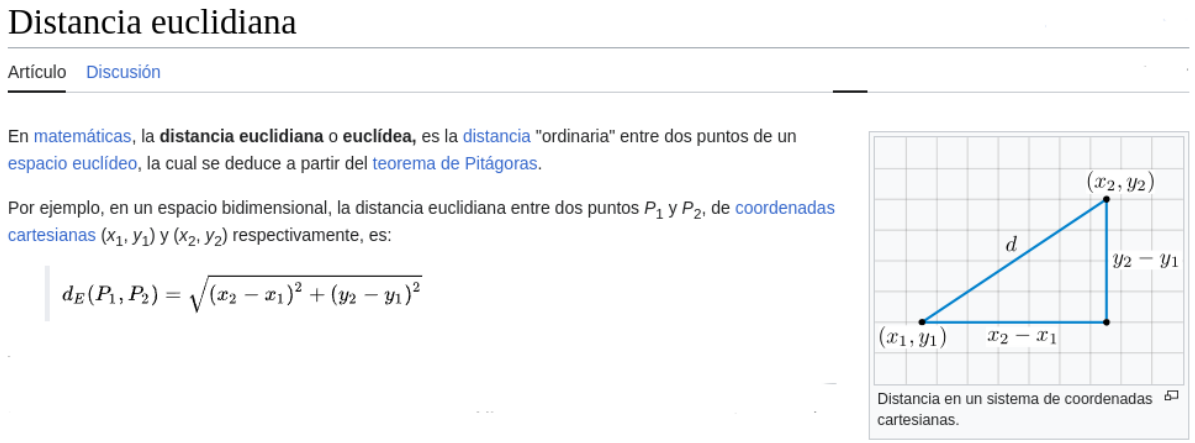
Matemáticas para ROS2

Distancia euclidiana



Distancia euclidiana

Artículo Discusión

En matemáticas, la **distancia euclidiana** o **euclídea,** es la distancia "ordinaria" entre dos puntos de un espacio euclídeo, la cual se deduce a partir del teorema de Pitágoras.

Por ejemplo, en un espacio bidimensional, la distancia euclidiana entre dos puntos P_1 y P_2, de coordenadas cartesianas (x_1, y_1) y (x_2, y_2) respectivamente, es:

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Distancia en un sistema de coordenadas cartesianas.

Ejemplo utilizado en tortuga_controller2.cpp

```
double dist_x{tortuga_objetivo_.x - pose_.x};
double dist_y{tortuga_objetivo_.y - pose_.y};
double distancia{
                std::sqrt(dist_x * dist_x + dist_y * dist_y)};
```

Este código calcula la distancia entre dos puntos en un plano 2D utilizando la fórmula de la distancia euclidiana. Los dos puntos son `tortuga_objetivo_` y `pose_`.

Aquí está el desglose:

1. `**double dist_x{tortuga_objetivo_.x - pose_.x};**`: Calcula la diferencia en la coordenada x entre `tortuga_objetivo_` y `pose_`.

2. `**double dist_y{tortuga_objetivo_.y - pose_.y};**`: Calcula la diferencia en la coordenada y entre `tortuga_objetivo_` y `pose_`.

3. `**double distancia{std::sqrt(dist_x * dist_x + dist_y * dist_y)};**`: Calcula la distancia euclidiana entre `tortuga_objetivo_` y `pose_` utilizando el teorema de Pitágoras. La distancia euclidiana es la longitud de la línea recta entre dos puntos en un espacio, y se calcula como la raíz cuadrada de la suma de los cuadrados de las diferencias en las coordenadas x e y.

Analizando la velocidad y orientación en tortuga_controller2.cpp

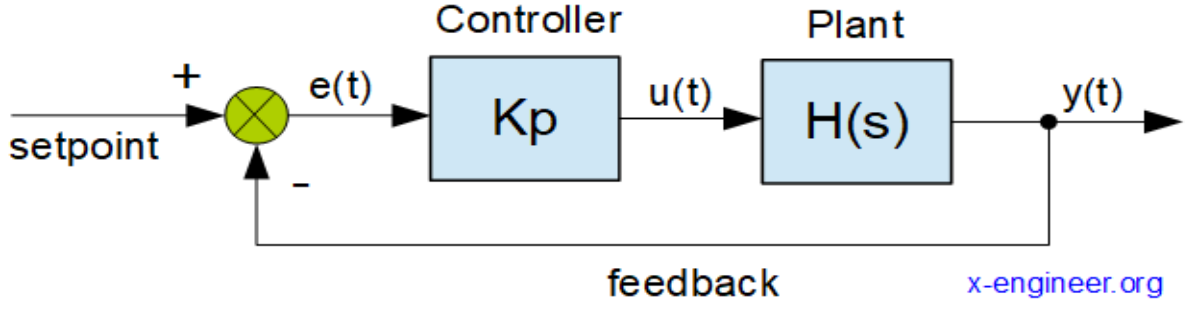
El código a analizar es el siguiente:

```
if (distancia > 0.5){
   // Posicion
   msg.linear.x = 2 * distancia;

   // Orientacion
   double angulo_direccion{std::atan2(dist_y, dist_x)};
   double angulo_diferencia{angulo_direccion - pose_.theta};

   if(angulo_diferencia > M_PI) angulo_diferencia -= 2 * M_PI;
   else if(angulo_diferencia  < -M_PI)
                               angulo_diferencia  += 2 * M_PI;

   msg.angular.z = 6 * angulo_diferencia;
}
```

P Controller

Un controlador de retroalimentación es un tipo de sistema de control que compara una salida deseada (también llamada punto de ajuste) con la salida real (medida por un sensor) y ajusta el sistema (a través de una señal de control) en consecuencia para lograr la salida deseada. Esto se logra mediante un bucle de retroalimentación que monitorea continuamente la salida del sistema y la compara con la salida deseada, luego ajusta las entradas del sistema para acercar la salida al valor deseado. Esto permite que el sistema se ajuste automáticamente a cambios en las condiciones y mantenga una salida estable incluso cuando la salida se desvía del punto de ajuste debido a perturbaciones externas o internas.

Existen varios tipos de controladores de retroalimentación, que incluyen **controladores proporcional (P)**, **integral (I)** y **derivativo (D)**, o una combinación de ellos (**PI**, **PD** o **PID**), cada uno de los cuales utiliza métodos diferentes para ajustar el sistema según el bucle de retroalimentación.



Un **controlador proporcional (P-controller)** es un tipo de sistema de control de retroalimentación que ajusta la señal de control de un sistema *u(t)* en proporción al error *e(t)*, calculado entre el punto de ajuste y la salida real *y(t)*. Este tipo de controlador se utiliza comúnmente en sistemas de control en lazo cerrado para asegurar que la salida del sistema permanezca lo más cerca posible del punto de ajuste.

La planta, que es el sistema controlado, puede ser cualquier tipo de sistema, por ejemplo, un sistema mecánico, eléctrico o hidráulico. En simulaciones, la planta se puede describir como un conjunto de ecuaciones diferenciales, matrices de espacio de estados o funciones de transferencia.

P Controller en el código
Para el caso que nos ocupa, la velocidad lineal del robot (`msg.linear.x`) es directamente proporcional a la distancia al objetivo.

El factor de proporcionalidad (en este caso, 2) determina cuánto cambia la velocidad lineal en respuesta a un cambio en la distancia. Un factor de proporcionalidad de 2 significa que la velocidad lineal será el doble de la distancia al objetivo.

$$u(t) = Kp \cdot e(t)$$

```
msg.linear.x = 2 * distancia;
```

El valor exacto del factor de proporcionalidad depende de lo que quieras lograr. Si quieres que el robot se mueva más rápido, puedes usar un factor de proporcionalidad mayor. Si quieres que se mueva más despacio, puedes usar un factor de proporcionalidad menor. En la práctica, tendrías que ajustar este valor basándote en pruebas y observaciones del comportamiento del robot.

Calcular la orientación

```
double angulo_direccion{std::atan2(dist_y, dist_x)};
double angulo_diferencia{angulo_direccion - pose_.theta};
```

El código utiliza la función **`atan2`** para calcular el ángulo. La función **`atan2`** devuelve el ángulo en radianes entre el eje x positivo y el punto dado (dist_x, dist_y). Esta es una forma común de calcular el ángulo en un sistema de coordenadas 2D.

La fórmula general para **`atan2`** es: `atan2(y, x)`

Donde **y** es la distancia en el eje **y** y **x** es la distancia en el eje **x**. En este caso, **y** es **dist_y** y **x** es **dist_x**.

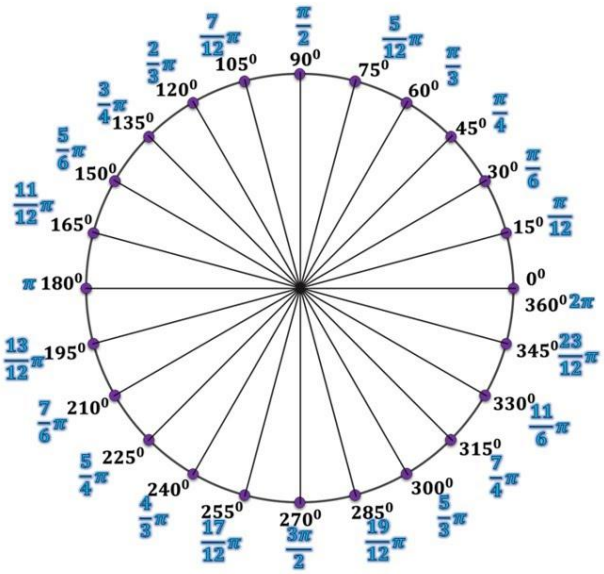
`**angulo_direccion**` es el ángulo desde el eje **x** hasta el punto objetivo (dist_x, dist_y). Es el ángulo al que deberías girar si estuvieras inicialmente alineado con el eje x.

`**angulo_diferencia**` es la cantidad que necesitas girar desde tu orientación actual (`pose_.theta`) para alinearte con `angulo_direccion`. Si `angulo_diferencia` es positivo, necesitas girar a la izquierda. Si es negativo, necesitas girar a la derecha.

Por lo tanto, `angulo_direccion` te dice a dónde quieres ir, y `angulo_diferencia` te dice cómo llegar allí desde donde estás actualmente.

Grados	$\pi\ rad$
15^0	$\pi/12$
30^0	$\pi/6$
45^0	$\pi/4$
60^0	$\pi/3$
75^0	$5\pi/12$
90^0	$\pi/2$
105^0	$7\pi/12$
120^0	$2\pi/3$
135^0	$3\pi/4$
150^0	$5\pi/6$
165^0	$11\pi/12$
180^0	π

Grados	$\pi\ rad$
195^0	$13\pi/12$
210^0	$7\pi/6$
225^0	$5\pi/4$
240^0	$4\pi/3$
255^0	$17\pi/12$
270^0	$3\pi/2$
285^0	$19\pi/12$
300^0	$5\pi/3$
315^0	$7\pi/4$
330^0	$11\pi/6$
345^0	$23\pi/12$
360^0	2π




```
if(angulo_diferencia > M_PI) angulo_diferencia -= 2 * M_PI;
else if(angulo_diferencia  < -M_PI)
                                angulo_diferencia  += 2 * M_PI;
```

Los condicionales `if-else` que siguen se utilizan para **normalizar** `angulo_diferencia` al rango de **-π** a **π**.

La función `atan2` devuelve un valor en el rango de -π a π. Cuando restas `pose_.theta` de `angulo_direccion` para obtener `angulo_diferencia`, el resultado puede salirse de este rango.

Por ejemplo, si `angulo_direccion` es **-π/2** (es decir, -90 grados) y `pose_.theta` es **π** (180 grados), entonces `angulo_diferencia` sería **-π/2 - π = -3π/2**, que está fuera del rango de **-π** a **π**.

Los condicionales `if-else` corrigen esto añadiendo o restando **2π** (que es un giro completo en radianes) a `angulo_diferencia` si es necesario. Esto asegura que `angulo_diferencia` siempre esté en el rango de **-π** a **π**, independientemente de los valores de `angulo_direccion` y `pose_.theta`.

Al normalizar el ángulo a un rango de **-π** a **π**, el robot siempre tomará el camino más corto para girar hacia el objetivo.

Si el 'angulo_diferencia' fuera 320 grados (o aproximadamente 5.6 radianes, que es mayor que **π**), el robot tendría que girar casi una vuelta completa en sentido antihorario para llegar al objetivo si los ángulos no se normalizaran. Pero al normalizar el ángulo, 'angulo_diferencia' se convierte en -40 grados (o aproximadamente -0.7 radianes), por lo que el robot solo necesita girar un poco en sentido horario para llegar al objetivo.

Esto hace que el movimiento del robot sea más eficiente, ya que siempre toma el camino más corto para girar hacia el objetivo.

La última línea es la velocidad de giro de la tortuga, que ya resulta familiar cómo se calcula.

$$u(t) = Kp \cdot e(t)$$

```
msg.angular.z = 6 * angulo_diferencia
```

El factor `6` actúa como una constante proporcional (Kp) en un controlador P. Cuanto mayor sea la diferencia de ángulo, mayor será la velocidad angular, lo que hace que el objeto gire más rápido.

Y ya por aclararlo todo, una vez alcanza el objetivo:

```
// Objetivo alcanzado
msg.linear.x = 0.0;
msg.angular.z = 0.0;
```

Tanto la velocidad lineal como la velocidad angular del robot se establecen en 0. Esto hará que el robot se detenga, ya que no se está moviendo hacia adelante (velocidad lineal = 0) ni girando (velocidad angular = 0). Por lo tanto, si este código se ejecuta cuando la distancia al objetivo es menor que 0.5, significa que el robot se detendrá cuando esté a 0.5 unidades de distancia del objetivo.