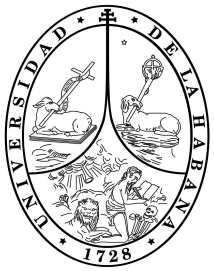


Universidad de La Habana

Facultad de Matemática y Computación

Proyecto de Compilación + Intenteligencia Artificial + Simulación

Simulador de un Jefe Técnico de MotoGP

Autores:

Arnel Sánchez Rodríguez Grupo: C312
arnelsanchezrodriguez@gmail.com

Samuel Efraín Pupo Wong Grupo: C312
s.pupo@estudiantes.matcom.uh.cu

Darián Ramón Mederos Grupo: C312
darianrm24@gmail.com

2021-2022

Índice

1. Motociclismo de Velocidad

El motociclismo de velocidad es una modalidad deportiva del motociclismo disputada en circuitos de carreras pavimentados. Las motocicletas que se usan pueden ser prototipos, es decir desarrolladas específicamente para competición, o derivadas de modelos de serie (en general motocicletas deportivas) con modificaciones para aumentar las prestaciones. En el primer grupo entran las que participan en el Campeonato Mundial de Motociclismo, y en el segundo las Superbikes, las Supersport y las Superstock.

Las motocicletas deben presentar una serie de características como son estabilidad, alta velocidad (tanto en recta como en paso por curva), gran aceleración, gran frenada, fácil maniobrabilidad y bajo peso.

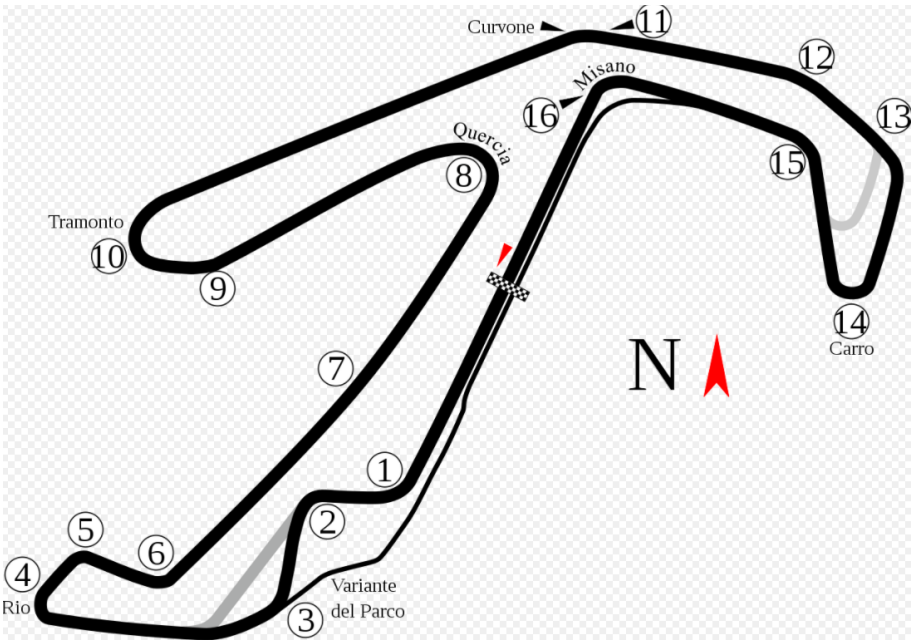
2. Estructura del Paddock

El **Jefe Técnico** de cada estructura se configura como una personalidad de bastante importancia dentro de un box, pues es quién se encarga de dirigir y controlar que todo funcione como un excelente engranaje que gane carreras. De igual importancia es la telemetría dentro de un box en MotoGP. Al fin y al cabo, los **Ingenieros Telemétricos** son las personas que se encargan de analizar, leer y comprender todos los datos proporcionados por el piloto, así como transmitirselos en boca al protago-

nista. Se trata de una figura de la que depende mucha de la información acerca de cualquier cambio realizado en la moto o asumir los puntos más fuertes de sus pilotos. Los **Mecánicos** también desempeñan un papel fundamental a la hora de construir la máquina perfecta.

3. Pista

Se utilizará como referencia el circuito de Misano, Misano World Circuit Marco Simoncelli, autódromo localizado en la fracción de Santa Mónica, comuna de Misano Adriático (provincia de Rímini), región de Emilia-Romaña, Italia.



4. Definición del Problema

Existirán varios pilotos con sus respectivas motos, las cuáles difieren entre sí en cuanto a sus prestaciones. Cada piloto posee su propio método de manejo, siendo algunos más cuidadosos y otros más agresivos. La pista se encuentra influenciada por el accionar del clima, puesto que no es lo mismo el manejo durante un día soleado que bajo la lluvia. Por tanto, el resultado de un piloto se verá condicionado por su moto, su modo de conducción y el clima.

Sin embargo, durante la carrera las condiciones pueden variar y es necesario realizar los ajustes necesarios para que el piloto mejore su rendimiento, los cuales se harán al pasar de una seccion a otra de la pista o al finalizar cada vuelta. Esto podrá hacerse utilizando un lenguaje imperativo, mediante el uso de palabras claves para que el piloto no necesite analizar situaciones complejas y pueda concentrarse en pilotar de la forma mas eficiente posible.

De esta manera, la simulación de la carrera será dinámica, puesto que entre las vueltas podrán existir variaciones provocadas por los ajustes propuestos por el Jefe Técnico, el cuál podrá ser una persona o una IA.

5. Definición del Lenguaje

5.1. Introducción a PySharp (P#)

5.1.1. Hello world!

```
1    method void main() {
2      print("Hello World!");
3    }
```

Los archivos de P# suelen tener la extensión de archivo .pys.

5.1.2. Estructura del Programa

Los conceptos organizativos clave en P# son programas, tipos y miembros. Los programas P# constan de un archivo fuente. Los programas declaran tipos y miembros. Las motocicletas y los motociclistas son ejemplos de tipos. Los métodos y propiedades son ejemplos de miembros.

5.1.3. Tipos y Variables

En P# solo existen tipos de valor, no hay de referencia. Por tanto, todas las variables contienen directamente sus datos, cada una tiene su propia copia y no es posible que las operaciones en una afecten a otra.

Categoría	Tipo	Descripción
Tipos	Tipos Simples	Entero con signo: int
		Punto flotante IEEE: double
		Booleanos: bool
		Cadenas Unicode: string
	Tipos que aceptan valores NULL	Extensiones de todos los demás tipos de valor con un valor nulo

5.1.4. Expresiones

Las expresiones se construyen a partir de operandos y operadores. Los operadores de una expresión indican qué operaciones aplicar a los operandos. Los ejemplos de operadores incluyen +, -, * y /. Los ejemplos de operandos incluyen literales, variables y expresiones.

Categoría	Expresión	Descripción
Primaria	x(...)	Invocación de método
	x[...]	Acceso a matrices e indexadores
Unaria	+x	Identidad
	-x	Negación
	!x	Negación lógica
Multiplicativa	x * y	Multiplicación
	x / y	División
	x % y	Resto
	x ** y	Exponenciación
Aditiva	x + y	Adición y concatenación de strings
	x - y	Substracción
Relacionales	x <y	Menor que
	x >y	Mayor que
	x <= y	Menor o igual que
	x >= y	Mayor o igual que
Igualdad	x == y	Igual
	x != y	Distinto
Condicionales AND	x && y	Evalúa y si y sólo si x es verdadera
Condicionales OR	x || y	Evalúa y si y sólo si x es falsa
Condicionales XOR	x ^ y	
Asignación	x = y	Asignación
	x op= y	Asignación compuesta; los operadores admitidos son *= /= %= **= += -= &&= ||= ^=

5.1.5. Declaraciones

Las acciones de un programa se expresan mediante declaraciones. P# admite varios tipos diferentes de declaraciones, algunas de las cuales se definen en términos de declaraciones integradas.

Un **bloque** permite escribir múltiples declaraciones en contextos donde se permite una sola declaración. Un bloque consta de una lista de declaraciones escritas entre

los delimitadores{y}.

Las sentencias de **declaración** se utilizan para declarar variables, valga la redundancia.

```
1    method void example() {
2      int a = 1;
3    }
```

Las **declaraciones de expresión** se utilizan para evaluar expresiones. Las expresiones que se pueden usar como declaraciones incluyen invocaciones de métodos, asignaciones que usan = y los operadores de asignación compuesta.

```
1    method void example() {
2      int a = 1;
3      print(a + 2);
4    }
```

La **instrucción de selección** se utiliza para seleccionar una de varias declaraciones posibles para su ejecución en función del valor de alguna expresión. Este es el caso de la sentencia **if**.

```
1    method void example(int a) {
2      if (a < 5) {
3        print(a);
4      }
5      else {
6        print(a % 5);
7      }
8    }
```

La **instrucción de iteración** se utiliza para ejecutar repetidamente una instrucción incorporada. Este es el caso de la instrucción **while**.

```
1    method void example(int a) {
2      while (a > 5) {
3        print(a);
4        a -= 1;
5      }
6    }
```

Las **sentencias de salto** se utilizan para transferir el control. En este grupo están las declaraciones de **break**, **continue** y **return**.

```
1    method int example() {
2      while (true) {
3        x = input()
4        if (x == "x") {
5          continue;
```

```
6           }
7           elif (x == "") {
8             break;
9           }
10          print(x);
11        }
12        return 0;
13    }
```

5.1.6. Tipos especiales

Los **tipos especiales** son los elementos más importantes de P#, constituyen estructuras de bloques compuestas por acciones (métodos). Un tipo proporciona una definición para casos de, por ejemplo, motociclistas o motocicletas. Su declaración comienza con un encabezado que especifica qué tipo se va a crear y el nombre que se le dará a esta instancia. El encabezado va seguido del cuerpo del tipo, que consiste en una lista de declaraciones de miembros escritas entre los delimitadores{y}.

```
1     rider Rossi() {
2       method int example() {
3         return 0;
4       }
5
6       method void curves() {
7         ...
8       }
9       ...
10    }
```

5.1.7. Métodos

Un **método** es un miembro que implementa un cálculo o acción que se puede realizar por tipo. Los métodos tienen una lista (posiblemente vacía) de **parámetros**, que representan valores o referencias de variables pasadas al método, y un tipo de retorno, que especifica el tipo de valor calculado y devuelto por el método. El tipo de retorno de un método es nulo si no devuelve un valor.

5.1.8. Parámetros

Los **parámetros** se utilizan para pasar valores o referencias de variables a métodos. Los parámetros de un método obtienen sus valores reales de los **argumentos**

que se especifican cuando se invoca el método. Las modificaciones de un valor de parámetro no afectan el argumento que se pasó para el parámetro.

5.1.9. Cuerpo del método y variables locales

El cuerpo de un método especifica las declaraciones que se ejecutarán cuando se invoca el método. El cuerpo de un método puede declarar variables que son específicas de la invocación del método. Estas variables se denominan variables locales. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y un valor inicial.

5.1.10. Operadores

Un **operador** es un miembro que define el significado de aplicar un operador de expresión particular. Se pueden definir solamente operadores binarios.

Operadores Binarios:

```
1     3+5;
2     true && falge;
```

5.1.11. Análisis Léxico

input
: input_element* new_line
| directive
;
input_element
: whitespace
| comment
| token
;
Terminadores de línea
new_line
: '<Caracter de retorno (U+000D)>'
| '<Caracter de avance de línea (U+000A)>'
;
whitespace
: '<Cualquier personaje con clase Unicode Zs>'
| '<Caracter de tabulación horizontal (U+0009)>'
;

Comentarios

comment

: '#' comment_section '#'
;

Tokens

token

: identifier
| keyword
| literal
| operator_or_punctuator
;

Identificadores

identifier

: '<Un identificador que no es una palabra clave>'
| identifier_start_character identifier_part_character*
;

identifier_start_character

: letter_character
| '<Caracter guión bajo (U+005F)>'
;

identifier_part_character

: letter_character
| decimal_digit
| '<Caracter guión bajo (U+005F)>'
;

letter_character

: uppercase_letter_character
| lowercase_letter_character
;

uppercase_letter_character

: 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
| 'I' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q'
| 'R' | 'S' | 'T' | 'V' | 'X' | 'Y' | 'Z'
;

lowercase_letter_character

: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
| 'i' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
| 'r' | 's' | 't' | 'v' | 'x' | 'y' | 'z'

;
decimal_digit
: '0' | '1' | '2' | '3' | '4'
| '5' | '6' | '7' | '8' | '9'
;
Palabras Claves
keyword
: 'bool'
| 'break'
| 'continue'
| 'double'
| 'elif'
| 'else'
| 'false'
| 'if'
| 'int'
| 'method'
| 'null'
| 'return'
| 'string'
| 'true'
| 'void'
| 'while'
| 'bike'
| 'rider'
| 'brakes'
| 'max_speed'
| 'weight'
| 'chassis_stiffness'
| 'speed'
| 'tyres'
| 'cornering'
| 'step_by_line'
;
Literales
literal
: boolean_literal
| integer_literal

| double_literal
| string_literal
| null_literal
;
Literales Booleanos
boolean_literal
: 'true'
| 'false'
;
Literales Enteros
integer_literal
: decimal_digit
;
Literales flotantes
double_literal
: decimal_digit+ '.' decimal_digit+
;
Literales de Cadenas
string_literal
: '"' string_literal_character* '"'
;
string_literal_character
: '<Cualquier caracter, excepto "(U+0022)'
;
Literales Nulos
null_literal
: 'null'
;
Operadores y signos de puntuación
operator_or_punctuator
: '{'
| '}'
| '['
| ']'
| '('
| ')'
| '.'
| ','

| ':'
| ';'
| '+'
| '-'
| '*'
| '/'
| '%'
| '**'
| '='
| '<'
| '>'
| '&&'
| '||'
| '=='
| '!='
| '<='
| '>='
| '+='
| '-='
| '*='
| '/='
| '%='
| '**='
| '&&='
| '||='
| '^='
;

Directivas

directive

: 'include' '<Nombre_del_archivo.pys>' ';'

;

5.2. Explicación de la Implementación

Para crear nuestra gramatica nos apoyamos en los lenguajes Python y CSharp, de ahi el nombre de nuestro DSL: PySharp.

Cuando se recibe el código, este se tokeniza. Luego el método *split_linese* recibe los tokens y los convierte en líneas, las cuales son pasadas al parser.

El parser que decidimos desarrolar fue el LL. Lo primero que hicimos fue las producciones, donde decidimos como serían correctas sintácticamente nuestras líneas, pues dichas producciones generan todas las posibles cadenas válidas para nuestro lenguaje y no existen cadenas que son generadas por nuestra gramática que no pertenezcan a nuestro lenguaje. De esta forma con una gramática válida empezamos el proceso de parsing. Nuestro parser necesita un método $hacer\_first$ primeramente para hacer los first de cada cadena posible de nuestra gramática, luego llamamos un método auxiliar $calcular\_first\_restantes$ el cual tiene la función de calcular los first de los no terminales que aún no lo tienen calculado, nos hace falta guardar los first de los no terminales porque los necesitamos para hallar los follows en el método $hacer\_follow$. Luego de invocar a $hacer\_follow$, se utiliza el método auxiliar $completar\_follows$, que se encarga de satisfacer la regla de los follows que dice que el follow de la cabeza de la producción es subconjunto del follow del último no terminal, si el último no terminal puede ser el último elemento de la producción.

Teniendo los first y los follows construimos la tabla LL(1) mediante el método $construir\_tabla\_LL$, al tener la tabla ya podemos comprobar que nuestra gramática no es ambigua, pues siempre existe solo una producción que aplicar. Luego, creamos el método parsear al que hay que pasarle todas las líneas de nuestro código una por una y este realiza la comprobacion sintáctica, mismo método donde vamos a ir creando nuestro AST para luego hacer el chequeo semántico. Para crear el AST utilizamos métodos como $CreaNododExpresion$, $CreaNodoCondicion$, $CreaNododFuncion$, $EligeTipoDdeclaracion$, entre otros, declarados en la clase Parser.

Nuestro AST tiene un nodo por cada declaración que se puede realizar en el código. Tiene un nodo para una definición de función, una definición de variables, redefinición de variables, If, WHile, Rider, Bike, Return. En cada uno de estos nodos excepto Rider y Bike sí existe un ámbito como es el caso del nodo If, el While y la definición de función, cada uno de estos nodo tiene como atributo un tipo de nodo Program, el cuál posee una lista de declaraciones y por lo tanto en él se pueden guardar la lista de declaraciones que se haga en el ámbito.

Explicada la estructura del AST, pasamos al chequeo semántico sobre este. Hacemos 3 recorridos sobre el AST, el primero para validar cada nodo. Un nodo es válido si todo lo que tiene guardado en sus atributos que es dependiente del contexto puede ser utilizado desde ese contexto y de la forma que se quiere. Decimos esto porque por ejemplo las variables solo se pueden redefinir en el contexto en que fueron definidas. Cada vez que creamos una función o un tipo creamos un contexto que responde a dicho ámbito. Todo lo que se defina en dicho ámbito pertenece a su contexto, no importa si se define dentro de un While o dentro de un If. En resumen, los contextos en nuestro programa funcionan como en python con la particularidad

de que no tenemos variables globales, si se quiere redefinir una variable, debe hacerse en el contexto donde fue definida y solo se crea un nuevo contexto cuando se crea una funcion fuera de un tipo o cuando se crea un tipo. Destacar que las funciones definidas dentro de los tipos solo tienen el mismo context, y no se le pueden pasar parámetros.

Volviendo al AST, hacemos una segunda pasada, en esta pasada verificamos los tipos, donde en los nodos en que hay expresiones inducimos su tipo. Una expresión puede ser una expresión aritmética, un bool, o un string, podemos incluir variables y llamado a función. La tercera pasada la hacemos para evaluar nuestros nodos. Si encontramos un nodo Def_Fun no lo evaluamos, una definición de función se evalua cuando se llama a la función.

Cuando se crea un tipo especial se importan las variables que puede tener este en la simulación, dentro de un tipo Bike solo se puede redefinir una función, la función $select\_configuration$ que debe ser void, aunque dentro de la funcion si se pueden redefinir las variables de Bike. En este caso el objetivo, de $select\_configuration$ es seleccionar el tipo de gomas dadas las características de la moto y el ambiente y por tanto modificar la variable tires del contexto del tipo, la cuál utilizará el simulador.

Dentro de un tipo Rider existen en el contexto, igual que en Bike, variables que pertenecen a Rider que fueron importadas desde la simulación, en este caso se pueden redefinir dos de ellas *cornering* y $step\_by\_line$ que posteriormente serán pasadas al simulador. Estas variables tienen mucha influencia en la simulación ya que son la habilidad en recta y en curva de un piloto. Son variables enteras que lo maximo que pueden ser es 10, en caso de que se entre un valor mayor se supondra que se quiso entrar el maximo de habilidad y la variable será igual a 10. En cuanto a las funciones se podrán definir $select\_action$ que debe retornar un valor entero y es la encargada de elegir que acción realizar, para hacer este método podemos tener en cuenta las características del piloto que pueden ser llamadas y están actualizadas. También podemos definir la funcion $select_acceleration$ la cual debe ser void y su función debe ser actualizar la aceleracion de un agente.

5.3. Conexion Simulacion - Compilacion

El resultado del DSL si se hacen los 3 recorridos del AST sin error, son dos listas, una con todos los pilotos que fueron creados y otra con todas las motos. A partir de estas listas se crean los pilotos y las motos en la simulacion. Los pilotos que no fueron creados en el código ejecutan sus métodos como normalmente. En el caso de un piloto que fue creado desde el DSL se ejecuta la función definida en el lenguaje, pero antes de ejecutarla se actualiza el contexto de la funcion para que dicha función pueda

apoyarse en la situación actual. Luego de la ejecución de la función dependiendo que función se ejecutó, se importa a la simulación la variable que se quiere desde el contexto del método.

6. Simulación

Nos propusimos simular una carrera de MotoGP contrareloj, compuesta por los agentes (tuplas moto-piloto) y el ambiente (tupla clima-pista). Una vez estos son generados, se simula cada sección de la pista, calculando la influencia que ejerce el ambiente sobre el agente y las propias interacciones del agente consigo mismo, el tiempo que le demora a cada piloto en circular dicha sección y la aceleración y la acción que escogerá dicho piloto en esa sección, ya sea mediante una función redefinida desde el DSL o por una Inteligencia Artificial.

6.1. Clima

En cada sección de la pista se hacen pequeñas variaciones a los parámetros del clima, pero no al estado de este. Para darle un poco más de complejidad al sistema, luego de cada vuelta sí se modifica completamente el estado del clima y sus parámetros. Estas simulaciones se realizan mediante la distribución normal, que nos permite generar una variable aleatoria pero sin cambios tan bruscos en el promedio de sus casos.

6.2. Interacciones

Como habíamos mencionado anteriormente, existen varias interacciones entre un agente y el ambiente, y dentro de un propio agente:

- Si la temperatura baja se enfrían los neumáticos y se pierde adherencia al pavimento
  - Disminuye el paso por curva
  - Disminuye el paso por recta
  - Aumenta la probabilidad de caerse de la moto
- Si la temperatura aumenta se calientan los neumáticos y se gana adherencia al pavimento, se desgastan más rápido
  - Aumenta el paso por curva

    - Aumenta el paso por recta
    - Disminuye la probabilidad de caerse de la moto
    - Aumenta la probabilidad de romper el motor de la moto
    - Aumenta la probabilidad de reventar los neumáticos de la moto
- Si la visibilidad baja
    - Disminuye el paso por curva
    - Disminuye el paso por recta
    - Aumenta la probabilidad de caerse de la moto
- Si la visibilidad aumenta
    - Aumenta el paso por curva
    - Aumenta el paso por recta
    - Disminuye la probabilidad de caerse de la moto
- Si la humedad aumenta se pierde adherencia al pavimento
    - Disminuye el paso por curva
    - Disminuye el paso por recta
    - Aumenta la probabilidad de caerse de la moto
- Si la humedad baja se gana adherencia al pavimento
    - Aumenta el paso por curva
    - Aumenta el paso por recta
    - Disminuye la probabilidad de caerse de la moto
- Si el viento de de frente
    - Disminuye el paso por curva
    - Disminuye el paso por recta
    - Aumenta la probabilidad de reventar los neumáticos de la moto
- Si el viento de de espaldas
    - Aumenta el paso por curva

        - Aumenta el paso por recta
        - Aumenta la probabilidad de caerse de la moto
        - Aumenta la probabilidad de romper el motor de la moto
    - Si el viento de de lado
        - Aumenta el paso por curva
        - Aumenta el paso por recta
        - Aumenta la probabilidad de caerse de la moto
    - Si el estado del clima es soleado aumenta la temperatura
    - Si el estado del clima es lluvioso aumenta la humedad
    - Si el estado del clima es nublado condiciones perfectas para correr
    - Si disminuye la rigidez del chasis tiene más torsión
        - Aumenta el paso por curva
        - Disminuye el paso por recta
    - Si aumenta la rigidez del chasis tiene menos torsión
        - Disminuye el paso por curva
        - Aumenta el paso por recta
    - Si disminuye la frenada se frena en más distancia
        - Aumenta el paso por curva
        - Disminuye el paso por recta
    - Si aumenta la frenada se frena en menos distancia
        - Disminuye el paso por curva
        - Aumenta el paso por recta

De acuerdo a las interacciones anteriores se debe escoger el neumático lo más preciso posible y este provocará una combinación entre estos factores.

7. Inteligencia Artificial

Como nuestro proyecto se basa en simular una carrera de MotoGP donde hay n corredores, aquellos que no hayan sido diseñados mediante DSL son controlados por IA, el comportamiento de la cual varía a partir de las condiciones del ambiente que lo rodea, las características de su moto y la experiencia del propio piloto.

Para la implementación de la IA se utilizan Sistemas Expertos generados mediante el uso de un procedimiento declarativo/imperativo, apoyándonos en la biblioteca PyKE de Python.

Se emplea como base de conocimiento el conjunto de variables que forman parte de la simulación y que influyen en su desempeño, como:

- las características de la moto (velocidad, frenos, chasis)
- el estado del clima (soleado, nublado o lluvioso)
- la intensidad y el sentido del viento
- la humedad y la temperatura del ambiente
- la sección de pista que se está corriendo (curva o recta)
- la velocidad máxima alcanzable en la sección
- las habilidades del piloto (en rectas y curvas)
- entre otros

7.1. Configuración de la moto

La primera heurística es la encargada de escoger la mejor configuración de la moto. De este modo, atendiendo a las condiciones iniciales de la carrera (que serán los hechos), la IA podrá tomar decisiones atendiendo a las reglas definidas. Al ser declarativas, las determinaciones tomadas se obtendrán mediante match hechos-reglas. Los hechos del sistema se presentan mediante una lógica difusa y son procesados para ser analizados:

```
1     rainy(True)
2     humidity(False)
3     windy(False)
4     wind_direction(2)
```

Las reglas del sistema se formulan del modo:

```
    soft
    use select_tires(0)
    when
    moto_facts.windy(True)
    moto_facts.wind_direction(3)

    medium_1
    use select_tires(1)
    when
    moto_facts.windy(False)

    medium_2
    use select_tires(1)
    when
    moto_facts.windy(True)
    moto_facts.wind_direction(2)

    hard
    use select_tires(2)
    when
    moto_facts.windy(True)
    moto_facts.wind_direction(1)
```

Luego, consultando los resultados obtenidos por las librerías de PyKE, mediante Python de manera imperativa será posible la interacción de la estructura antes expuesta con la simulación de la carrera. Atendiendo al ejemplo mostrado de selección de gomas, el sistema declarativo de PyKE devuelve el cálculo de una ecuación:

$$valor\_tipo\_goma + valor\_goma = selección$$

El resultado obtenido se utiliza para generar el tipo de las gomas seleccionadas, mediante el uso de un Enum en Python:

```
    class Tires(Enum):
    Slick_Soft = 0
    Slick_Medium = 1
    Slick_Hard = 2
    Rain_Soft = 3
    Rain_Medium = 4
```

7.2. Selección de acciones

Una segunda heurística será la encargada de escoger la acción que debe ejecutar el piloto, atendiendo a las condiciones de la carrera:

- aumentar/disminuir/mantener velocidad
- doblar
- ir a los pits
- combinaciones de todas las anteriores

(Ejemplo: aumentar velocidad + doblar + ir a los pits)

En este caso, se utiliza el mismo método expuesto en la configuración de la moto: declarativo/imperativo con el uso de PyKE. Se analizan los parámetros de velocidad, sección de la pista y estado del clima:

```
speed(3)
curve(False)
tires(False)
rainy(True)
humidity(False)
```

Y se decide tomándolos en cuenta:

```
speed_up
use select_action(0)
when
action_facts.speed(3)

keep_speed
use select_action(1)
when
action_facts.speed(2)

brake
use select_action(2)
when
action_facts.speed(1)
```

Asimismo, se calcula mediante una ecuación el valor de la decisión tomada:

$$acción\_velocidad + doblar + pits = acción$$

```
class AgentActions(Enum):
SpeedUp = 0
KeepSpeed = 1
Brake = 2
SpeedUp_Turn = 3
KeepSpeed_Turn = 4
```

```
7       Brake_Turn = 5
8       SpeedUp_Pits = 6
9       KeepSpeed_Pits = 7
10      Brake_Pits = 8
11      SpeedUp_Turn_Pits = 9
12      KeepSpeed_Turn_Pits = 10
13      Brake_Turn_Pits = 11
```

7.3. Selección de la aceleración

Una tercera heurística es la encargada de escoger la mejor aceleración en una sección dada de la pista, con el objetivo de alcanzar la mayor velocidad posible sin causar un accidente que obligue a abandonar la carrera. Primeramente, se calcula la aceleración máxima alcanzable, tomando en cuenta que no se exceda la velocidad máxima admitida por la moto y la sección que se recorre.

$$a_{max} = (V_{max}^2 - V^2)/(2 * X)$$

Luego, se establece un sistema de penalizaciones que disminuyen dicha aceleración si no se poseen las condiciones óptimas del ambiente (clima, humedad, temperatura) y del piloto (destreza en curvas y rectas). El resultado obtenido es incorporado a la simulación para continuar la carrera. De su valor depende mucho si el piloto obtiene un buen tiempo o sufre un accidente que lo saque de la competencia.