UA

Universidad
de Alicante

ESCUELA POLITÉCNICA SUPERIOR

RAZONAMIENTO Y REPRESENTACIÓN DEL CONOCIMIENTO

Curso académico 2024 / 2025
Semestre 1
Grado en Ingeniería en Inteligencia Artificial
Grupo 1

Profesora: Diego Viejo Hernando
Alumno: Santiago Álvarez Geanta

Universitat d'Alacant
Universidad de Alicante

Septiembre - Octubre 2024

ÍNDICE

Librerías utilizadas

En nuestro fichero **ExpertSystem.py** he utilizado librerías como **math** para realizar todos los cálculos matemáticos necesarios. Además, en mi caso, he utilizado la librería **pandas** para realizar operaciones con archivos .csv para almacenar varios resultados de distintos parámetros.

ExpertSystem: __init__(self)

Principalmente, nuestro ExpertSystem se caracteriza porque utiliza funciones matemáticas, que más adelante explicaremos, para devolver los valores de la velocidad lineal y la velocidad angular. Junto a estas funciones matemáticas utilizamos una constante que nos ayudará a ajustar dicha función. Algo más es que nuestro programa se caracteriza porque trata a las rectas de forma distinta que a los triángulos, para ello utilizamos el atributo **tipoSegmento**.

También contamos con todos los distintos tipos de **checkpoints** utilizados como podrían ser:

- **inicioAlcanzado**: Valor booleano que indica cuándo se ha alcanzado el inicio del segmento
- **objetivoAlcanzado**: Valor booleano que indica cuándo se ha alcanzado el final del segmento

Por último creamos un atributo en el que guardaremos la información del robot recibida en el método **tomarDecision()**.

```
5   class ExpertSystem:
6       def __init__(self) -> None:
8           """
9           Variables constantes
10          ----------
11          KR_lineal: constante que determina la velocidad lineal en una recta
12          KR_angular: constante que determina la velocidad angular en una recta
13          KR_deteccion: determina a partir de que momento el robot detecta un objetivo como alcanzado en una recta
14
15          KT_lineal: constante que determina la velocidad lineal en un triángulo
16          KT_angular: constante que determina la velocidad angular en un triángulo
17          KT_deteccion: determina a partir de que momento el robot detecta un objetivo como alcanzado en un triángulo
18
19          """
20
21          self.segmentoObjetivo = None    # Inicializamos valores en P1Launcher
22          self.objetivoAlcanzado = False  # Final del segmento
23          self.inicioAlcanzado = False    # Inicio del segmento (Punto cercano: Recta, PuntoMedio: Triángulo)
24          self.tipoSegmento: str = "recta" # Comenzamos como una recta siempre
25
26          self.KR_lineal = 10.0
27          self.KR_angular = 0.07
28          self.KR_deteccion = 2.42
29
30          self.KT_lineal = 10.4
31          self.KT_angular = 0.02
32          self.KT_deteccion = 0.9
33
34          self.poseRobot = ("xRob", "yRob", "angRob", "vR", "wR")
```

class ExpertSystem -> Método __init__()

tomarDecision(self, poseRobot)

```python
def tomarDecision(self, poseRobot:tuple) -> tuple:
    """
    Determina que velocidad lineal y angular tiene el robot en el instante de
    tiempo actual y en función de las características del segmentoObjetivo.

    Return
    ------
    Devuelve una tupla con la velocidad lineal y velocidad angular.
    """
    self.poseRobot = poseRobot

    if self.tipoSegmento == "recta":
        return self.decisionRecta()
    else:
        return self.decisionTriangulo()
```

tomarDecision(self, poseRobot)

El objetivo del siguiente método consiste en devolver una tupla de dos valores (velocidad lineal, velocidad angular). Para ello he decidido dividir dichos cálculos dependiendo del tipo de segmento.

Segmento Recta

Si el robot se sitúa en un segmento recta, su objetivo inicial será el punto más cercano de la recta respecto al robot. Si ya se ha alcanzado dicho punto, nuestro robot se dirigirá al punto situado en la recta a una distancia **t** entre el robot y el punto final. Posteriormente, calcularemos el ángulo que nos falta para llegar al objetivo correspondiente.

Será importante definir puntos de detección para cada objetivo. Si el objetivo es el punto inicial no tendremos ninguna restricción, por lo que he decidido utilizar la fórmula del MRUA para despejar x:

$$v^2 = vo^2 + 2ax$$

$$-vo^2 / 2a \text{ (acelaración negativa = -1)}$$

$$vo^2 / 2$$

Como se mencionó anteriormente, nos apoyaremos de una constante, en este caso **KR_DETECCION**. Cuando el robot alcancé dicho rango marcaremos nuestro checkpoint inicial como alcanzado.

En cambio, si el objetivo es el punto interpolado nos centraremos únicamente en la distancia al punto final. En este caso estamos restringidos a utilizar un rango máximo de valor **0.5**.

Para el cálculo de la velocidad lineal he utilizado la siguiente función matemática:

KR_lineal / sqrt(x=error_angular)

Se recomienda un valor de 10 para entrar en el rango:

[≈0.75, inf]

He utilizado esta función porque a mayor error angular menor es la velocidad lineal. Además utilicé la raíz cuadrada para amortiguar el peso que tiene la variable error angular. El parámetro sirve simplemente para definir un rango que se ajuste a nuestras necesidades.

Para el cálculo de la velocidad angular he utilizado la misma fórmula, solo que en este caso no utilizamos la inversa de la raíz:

sqrt(x=error_angular) * KR_angular

```python
    # Primer Objetivo: Punto más cercano recta AB respecto al robot R
    # Siguientes Objetivos: Punto (interpolado a una distancia de t)situado en AB entre el punto final B y el robot R
    objetivo = self.puntoCercano() if not self.inicioAlcanzado else self.puntoInterpolado(t=1/9)

    # Diferencia de angulos entre el angObjetivo y el angRobot
    error_angular = self.calcularErrorAngular(objetivo)


#### DETECCIÓN OBJETIVO INICIAL (PUNTO CERCANO)
    distanciaObjetivo = math.sqrt(abs((xRob - objetivo[0]) ** 2 + (yRob - objetivo[1]) ** 2))

    # No es necesario utilizar 0.5 como distancia de detección para el puntoCercano()
    # Utilizamos la fórmula del MRUA para despejar x: v**2 = vo**2 + 2ax
    # Despejando: -vo**2 / 2a (acelaración negativa = -1) -> vo**2 / 2
    if not self.inicioAlcanzado and distanciaObjetivo <= (vRob**2 - self.KR_deteccion)/2:
        self.inicioAlcanzado = True

#### DETECCIÓN OBJETIVO FINAL
    xFinal, yFinal = self.segmentoObjetivo.getFin()
    distanciaFinal = math.sqrt((xRob - xFinal) ** 2 + (yRob - yFinal) ** 2)

    # Sí es necesario utilizar 0.5 como distancia de detección para el puntoFinal()
    if distanciaFinal <= 0.5:
        self.setobjetivoAlcanzado()

#### V_LINEAL

    # Función matemática utilizada: KR_lineal / sqrt(x=error_angular)
    # Donde la constante aumenta el valor de la Y respecto de X
    # Se recomienda un valor de 10 para entrar en rango: [≈0.75, inf]
    # Cuando el error_angular es 0 otrogamos una velocidad máxima [3]
    try:

        v_lineal = self.KR_lineal/math.sqrt(abs(error_angular))
    except (ZeroDivisionError, ValueError):

        v_lineal = 3

#### W_ANGULAR

    # Función matemática utilizada: sqrt(x=error_angular) * KR_angular
    # Donde la constante aumenta el valor de la Y respecto de X
    # Se recomienda un valor de 0.07 para entrar en el rango de [0, 1]
    # No buscamos w_angulares muy altas en rectas

    try:
        w_angular =  math.sqrt(error_angular) * self.KR_angular if error_angular >= 0

    except (ValueError):
        w_angular = 0 # utilizado por si ocurre algún problema inesperado

    return v_lineal, w_angular
```

Segmento Triángulo
El objetivo inicial en este segmento vendrá dado por el método del objeto segmento **.getMedio()**. Si ya se ha alcanzado dicho punto, nuestro robot se dirigirá al punto final. Posteriormente, calcularemos el ángulo que nos falta para llegar al objetivo correspondiente.

Será importante definir puntos de detección para cada objetivo. Si el objetivo es el punto inicial no tendremos ninguna restricción, por lo que he utilizado una fórmula similar a la de las rectas pero en este caso **KT_DETECCION** viene acompañado con la velocidad actual del robot.

En cambio, si el objetivo es el punto interpolado nos centraremos únicamente en la distancia al punto final. En este caso estamos restringidos a utilizar un rango máximo de valor **0.5**.

Para el cálculo de la velocidad lineal he utilizado la siguiente función matemática:

3 -((error angular - 120) * KT_lineal)

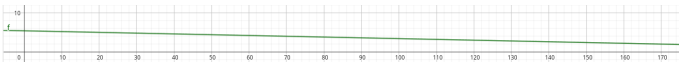
El comportamiento para la función es el siguiente. La velocidad siempre tendrá el valor de 3, pero cuando el error angular es mayor de 120 el valor de la función comienza a disminuir a una velocidad que viene dada por la constante **KT_lineal**.

Para el cálculo de la velocidad angular simplemente utilizo una fórmula lineal.

error_angular * KT_angular

Finalmente, me gustaría añadir que todas las fórmulas utilizadas fueron creadas con anterioridad en **GeoGebra** (antes de probar en Python), y que fueron modificadas varias veces antes de obtener su versión final.

```python
    # Primer Objetivo: Punto Medio del triángulo
    # Segundo Objetivo: Fin del triangulo
    objetivo = self.segmentoObjetivo.getMedio() if not self.inicioAlcanzado else self.segmentoObjetivo.getFin()

    # Diferencia de angulos entre el angObjetivo y el angRobot
    error_angular = self.calcularErrorAngular(objetivo)


#### DETECCIÓN OBJETIVO INICIAL (PUNTO CERCANO)
    if not self.inicioAlcanzado:

        distanciaObjetivo = math.sqrt((xRob - objetivo[0]) ** 2 + (yRob - objetivo[1]) ** 2)

        # En este caso hemos añadido, además de la constante de Deteccion
        # La velocidad del robot para ser más precisos cuándo alcanza el objetivo
        if distanciaObjetivo < abs(vRob**2 - self.KT_deteccion*vRob)/2:
            self.inicioAlcanzado = True

#### DETECCIÓN OBJETIVO FINAL
    xFinal, yFinal = self.segmentoObjetivo.getFin()
    distanciaFinal = math.sqrt((xRob - xFinal) ** 2 + (yRob - yFinal) ** 2)

    if distanciaFinal <= 0.5:
        self.setobjetivoAlcanzado()
#### V_LINEAL

    # La velocidad siempre tendrá el valor de 3
    # cuando el error angular es mayor de 120 el valor de la función comienza a disminuir
    # La velocidad de decrecimiento viene dada por la constante KT_lineal
    v_lineal = 3 - ((error_angular-120) * self.KT_lineal)


#### W_ANGULAR
    w_angular = self.KT_angular * error_angular
```



velocidad lineal (triángulo)

FuzzyExpert

El desarrollo del **FuzzyExpert** utiliza los mismos métodos que el **ExpertSystem.** Por tanto solo voy a inferir en aquello que considero importante.

En primer lugar tendremos que instalar la librería **fuzzy_expert**. De esta librería solamente utilizaremo la parte para definir las variables, las reglas y el DescompositionalInference.

```python
from fuzzy_expert.variable import FuzzyVariable
from fuzzy_expert.rule import FuzzyRule
from fuzzy_expert.inference import DecompositionalInference
```

Comenzaremos definiendo nuestras variables, cada variable tendrá un rango (universe_range) con un valor mínimo y un valor máximo.

Además, contamos con unos términos, cada término recibe como argumento una lista de tuplas (valor, grado de pertenencia al término).

En mi caso he definido las siguiente variables:

- **Error angular**
- **Distancia**
- **v_lineal**
- **w_angular**

```
##################################### variables
    self.variables: dict = {
        "error_angular": FuzzyVariable(
            universe_range=(0, 180),
            terms={
                "grande": [(85, 0), (120, 0.33), (140, 0.66), (180, 1)],
                "medio": [(10, 1), (15, 0.66), (30, 0.33), (90, 0)],
                "pequeño": [(2, 1), (4, 0.66), (6, 0.33), (10, 0)],
            },
        ),
        "distancia": FuzzyVariable(
            universe_range=(0.5, 100),
            terms={
                "lejos": [(3, 0), (7, 0.33), (20, 0.66), (25, 1)],
                "cerca": [(0, 1), (2, 1), (3, 0.66), (4, 0)],
            },
        ),
        "v_lineal": FuzzyVariable(
            universe_range=(0, 3),
            terms={
                "rapido": [(2, 0), (2.5, 0.66), (3, 1), (4, 1)],
                "medio":  [(1, 1), (1.5, 0.66), (1.75, 0.4), (2, 0)],
                "lento":  [(0, 1), (0.1, 0.66), (0.5, 0.33), (0.8, 0)],

            },
        ),
        "w_angular": FuzzyVariable(
            universe_range=(0, 3),
            terms={
                "rapido": [(1.2, 0), (1.4, 0.33), (1.8, 0.66), (2.2, 1)],
                "medio": [(1.4, 1), (1, 0.66), (0.6, 0.33), (0.4, 0)],
                "lento": [(0, 1), (0.02, 0.66), (0.05, 0.33), (0.1, 0)],

            },
        ),
    }
```

El siguiente paso, ya creadas las variables, es crear las reglas. Cada regla contiene una premisa y una consecuencia. Las premisas reciben una lista de tuplas del siguiente formato (OPERADOR LÓGICO, variable, término de variable). La consecuencia es aquello que obtendremos en caso de que se cumpla la regla, el formato es parecido al anterior sólo que esta vez obtendremos solamente (variable, término de variable).

```
######################################## reglas
    self.rules: list[object] = [

    ### LEJOS Y BIEN ORIENTADO
        FuzzyRule(
            premise=[
                ("error_angular", "pequeño"),
                ("AND", "distancia", "lejos"),
            ],
            consequence=[("v_lineal", "rapido"), ("w_angular", "lento")],
        ),
    ### LEJOS Y NORMAL ORIENTADO
        FuzzyRule(

            premise=[
                ("error_angular", "medio"),
                ("OR", "error_angular", "grande"),
                ("AND", "distancia", "lejos"),
            ],
            consequence=[("v_lineal", "medio"), ("w_angular", "rapido")],
        ),
    ### LEJOS Y MAL ORIENTADO
        FuzzyRule(
            premise=[
                ("error_angular", "grande"),
                ("AND", "distancia", "lejos"),
            ],
            consequence=[("v_lineal", "medio"), ("w_angular", "rapido")],
        ),

    ### CERCA Y BIEN ORIENTADO
        FuzzyRule(
            premise=[
                ("error_angular", "pequeño"),
                ("AND", "distancia", "cerca"),
            ],
            consequence=[("v_lineal", "medio"), ("w_angular", "lento")],
        ),

    ### CERCA Y NORMAL ORIENTADO
        FuzzyRule(
            premise=[
                ("error_angular", "medio"),
                ("AND", "distancia", "cerca"),
            ],
            consequence=[("v_lineal", "medio"), ("w_angular", "medio")],
        ),

    ### CERCA Y MAL ORIENTADO
        FuzzyRule(
            premise=[
                ("error_angular", "grande"),
                ("AND", "distancia", "cerca"),
            ],
            consequence=[("v_lineal", "lento"), ("w_angular", "rapido")],
        ),
    ]
```

Por último tendremos que definir nuestro modelo utilizando el método **DecompositionalInference()**. En mi caso utilicé el mismo que se utiliza en el Tutorial 1 de la documentación de la correspondiente librería. Definido el modelo utilizado podremos utilizar el método **model()**. Este método recibirá el diccionario de variables, la lista de reglas y todos los valores actuales de las variables definidas. De esta forma, según el valor de las variables introducidas obtendremos un resultado u otro según hayamos definido en nuestras reglas. Es importante recalcar que este método nos devuelve una tupla en el que el primer elemento corresponde con el diccionario **consequence** definido en las reglas y como segundo elemento un float con valor 1.0.

```
######################################## evaluación de reglas
    model = DecompositionalInference(
        and_operator="min",
        or_operator="max",
        implication_operator="Rc",
        composition_operator="max-min",
        production_link="max",
        defuzzification_operator="cog",
    )

######################################## resultado
    result = model(
        variables=self.variables,
        rules=self.rules,
        error_angular=abs(error_angular),
        distancia=distancia,
        v_lineal=vR,
        w_angular=wR,
    )
```


UA

Universidad de Alicante

Universidad de Alicante