2, verde en la avenida 2, rojo en la 1
20 * }
21 *
22 * @param  vector< int >       _carriles vector con el número de
23 * carriles por avenida. e.g. carriles = { 2, 3 }
24 *
25 * @param  SensorVehiculos&      _sensor   referencia al sensor
26 */
27 FuzzySemaforo::FuzzySemaforo
28 (
29   vector< vector< int > > _ciclo,
30   vector< int >     _carriles,
31   SensorVehiculos&    _sensor
32 )
33 : ciclo(_ciclo), carriles(_carriles), sensor(_sensor)
34 {
35   // Inicializa el numero de avenidas de la interseciión
36   num_avenidas  = carriles.size();
37
38   // Inicializa el numero de fases del semaforo
39   num_fases     = ciclo.size();
40
41   // Iniciliza la suma de carriles en 0
42   num_carriles  = 0;
43
44   // Establece la fase inicial del semaforo
45   fase      = 0;
46
47   // Crea el vector de pesos para ponderar más adelante
48   // la cantidad de carros por avenida
49   pesos = vector<double>( num_avenidas );
50
51   // Calcula el total de carriles de la intersección
52   for( int n : carriles )
53   {
54     num_carriles += n;
55   }
```

```
56
57    // Calcula el valor de ponderación para cada avenida como un cociente
58    // de el número de carriles por avenida sobre el total de carriles de
59    // la intersección
60    for( int n = 0; n <  carriles.size(); ++n)
61    {
62      pesos[ n ] = (static_cast<double>(carriles[n]) / (num_carriles));
63    }
64
65  }
66
67  /**
68  * Inicia el bucle permanente
69  *
70  * Mantiene un buccle perpetuo donde se encunetra el algoritmo de
71  * sincronización de semáforos
72  */
73  void FuzzySemaforo::run()
74  {
75    /**
76      // Variables usadas para medir el tiempo de ejecución
77      // Descomentar en caso de requerirse.
78
79    double    time = 0;
80    unsigned long  tick = 0;
81    unsigned long tock = 0; */
82
83    // Buccle permanente
84    while( true ) {
85
86      // obtiene la catidad de autos en cada avenida
87      autos = sensor.read();
88
89      /**
90        // toma de tiempo inicial
91        // Descomentar en caso de requerirse.
92
93        tick = clock(); */
94
95      // calcula la media ponderada de los vehiculos en las avenidas que
           pasaran a verde
96      vehiculos   = get_media( fase, 1 );
97
98      // calcula la media ponderada de la congestion, esto es la cantidad de
            autos en las avenidas que quedaran en rojo
99      congestion  = get_media( fase, 0 );
100
101     // Infiere el tiempo en verde a asignar mediante Logiga Difusa
102     tiempo = get_time( vehiculos, congestion );
103
104     /**
105       // toma de tiempo inicial
```

```
106       // Descomentar en caso de requerirse.
107
108       tock = clock(); */
109
110     // Estable el cambio de fase
111     set_lights( fase, tiempo );
112
113     // Siguiente fase
114     fase = (fase + 1) % num_fases;
115
116     /*
117        // Calculo e impresión del tiempo de ejecución.
118        // Descomentar en caso de requerirse.
119
120       time = ((double)(tock-tick)/CLOCKS_PER_SEC);
121       printf("tiempo de ejecución: %f\n", tock-tick);*/
122   }
123 }
124
125 /**
126 * Calcula la suma ponderada de los vehiculos en las avenidas
127 *
128 * Mediante la fase actual (fase) y el paso: verde o rojo, determina
129 * la media de ponderada de vehiculos o congestión, respectivamente.
130 *
131 * @param  int   fase  la fase del ciclo a evaluar
132 * @param  int   estado  ROJO o VERDE
133 * @return double
134 */
135 double FuzzySemaforo::get_media( int fase, int estado )
136 {
137   // inicialización de variables
138   double x = 0; //! autos por carril
139   double xy = 0;  //! sumatoria de de x ponderada
140   double y = 0; //! sumatoria de los pesos
141
142   // Itera todas las avenidas
143   for( int n = 0; n < num_avenidas; ++n )
144   {
145     // Si estado es igual a 1 (verde) analiza solo las avenidas que
146     // tendrán fase verde en la fase actual. Si es 0 (rojo) analiza
147     // las que tendrán fase roja en la fase actual.
148     if( ciclo[ fase ][ n ] == estado )
149     {
150       // Calcula autos por carril
151       x = autos[n] / carriles[n];
152
153       // Suma X ponderada por el peso
154       xy += x * pesos[n] ;
155
156       // Suma los pesos
157       y += pesos[n];
```

```
158     }
159   }
160
161   // retorna la suma ponderada
162   return xy / y;
163 }
164
165 /**
166 * Usa lógica difusa para inferir la duración de la fase
167 *
168 * Calcula el tiempo de duración de la fase mediate un Sistema
169 * de Ieferencia Difusa. Esta es la parte central del proyecto.
170 *
171 * @param  int   v   suma ponderada de los vehiculos en las
172 * intersecciones a los que se les cederá el paso
173 *
174 * @param  int   c   suma ponderada de los vehiculos en las
175 * intersecciones que se mantendrán en espera.
176 *
177 * @return double
178 */
179 double FuzzySemaforo::get_time( int v, int c )
180 {
181   // Declara el universo de discuro como un conjunto
182   FuzzySet universo_discurso(0,0.1,80);
183
184
185
186   // Declara los terminos linguisticos de la variable
187   // de entrada Vehiculos
188   SigmoidalMF vehiculos_bajo( -0.8, 4 );
189   GaussianaMF vehiculos_medio( 1.6, 7 );
190   SigmoidalMF vehiculos_alto( 0.8, 10 );
191
192   // Declara los terminos linguisticos de la variable
193   // de entrada Congestión
194   SigmoidalMF congestion_bajo( -0.8, 5);
195   SigmoidalMF congestion_alto( 0.8, 5);
196
197   // Declara los terminos linguisticos de la variable
198   // de salida Tiempo
199   TriangularMF tiempo_minimo(  0, 0, 25);
200   TriangularMF tiempo_bajo( 0, 25, 40);
201   TriangularMF tiempo_medio( 20, 40, 60);
202   TriangularMF tiempo_alto( 40, 60, 80);
203   TriangularMF tiempo_extra( 60, 80, 100);
204
205
206
207   // Declara los conjutos consecuentes de las reglas
208   // difusas
209   FuzzySet salida_vminimo( universo_discurso, tiempo_minimo );
```

```
210   FuzzySet salida_vbajo( universo_discurso, tiempo_bajo );
211   FuzzySet salida_vmedio( universo_discurso, tiempo_medio );
212   FuzzySet salida_valto( universo_discurso, tiempo_alto );
213   FuzzySet salida_vextra( universo_discurso, tiempo_extra );
214
215
216
217   // Declara el conjunto conclusión como la union del las
218   // imlicaciones siguientes:
219   FuzzySet conclusion =
220
221   // si Vehiculos es Bajo -> Tiempo es Minimo
222   (vehiculos_bajo(v)  >> salida_vminimo) &
223
224   // si Vehiculos es Medio y Congestión es Bajo -> Tiempo es Medio
225   ((vehiculos_medio(v)  & congestion_bajo(c)) >> salida_vmedio) &
226
227   // si Vehiculos es Medio y Congestión es Alto -> Tiempo es Bajo
228   ((vehiculos_medio(v)  & congestion_alto(c)) >> salida_vbajo ) &
229
230   // si Vehiculos es Alto y Congestión es Bajo -> Tiempo es Extra
231   ((vehiculos_alto(v)   & congestion_bajo(c)) >> salida_vextra ) &
232
233   // si Vehiculos es Alto y Congestión es Alto -> Tiempo es Alto
234   ((vehiculos_alto(v)   & congestion_alto(c)) >> salida_valto );
235
236
237
238   // Obtiene el valor de salida mediante el metodo de centroide
239   double out = conclusion.get_centroid();
240
241
242   return out;
243 }
```

Código fuente 1.6: Extracto de *FuzzySemaforo.cpp*

1.3. Clase *SensorVehiculos*

Este archivo define la clase SensorVehiculos, el propósito de esta clase es definir la función virtual pura `read()`. La clase FuzzySemaforo recibe una referencia de este tipo para obtener los cantidad de carros de las avenidas. El usuario final debe implementar dicha función.

1.3.1. Definición de la clase *SensorVehiculos*:

```
1  /**
2  * Clase SensorVehiculos
3  *
4  * Esta clase define el metodo read usado por la clase FuzzySemaforo
5  * para obtener el número de carros en la intersección
6  */
7  class SensorVehiculos
8  {
9  public:
10
11   // funcion virtual pura que debe implementar el usuario
12   virtual vector<double> read() const = 0;
13 };
```

Código fuente 1.7: Extracto de *SensorVehiculos.hpp*

1.4. Clase *FuzzySet*

Este archivo define la clase FuzzySet que representa un conjunto difuso. Los conjuntos difusos son una parte central en el proceso de inferencia. Esta clase define las operaciones elementales sobre conjuntos difusos como: implicación, desfusificación y conjunción.

1.4.1. Interfaz de la clase *FuzzySet*:

```
/**
* Clase FuzzySet
*
* Modela un conjunto difuso discreto para relizar los procesos de
* inferencia.
*/
class FuzzySet
{
  // Sobrecarga el operador de inserción de flujo para realizar
  // la implicación
  friend FuzzySet operator>>(const FuzzyValue& value, FuzzySet& set );

  // Sobrecarga el operador AND para realizar la unión de conjuntos
  friend FuzzySet operator&(const FuzzySet&, const FuzzySet& );

public:

  // Constructor principal, crea un conjunto con un dominio definido
  // por un rango y con imagen 0
  FuzzySet( double, double, double );

  // Constructor que crea un conjunto difuso a partir de una función
  // de membresía y otro conjunto.
  FuzzySet( FuzzySet &, MembershipFunction & );

  // Imprime el conjunto difuso en pantalla
  void print( bool cero = false );

  // Defusifica mediante el metodo de centroide
  double get_centroid() const;

private:
  vector<double> set_u; //! imagen del conjunto
  vector<double> set_x; //! dominio del conjunto
};
```

Código fuente 1.8: Extracto de *FuzzySet.hpp*

1.4.2. Implementación de la clase *FuzzySet*:

```
/*
* Sobrecarga del operador de iserción de flujo
*
* El operador es sobrecargado como un operador de implicación modus ponens
    , g.e.
* A → B puede ser modelado como  A >> B
* devuelve un conjunto conclusión
*
* @param  FuzzyValue& value   antecedente
* @param  FuzzySet& set   consecuente
* @return FuzzySet
*/
FuzzySet operator>>(const FuzzyValue& value, FuzzySet& set )
{
  // Itera sobre todo el dominio del conjunto
  for( size_t i = 0; i < set.set_u.size(); i++ )
  {
    // Corta el conjunto a la altura del valor de membresía del
       antecedente
    set.set_u[ i ] = (set.set_u[ i ] < value) ? set.set_u[ i ] :
       static_cast<double>(value);
  }

  // devuelve el conjunto ya cortado
  return set;
}

/**
* Sobrecarga el operador AND
*
* Devuelve un conjunto resulato de la unión de los dos conjuntos
* que le son proporcionados
*
* @param  FuzzySet& setl operando izquierdo
* @param  FuzzySet& setr operando derecho
* @return FuzzySet
*/
FuzzySet operator&(const FuzzySet& setl, const FuzzySet& setr )
{
  // Crea una copia de uno de los conjuntos
  FuzzySet setc( setl );

  // Itera ambos conjuntos (operando izq. y der.)
  for (int i = 0; i < setl.set_u.size(); ++i)
  {
    // Realiza la unión conservando el mayor de los valores de
    // membresía para cada valor del universo de discurso (operador max)
    setc.set_u[ i ] = (setl.set_u[i] > setr.set_u[i]) ? setl.set_u[i] :
       setr.set_u[i];
  }

  // Devuelve el conjunto resultado
```

```
49   return setc;
50 }
51
52 /**
53 * Defusifica mediante el método de centroide
54 *
55 * Aplica el metodo de centroide sobre los valores de conjunto
56 * y devuelve el el valor certero.
57 *
58 * @return double
59 */
60 double FuzzySet::get_centroid() const
61 {
62   double ux_x = 0;
63   double ux = 0;
64   // Rezaliza la sumatoria
65   for( size_t i = 0; i < set_u.size(); ++i )
66   {
67     // u(x) * x
68     ux_x += set_x[ i ] * set_u[ i ];
69     // u(x)
70     ux += set_u[ i ];
71   }
72
73   return ux_x / ux;
74 }
75
76 /**
77 * Constructor
78 *
79 * Crea un conjunto difuso discreto con membresía 0 donde el universo de
80 * discuro se encuentra en el rango [begin, end] con step como intervalo
81 * e.g. FuzzySet(2,2,10) -> {0/2, 0/4, 0/6, 0/8, 0/10}
82 *
83 * @param  double  begin   primer elemento (inclusive)
84 * @param  double  step  intervalo o salto
85 * @param  double  end   último elemento
86 */
87 FuzzySet::FuzzySet( double begin, double step, double end )
88 {
89   // Calcula el total de elementos que tendrá el cojunto
90   double elements = (end - begin) / step +1;
91
92   // inicializa la imagen del conjunto
93   set_u = vector<double>(elements);
94
95   // inicializa el dominio del conjunto
96   set_x = vector<double>(elements);
97
98   for( size_t i = 0; i < elements; ++i )
99   {
100    // asigna los valores del dominio
```

```
101     set_x[ i ] = begin + ( i * step );
102     // asigna 0 como valor de membresía
103     set_u[ i ] = 0;
104   }
105 
106 }
107 
108 /**
109 * Constructor
110 *
111 * Crea un conjunto difuso discreto, resultado de aplicar la funcion
112 * de membresía mf a los elementos del conjunto fset.
113 *
114 * @param  FuzzySet&     begin   conjunto de entrada
115 * @param  MembershipFunction& step  función de membresía
116 */
117 FuzzySet::FuzzySet( FuzzySet &fset, MembershipFunction &mf )
118 {
119   // inicializa el dominio y la imagen del conjunto
120   // con el tamaño exacto del conjunto entrada
121   set_u = vector<double>( fset.set_u.size() );
122   set_x = vector<double>( fset.set_x.size() );
123 
124   for (int i = 0; i < set_x.size(); ++i)
125   {
126     // copia el dominio
127     set_x[ i ] = fset.set_x[ i ];
128 
129     // aplica la función de membresía a la imagen
130     set_u[ i ] = mf( set_x[ i ] );
131   }
132 }
133 
134 /**
135 * Imprime el conjunto
136 *
137 * Imprime el conjunto difuso usando la notación propia de los
138 * conjuntos difusos discretos: x / u(x)
139 *
140 * @param  bool  cero  indica si deben imprimir los valores u(x)=0
141 */
142 void FuzzySet::print( bool cero )
143 {
144   printf("( ");
145 
146   // Si no está vacio
147   if( set_x.size() > 0 )
148   {
149     // imprime si el primer elemento es diferente de cero
150     if( set_u[ 0 ] != 0 || cero )
151       printf( "%.2f/%.2f", set_x[ 0 ], set_u[ 0 ] );
152 
```

```
153     // recoore el resto de elementos
154     for( size_t item = 1; item < set_x.size(); ++item )
155     {
156       // imprime si u(x) es diferen de cero
157       if( set_u[ item ] != 0 || cero )
158         printf( ", %.2f/%.2f", set_x[ item ], set_u[ item ] );
159     }
160   }
161
162   printf(" )");
163 }
```

Código fuente 1.9: Extracto de *FuzzySet.cpp*

1.5. Clase *FuzzyValue*

Las funciones de membresía devuelven el valor de fusificación como un objeto de esta clase, permitiendo así usar los operadores de unión, intersección y complemento sobre los resultados obtenidos.

1.5.1. Interfaz de la clase *FuzzyValue*:

```
/**
* Clase FuzzyValue
*
* Es una envoltura para un valor de membresía de tipo double. De esta
* manera se define las operadores que se pueden utilizar sobre los
* objetos de esta clase.
*/
class FuzzyValue
{

public:

  // Constructor por defecto.
  FuzzyValue( const double = 0 );

  // Operador de conversión double
  operator double() const;

  // Sobrecarga el operador AND
  FuzzyValue operator&( const FuzzyValue &rvalue ) const;

  // Sobrecarga el operador OR
  FuzzyValue operator|( const FuzzyValue &rvalue ) const;

  // Sobrecarga el operador Complemento
  FuzzyValue operator~() const;

private:
  double x; //! valor de membresía
};
```

Código fuente 1.10: Extracto de *FuzzyValue.hpp*

1.5.2. Implementación de la clase *FuzzyValue*:

```
/**
* Constructor por defecto
*
* Crea un objeto FuzzyValue a partir de un valor de tipo fundamental
```

```
5 *
6 * @param  double  _x
7 */
8 FuzzyValue::FuzzyValue( const double _x ) : x(_x)
9 {
10   // Constructor vacio
11 }
12
13 /**
14 * Operador de conversioón double
15 *
16 * Permite utilizar el operador de conversión de tipo, ya sea de
17 * manera explícita o implícita.
18 *
19 * @return double valor de mebresía
20 */
21 FuzzyValue::operator double() const
22 {
23   // para convertirlo a double basta con devolver el valor de mebresía
24   return x;
25 }
26
27 /**
28 * Sobrecarga el operador AND
29 *
30 * Permite utilizar el operador & (AND) y devuelve el resulatdo de
31 * efectuar el operador min sobre los operandos
32 *
33 * @param  FuzzyValue rvalue operando derecho
34 * @return FuzzyValue
35 */
36 FuzzyValue FuzzyValue::operator&( const FuzzyValue &rvalue ) const
37 {
38   // Compara los valores de mebresía actual y el del operador
39   // derecho y devuelve el más pequeño.
40   return (x < rvalue.x) ? x : rvalue.x;
41 }
42
43 /**
44 * Sobrecarga el operador OR
45 *
46 * Permite utilizar el operador | (OR) y devuelve el resulatdo de
47 * efectuar el operador max sobre los operandos
48 *
49 * @param  FuzzyValue rvalue operando derecho
50 * @return FuzzyValue
51 */
52 FuzzyValue FuzzyValue::operator|( const FuzzyValue &rvalue ) const
53 {
54   // Compara los valores de mebresía actual y el del operador
55   // derecho y devuelve el más grande.
56   return (x > rvalue.x) ? x : rvalue.x;
```

```
57 }
58
59 /**
60 * Sobrecarga el operador ~
61 *
62 * Permite utilizar el operador ~ (Complemento) y devuelve el
63 * resulatdo de efectuar el complemento a 1
64 *
65 * @return FuzzyValue
66 */
67 FuzzyValue FuzzyValue::operator~() const
68 {
69   // Devuelve el complemento a uno del valor de mebresía
70   return 1 - x;
71 }
```

Código fuente 1.11: Extracto de *FuzzyValue.cpp*

1.6. Clase *MembershipFunction*

Una parte fundamental del proyecto son las funciones de membresía, estas se encuentran en el paquete MembershipFunction. La interfaz y la implementación de las clases, se encuentran separadas en los archivos *MembershipFunction.hpp* y *MembershipFunction.cpp*, respectivamente.

En seguida se muestran los extractos de ambos archivos correspondientes a cada clase.

1.6.1. Definición de la clase *MembershipFunction*

La clase MembershipFunction es meramente una interfaz que define una función virtual pura que deberá ser implementada por las clases derivadas.

El resto de clases que hacen uso de funciones de membresía, lo hacen a través de referencias del tipo de esta clase: *MembershipFunction*. Y cualquier función de membresía que el usuario desee definir debe extender de dicha clase. El propósito de esta clase es declarar la *sobrecarga del operador '()'* como una *función virtual pura*. De esta manera se permite el procesamiento polimórfico de las funciones.

Código de la definición de la clase *MembershipFunction*:

```
1  /**
2  * Clase MembershipFunction
3  *
4  * Define una única función. Tal función es bastante especial puesto que
5  * sobrecarga el operador () como una función virtual pura. Toda clase
6  * que represente una función de membresía definida por el usuario, debe
7  * extender de esta clase y debe implementar dicho operador.
8  */
9  class MembershipFunction
10 {
11 public:
12
13   /**
14   * Función virtual pura operator()
15   *
16   * Sobrecarga del operador (), que debe ser sobreescrito por las clases
       derivadas
17   *
18   * @param  double    x   valor a fuzificar
19   * @return FuzzyValue    clase que representa un valor difuso
```

```
20    */
21    virtual FuzzyValue  operator()  (double x) const = 0;
22 };
```

Código fuente 1.12: Definición de la clase *MembershipFunction*

1.7. Clase *TriangularMF*

Esta clase, como su nombre lo sugiere, representa una función de membresía triangular. Para poder ser usada como tal dentro del marco de trabajo, debe extender de la clase anteriormente mencionada.

1.7.1. Interfaz de la clase *TriangularMF*:

```
/**
* Interfaz de la clase TriangularMF
*
* Representa una función de membresía triangular
*/
class TriangularMF : public MembershipFunction
{
public:

  // Constructor de TriangularMF
  TriangularMF( double, double, double);

  // Calcula el grado de membresía de un valor dado
  virtual FuzzyValue operator()(double ) const;

private:
  double a; //! vertice A de la función triangular
  double b; //! vertice B de la función triangular
  double c; //! vertice C de la función triangular
};
```

Código fuente 1.13: Interfaz de la clase *TriangularMF*

1.7.2. Implementación de la clase *TriangularMF*:

```
/**
* Constructor de TriangularMF
*
* Crea una funcion triangular a partir de sus tres vertices: A, B y C.
*
* @param double _a  vertice A
* @param double _b  vertice B
* @param double _c  vertice C
*/
TriangularMF::TriangularMF(double _a, double _b, double _c ) : a(_a), b(_b
    ), c(_c)
{
  //Constructor vacio
}
```

```
14
15
16 /**
17 * Sobrecarga del operador ()
18 *
19 * Calcula el valor de membresia en base a la funcion definida
20 * por los vertices A, B y C.
21 *
22 * @param  double  x   valor discreto a fuzificar.
23 * @return FuzzyValue  valor de membresia u(x).
24 */
25 FuzzyValue TriangularMF::operator()(double x) const
26 {
27   double u;
28   if     ( x <= a)      u = 0;
29   else if ( x > a && x < b )  u = (x - a) / (b - a);
30   else if ( x == b )       u = 1;
31   else if ( x > b && x < c )  u = (c - x) / (c - b);
32   else             u = 0;
33   return u;
34 }
```

Código fuente 1.14: Implementación de la clase *TriangularMF*

1.8. Clase *SigmoidalMF*

El trabajo realizado con esta función es análogo a la anterior.

1.8.1. Interfaz de la clase *SigmoidalMF*:

```
/**
* Interfaz de la clase SigmoidalMF
*
* Representa una función de membresía Sigmoidal
*/
class SigmoidalMF : public MembershipFunction
{
public:

  // Constructor de SigmoidalMF
  SigmoidalMF( double, double);

  // Calcula el grado de mebresía de un valor dado
  virtual FuzzyValue operator()(double) const;

private:
  double a; //! valor de la pendiente de la función
  double x0;  //! punto de cruce
};
```

Código fuente 1.15: Interfaz de la clase *SigmoidalMF*

1.8.2. Implementación de la clase *SigmoidalMF*:

```
/**
* Constructor de SigmoidalMF
*
* Crea una función Sigmoidal a partir de su pendiente y punto de cruce
*
* @param double a   determina la pendiente de la curva
* @param double x0  determina el punto de cruce
*/
SigmoidalMF::SigmoidalMF(double _a, double _x0) : a(_a), x0(_x0)
{
  //Constructor vacio
}

/**
* Sobrecarga del operador ()
*
* Calcula el valor de membresia en base a la funcion definida
* por la pendienete y punto de cruce
*
```

```
20 * @param  double  x   valor discreto a fuzificar.
21 * @return FuzzyValue  valor de membresia u(x).
22 */
23 FuzzyValue SigmoidalMF::operator()(double x) const
24 {
25   double u;
26   u = 1 / ( 1 + exp( (-1 * a )*(x-x0) ));
27   return u;
28 }
```

Código fuente 1.16: Implementación de la clase *SigmoidalMF*

1.9. Clase *GaussianaMF*

El trabajo realizado con esta función es igualmente análogo a la primera.

1.9.1. Interfaz de la clase *GaussianaMF*:

```
/**
* Interfaz de la clase GaussianaMF
*
* Representa una función de membresía Gaussiana
*/
class GaussianaMF : public MembershipFunction
{
public:

  // Constructor de SigmoidalMF
  GaussianaMF( double, double);

  // Calcula el grado de mebresía de un valor dado
  virtual FuzzyValue operator()(double x) const;

private:
  double a; //! pendiente de la función
  double x0;  //! punto de cruce
};
```

Código fuente 1.17: Interfaz de la clase *GaussianaMF*

1.9.2. Implementación de la clase *GaussianaMF*:

```
/**
* Constructor de GaussianaMF
*
* Crea una función Gaussiana a partir de su pendiente y punto de cruce
*
* @param double a   determina la pendiente de la curva
* @param double x0  determina el punto de cruce
*/
GaussianaMF::GaussianaMF(double _a, double _x0) : a(_a), x0(_x0)
{
  //Constructor vacio
}

/**
* Sobrecarga del operador ()
*
* Calcula el valor de membresia en base a la funcion definida
* por la pendienete y punto de cruce
*
```

```
20 * @param  double  x   valor discreto a fuzificar.
21 * @return FuzzyValue  valor de membresia u(x).
22 */
23 FuzzyValue GaussianaMF::operator()(double x) const
24 {
25   double u;
26   u = exp( -0.5 * pow( ((x-x0)/a) ,2));
27   return u;
28 }
```

Código fuente 1.18: Implementación de la clase *GaussianaMF*