

LABORATORIO 1

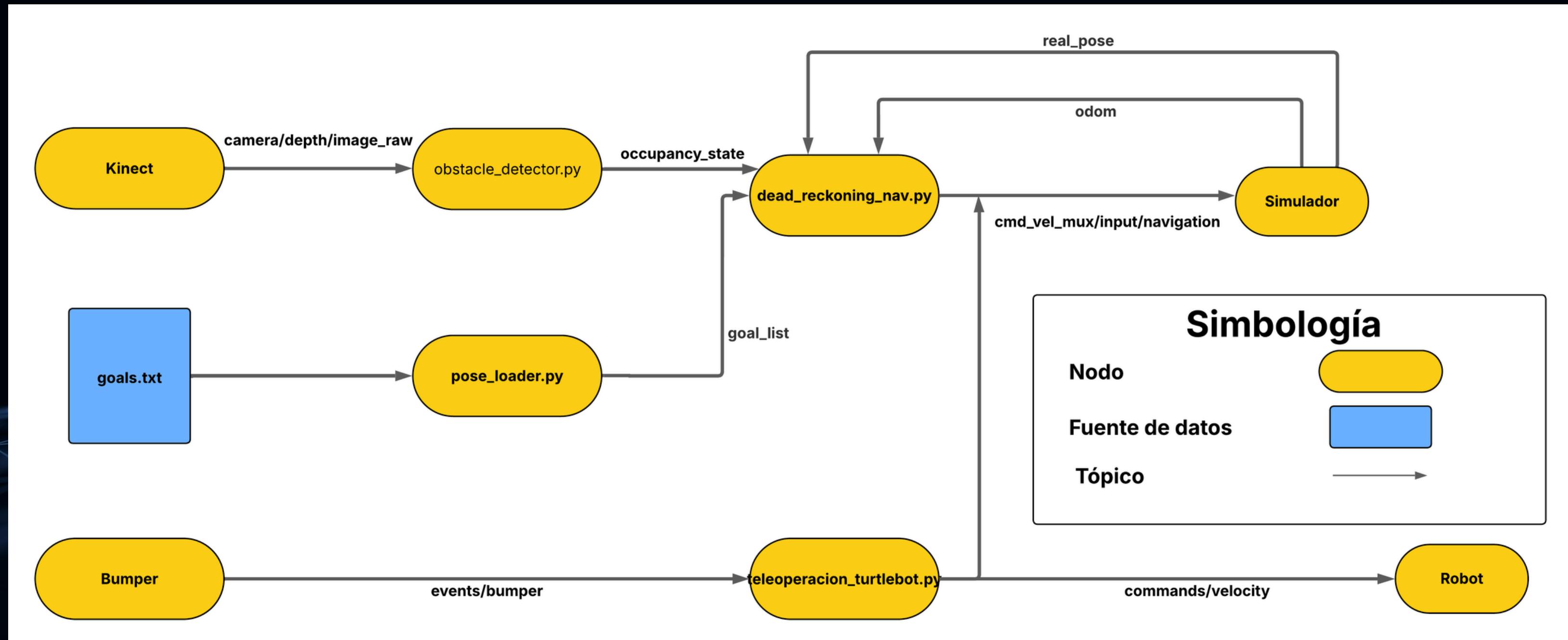
GRUPO 1

JUAN ALARCÓN
BRUNO SCHILLING
DIEGO VILLENA

ÍNDICE

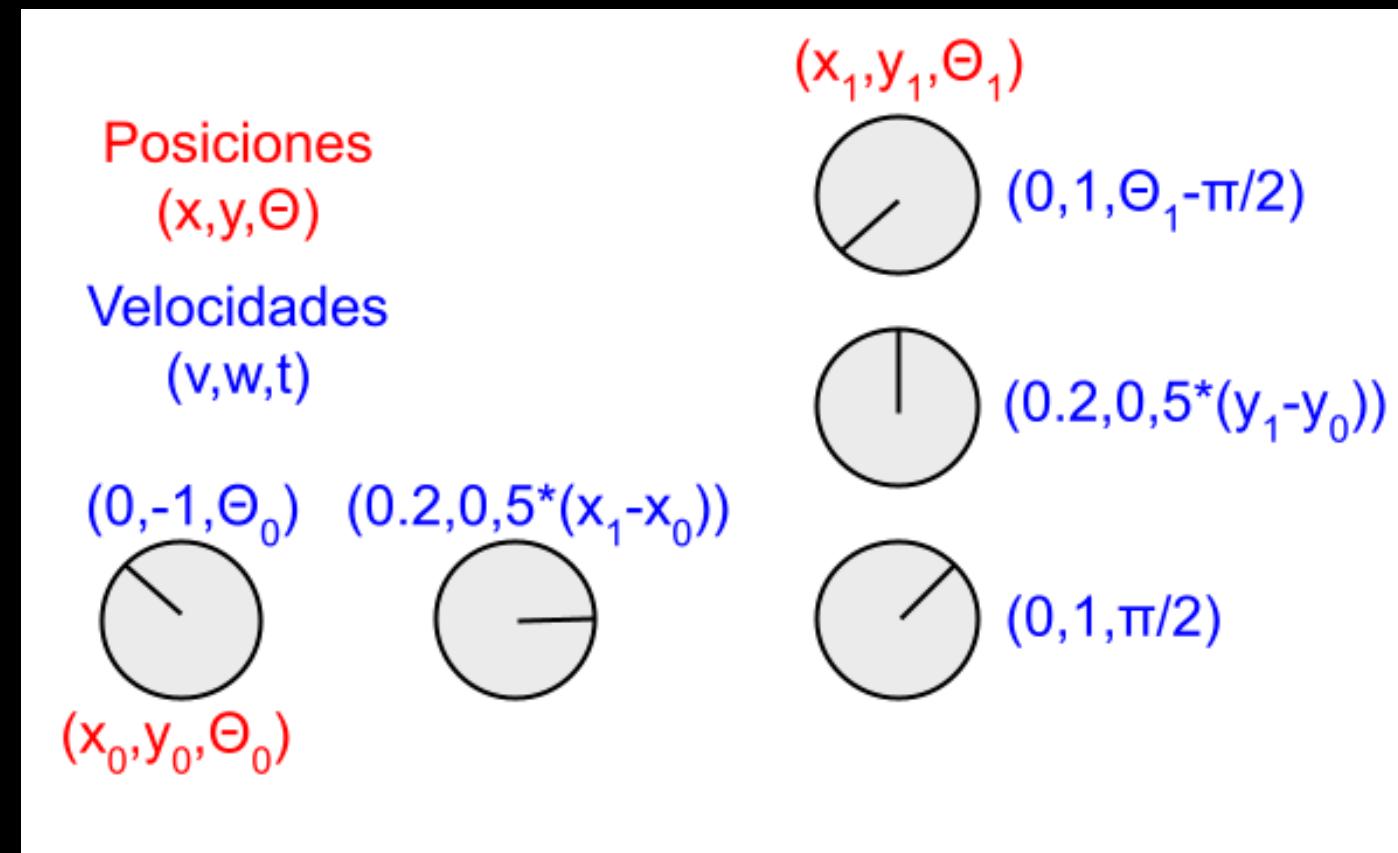
- DIAGRAMA DE NODOS.....3
- ACTIVIDAD 1: AVANZAR Y ROTAR.....4
- ACTIVIDAD 2: KINECT.....10
- ACTIVIDAD 3: TELEOPERACIONES.....16
- CONCLUSIONES.....20

DIAGRAMA DE NODOS



ACTIVIDAD 1

Objetivo: Lograr que el robot alcance puntos objetivos mediante la aplicación de velocidades por un tiempo determinado, sin el uso de odometría durante el trayecto.



Pasos generales de la solución:

- 1) Cargar los puntos objetivos provenientes de un archivo txt.
- 2) Crear una Lista de posiciones con los puntos
- 3) Realizar el movimiento punto a punto.

Pasos para realizar el movimiento:

- 1) Alinear el robot horizontalmente.
- 2) Desplazarse horizontalmente
- 3) Rotar 90 grados
- 4) Desplazarse verticalmente.
- 5) Alinear el robot con el angulo objetivo

ACTIVIDAD 1

Código utilizado en el nodo *dead reckoning_nav.py*:

```
def aplicar_velocidad(self, x, w, t):
    twist = Twist()
    twist.linear.x = float(x)
    twist.angular.z = float(w)
    t_start = time.time()
    while time.time() - t_start < t:
        self.publisher.publish(twist)
    self.publisher.publish(Twist())
```

} Este método aplica una velocidad x y una velocidad angular w por un tiempo t. Finalmente detiene el robot para que no avance por más tiempo

```
def mover_robot_a_destino(self, x, y, o):
    start_x = self.x
    start_y = self.y
    start_o = self.o

    dist_x = x-start_x
    dist_y = y-start_y
    print(f"standing on {self.x}, {self.y}, {self.o}")
    print(f"going to {x}, {y}, {o}")
    correction_factor = 1.14

    self.aplicar_velocidad(0, -sign(start_o), abs(start_o)*correction_factor)
    self.aplicar_velocidad(0.2*sign(dist_x), 0, abs(dist_x)*5)
    self.aplicar_velocidad(0, 1, math.pi/2*correction_factor)
    self.aplicar_velocidad(0.2*sign(dist_y), 0, abs(dist_y)*5)
    self.aplicar_velocidad(0, sign(o-math.pi/2), abs(o-math.pi/2)*correction_factor)
```

} Este método realiza los calculos para lograr que el robot se mueva al punto final considerando la posición inicial

ACTIVIDAD 1

Código utilizado en el nodo *dead reckoning_nav.py*:

```
def odom_callback(self, msg):
    self.x = msg.pose.pose.position.x
    self.y = msg.pose.pose.position.y
    (a,b,self.o) = euler_from_quaternion([
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w,])
```

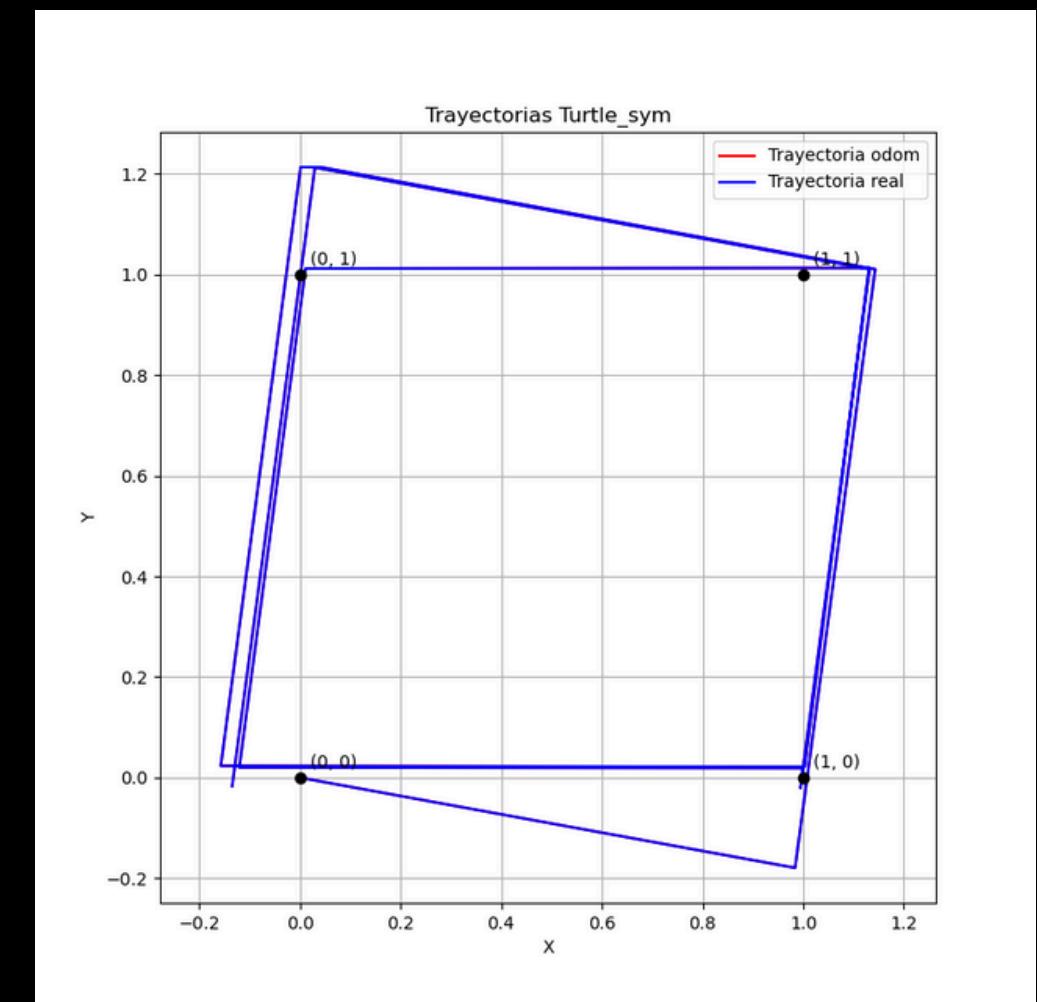
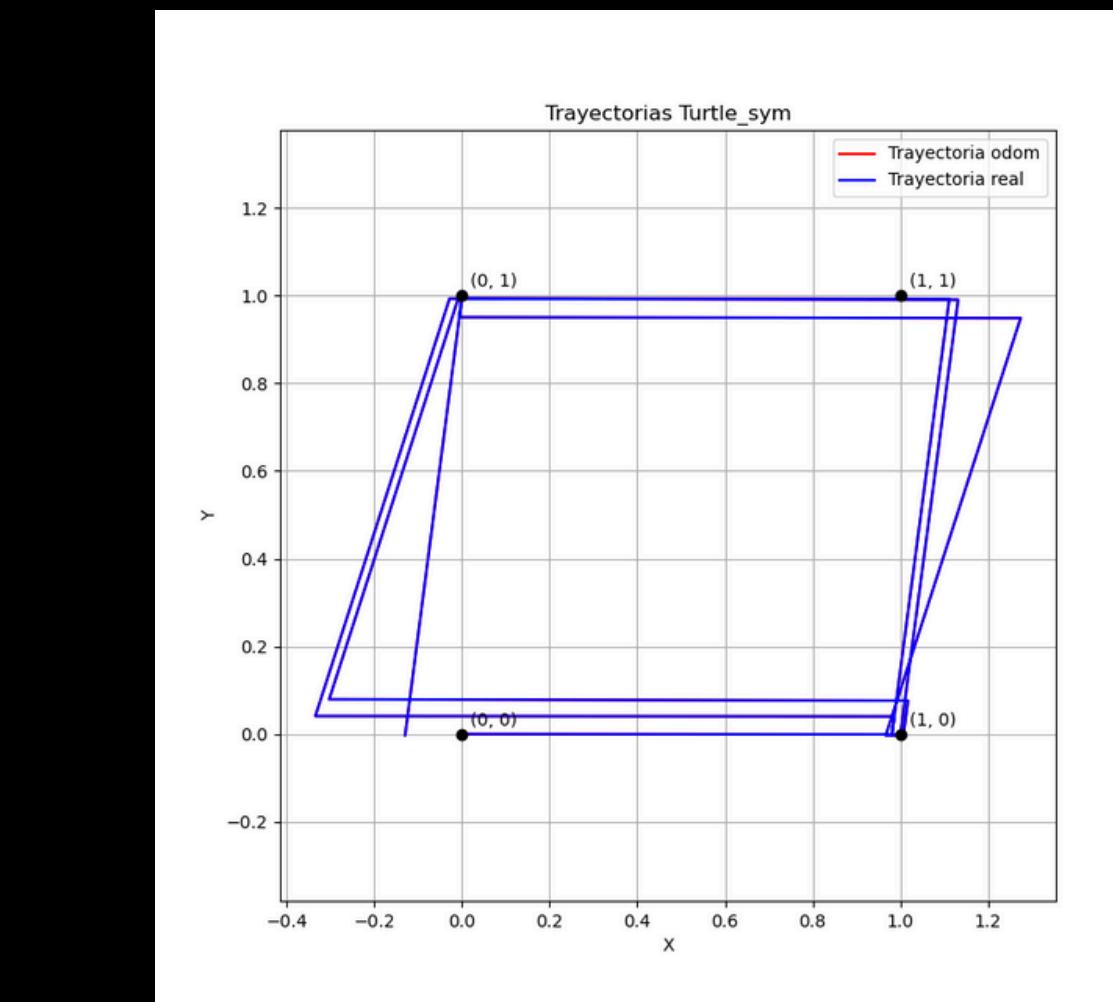
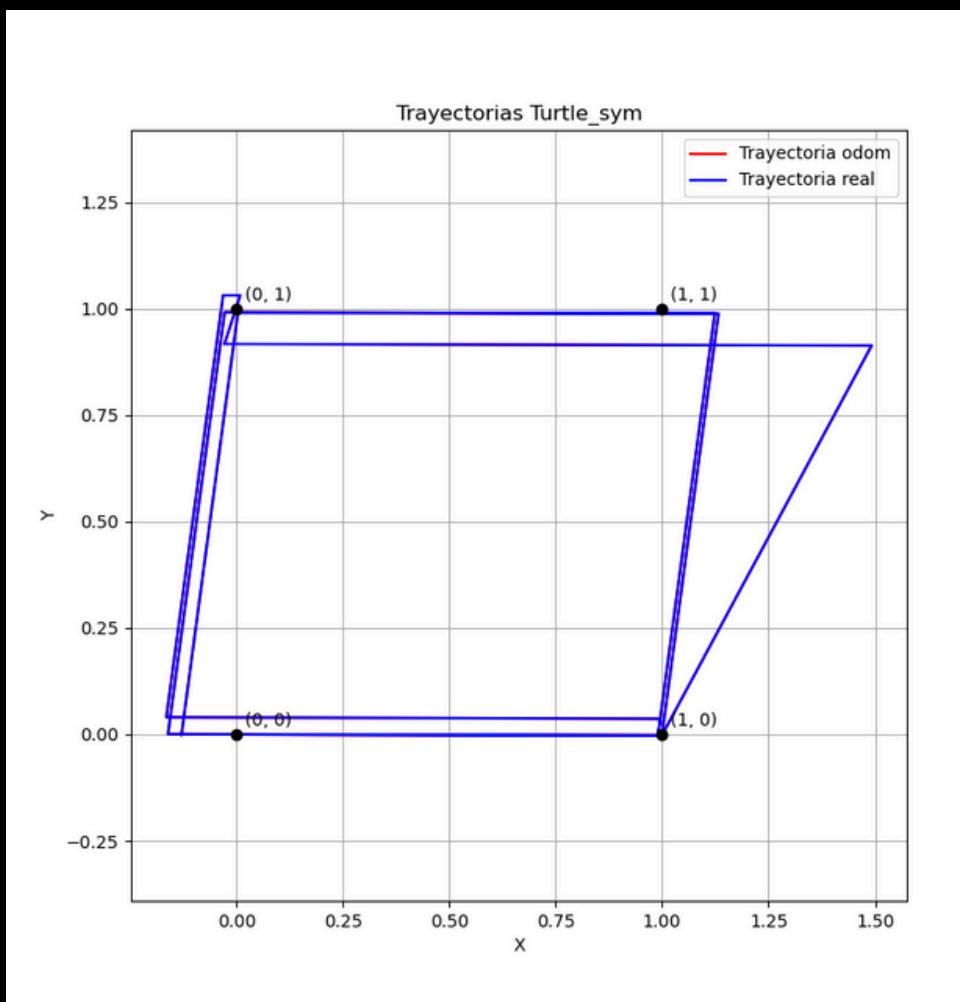
} Este método guarda la pose dada por la odometría y la guarda en variables de la clase

```
def accion_mover_cb(self, msg):
    therad = Thread(target = self.mover_robot_a_lista_de_destinos, args = (msg.data,), daemon = True)
    therad.start()

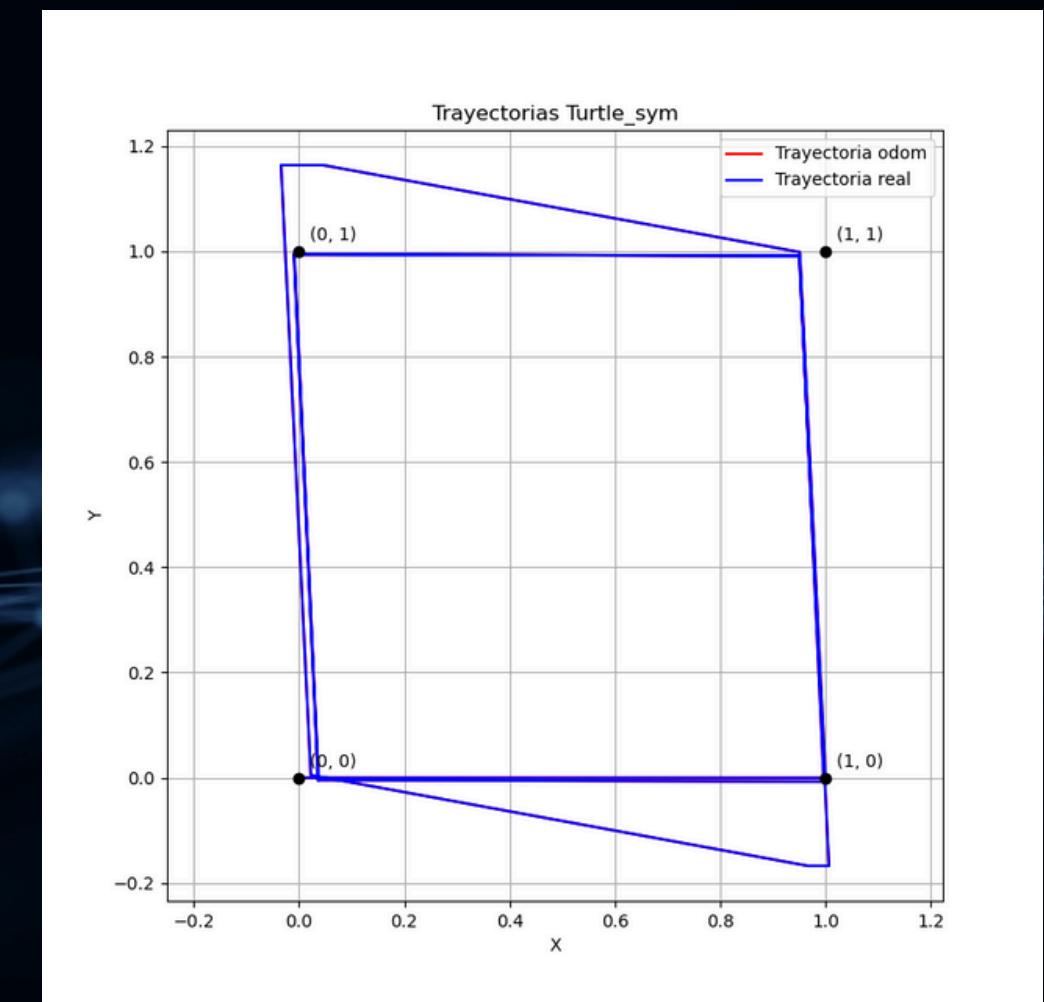
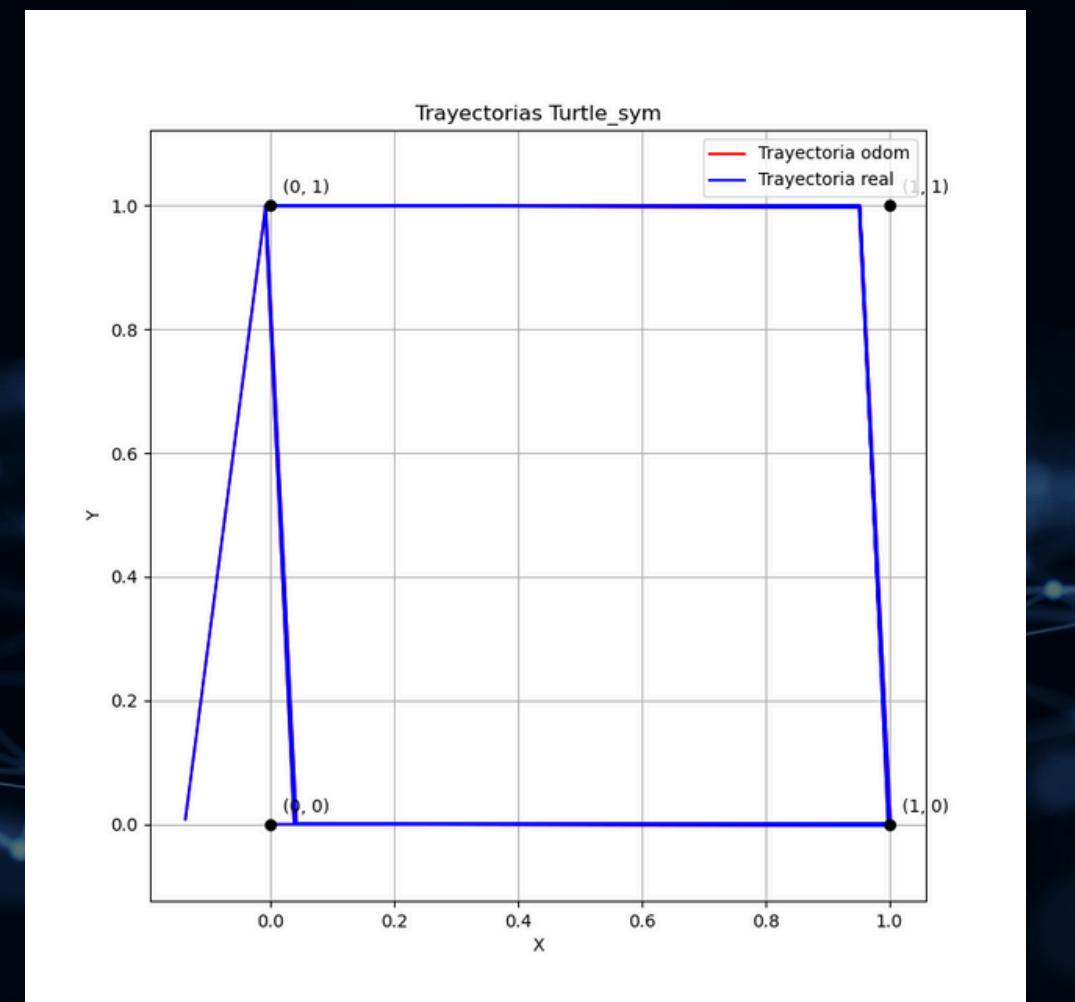
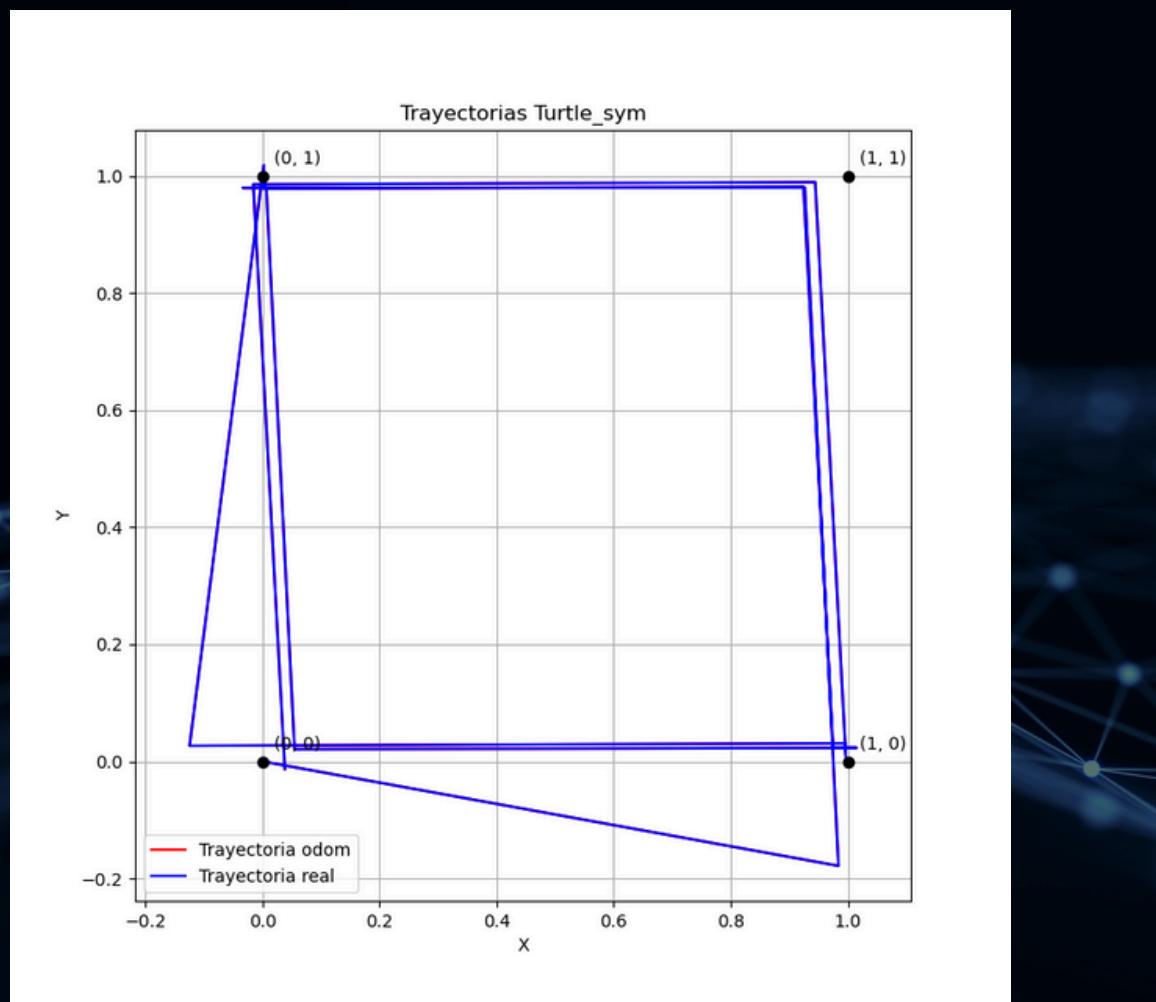
def mover_robot_a_lista_de_destinos(self, data):
    with self.lock:
        datos = data.split(";")
        for line in datos:
            line = line.split(",")
            x = float(line[0])
            y = float(line[1])
            o = float(line[2])
            self.mover_robot_a_destino(x, y, o)
```

} Estos métodos reciben la información de pose_loader y crean un thread que se encargue de mover el robot a cada punto de la lista.

TRAYECTORIAS SIN FACTOR DE CORRECIÓN

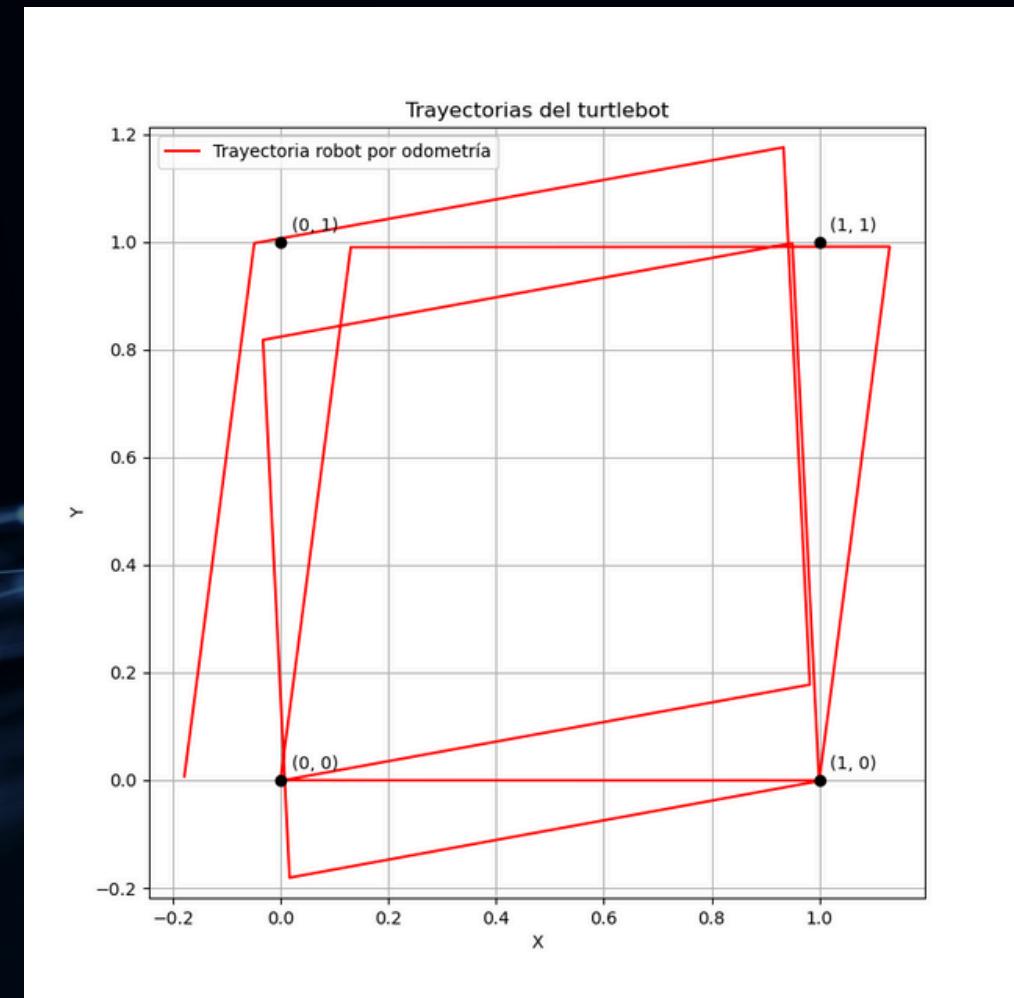


TRAYECTORIAS CON FACTOR DE CORRECIÓN 1.14

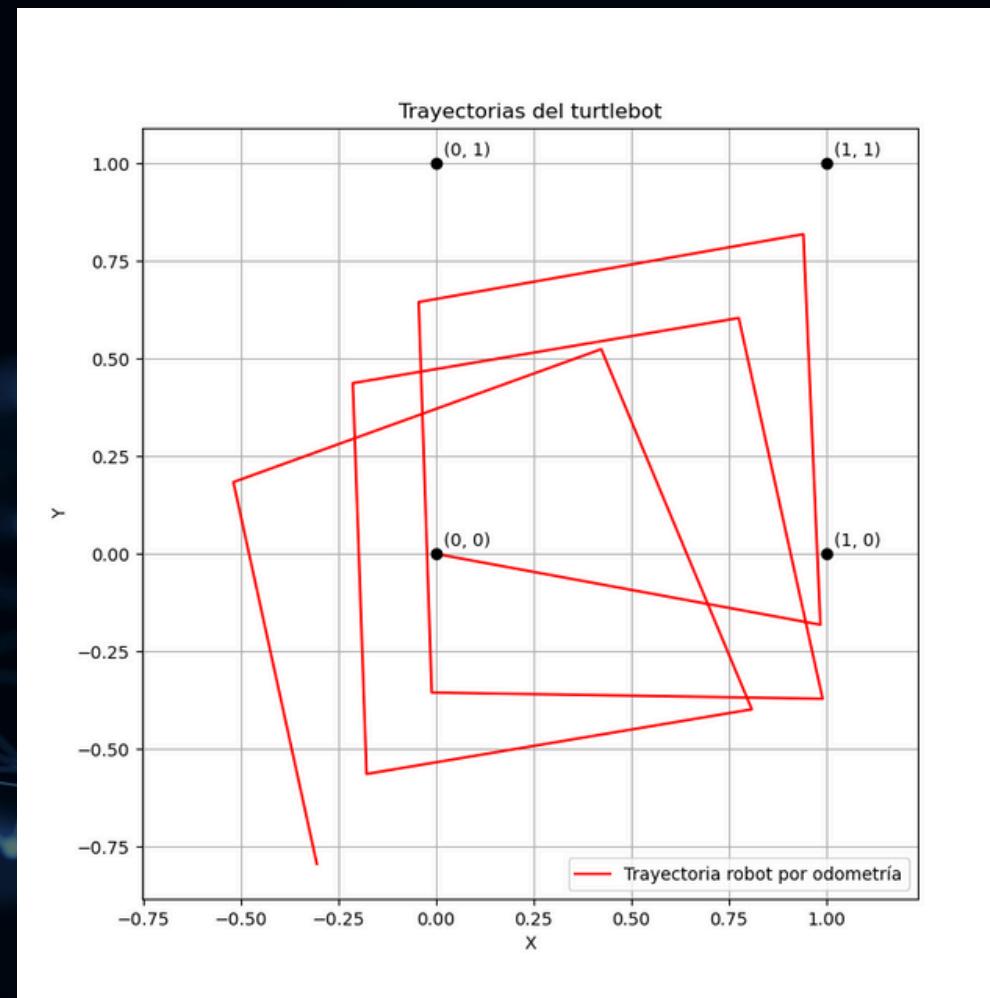


TRAYECTORIAS SIN LECTURA DE ODOMETRÍA

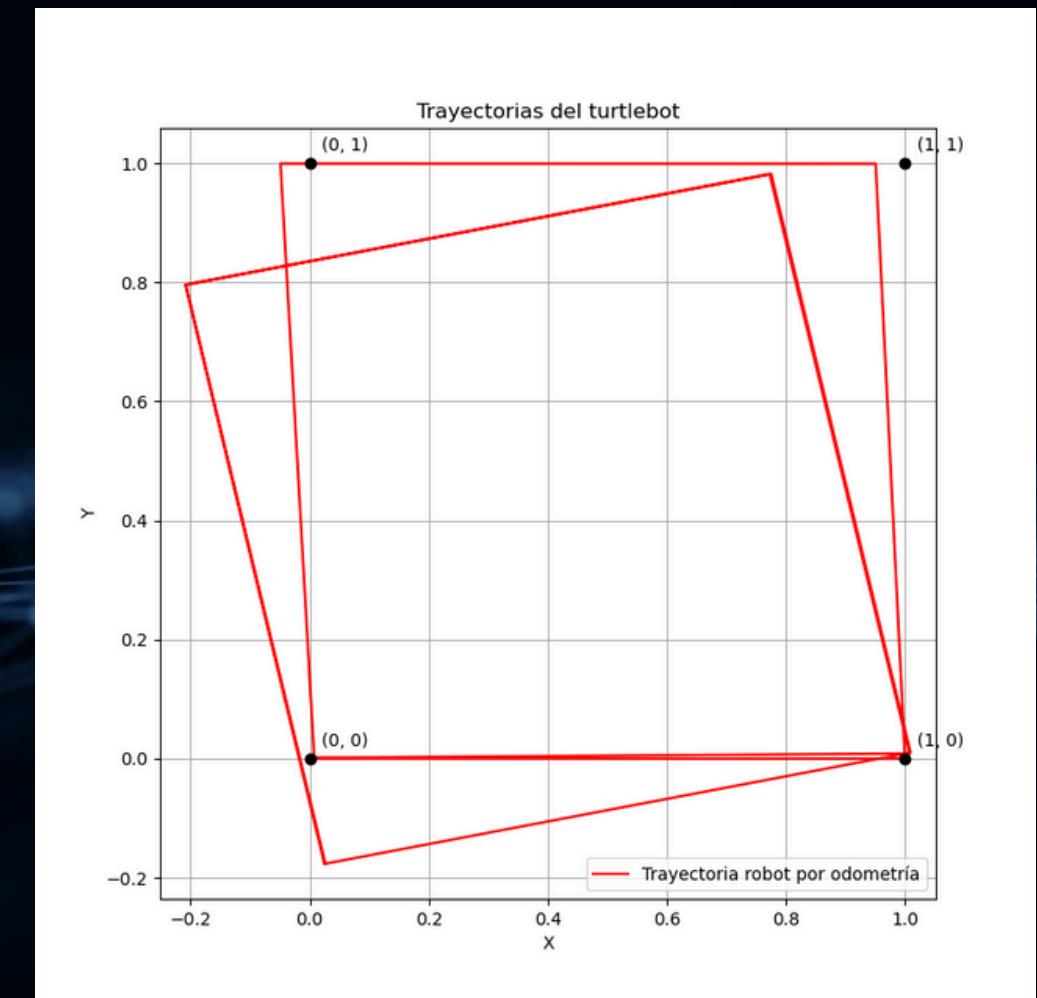
FACTOR 1



FACTOR 1.1



FACTOR 1.14



¿POR QUÉ VEMOS VARIACIONES?

Variaciones por Software:

- `time.time()` genera variaciones, ya que no es continua.
- El simulador, por defecto, ya tiene implementado un cierto grado de incertidumbre.

Variaciones por Hardware:

- El piso irregular puede generar errores imperceptibles de desplazamiento.
- La fricción entre el suelo y las ruedas también pueden afectar esta medición.
- Un error en distancia entre las ruedas puede generar fallas al rotar.

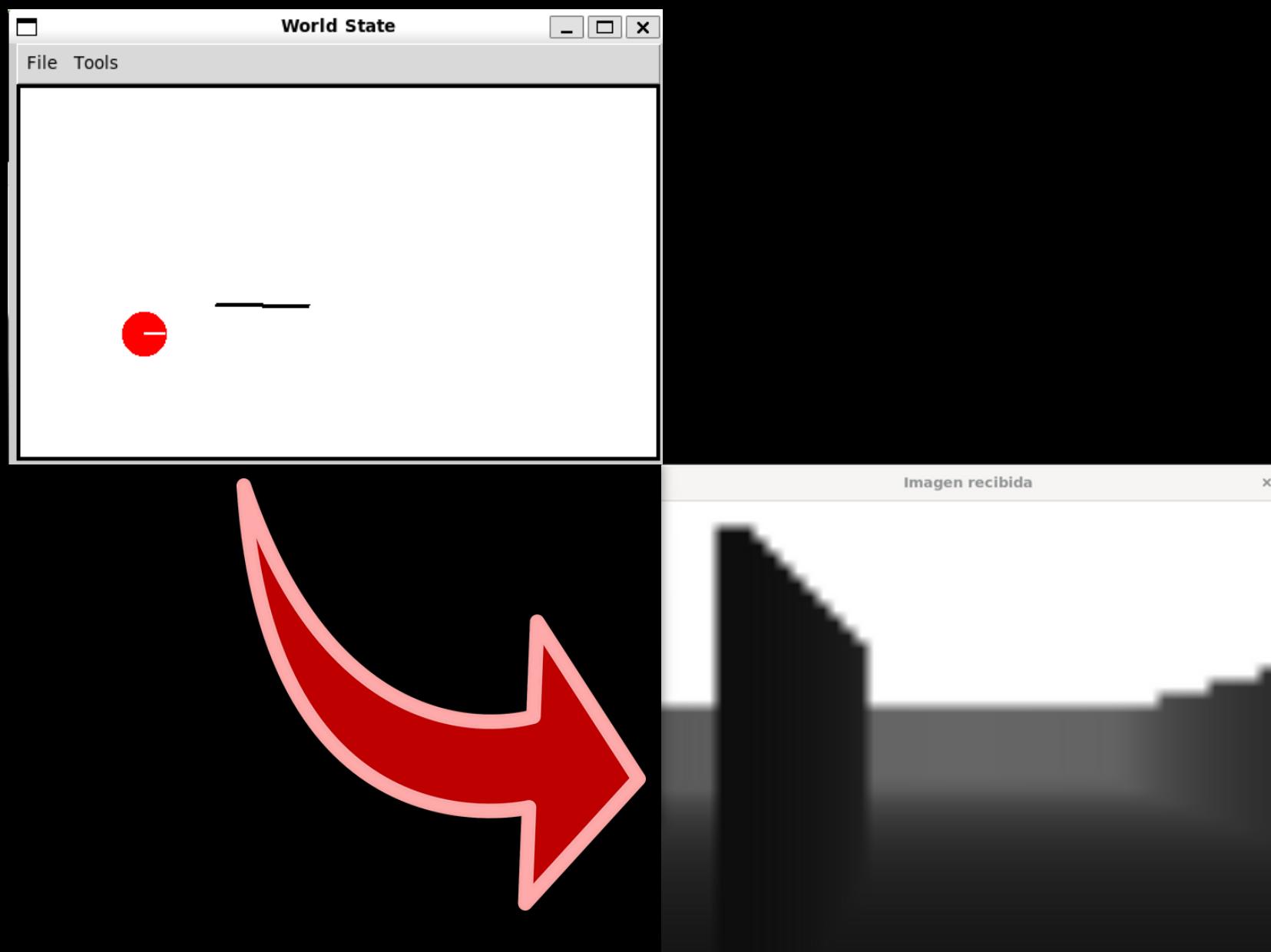
ACTIVIDAD 1

Dificultades de la actividad:

- 1) Complicaciones al ver el sentido de giro necesario para alinearse
- 2) Complicaciones en el uso de threads para asegurarse de que 2 threads no muevan al robot simultáneamente.

ACTIVIDAD 2

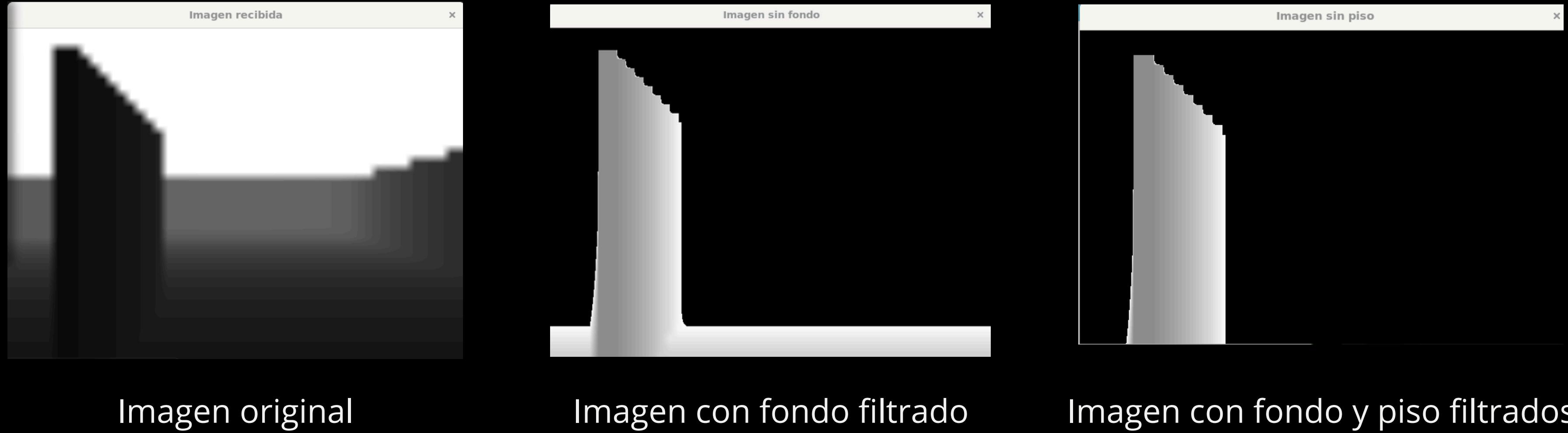
Objetivo: Lograr que el robot sea capaz de detectar obstáculos y detenerse ante la proximidad de estos.



Pasos generales de la solución:

- 1) Procesar imágenes recibidas para eliminar el piso y el fondo.
- 2) Chequear la existencia de obstáculos. En caso de haber, detectar su lugar de procedencia respecto al robot.
- 3) Notificar al robot la presencia del obstáculo, indicando si se encuentra a la izquierda, al centro o a la derecha del robot
- 4) Frenar el robot ante la presencia de este obstáculo mediante un manejo de lógica en el nodo *dead_reckoning_nav.py*

ACTIVIDAD 2



```
root@LAPTOP-5RH7COU6:~# python3 obstacle_detector.py
Oh no, un obstaculo
el obstaculo esta a la izquierda
Publishing: "geometry_msgs.msg.Vector3(x=1.0, y=0.0, z=0.0)"
```

ACTIVIDAD 2

Código utilizado en el nodo *osbtacle_detector.py*:

```
def depth_cb(self, data):
    depth_image = self.bridge.imgmsg_to_cv2(data, desired_encoding='passthrough')
    depth_image = depth_image.astype(np.float32)
    depth_clean = np.nan_to_num(depth_image, nan=0.4)
    depth_clean = np.where(depth_clean<1*self.threshold_value, depth_clean, 0)
    aux_sin_piso = depth_clean[:round(0.85*self.filas), :]
    self.current_cv_depth_image = aux_sin_piso
    self.identificar_zona()
```



Con este método se logra filtrar la imagen capturada del kinect, eliminando los objetos indeseados: el fondo y el piso. Finalmente se procede a aplicar el **método *identificar_zona()***.

ACTIVIDAD 2

Código utilizado en el nodo *osbtacle_detector.py*:

```
def identificar_zona(self):
    if self.current_cv_depth_image is not None:
        alto, ancho = self.current_cv_depth_image.shape

        img_aux1 = self.current_cv_depth_image[:, :round(ancho/3)]
        img_aux2 = self.current_cv_depth_image[:, (round(ancho/3)+1):2*round(ancho/3)]
        img_aux3 = self.current_cv_depth_image[:, (2*round(ancho/3)+1):]

        area_tot = alto*ancho
        n1 = cv2.countNonZero(img_aux1)
        n2 = cv2.countNonZero(img_aux2)
        n3 = cv2.countNonZero(img_aux3)

        if (n1 + n2 + n3) > 0.1*area_tot:
            print("Oh no, un obstáculo")
            if n1>n2 and n1>n3:
                print("el obstáculo esta a la izquierda")
                msg = Vector3()
                msg.x = float(1)
                msg.y = float(0)
                msg.z = float(0)
                self.publisher_.publish(msg)
                print('Publishing: "%s"' % msg)

            elif n2>n1 and n2>n3:
                print("el obstáculo esta al centro")
                msg = Vector3()
                msg.x = float(0)
                msg.y = float(1)
                msg.z = float(0)
                self.publisher_.publish(msg)
                print('Publishing: "%s"' % msg)

            elif n3>n1 and n3>n2:
                print("el obstáculo a la derecha")
                msg = Vector3()
                msg.x = float(0)
                msg.y = float(0)
                msg.z = float(1)
                self.publisher_.publish(msg)
                print('Publishing: "%s"' % msg)

        else:
            print("Camino limpio")
            msg = Vector3()
            self.publisher_.publish(msg)
```



Se cuenta la cantidad de elementos no nulos de la imagen filtrada. Si estos superan cierto umbral, significa que hay un obstáculo en el camino.

Si la presencia de un obstáculo es confirmada, la imagen es dividida en 3 zonas (izquierda, derecha y centro) para luego comparar cuál de estas tiene la mayor cantidad de elementos no nulos. Con esto, se puede determinar en dónde se encuentra el obstáculo.

ACTIVIDAD 2

Cambios en el nodo *dead reckoning nav.py*:

```
def aplicar_velocidad(self, x, w, t):
    twist = Twist()
    twist.linear.x = float(x)
    twist.angular.z = float(w)
    t_passed = 0
    t_start = time.time()
    while time.time() - t_start + t_passed < t:
        if self.obstacle_detected:
            t_passed += time.time() - t_start
            self.publisher.publish(Twist())
            while self.obstacle_detected:
                time.sleep(0.1)
                t_start = time.time()
            self.publisher.publish(twist)
    self.publisher.publish(Twist())
```

```
def obstacle_callback(self, msg):
    self.left_side = msg.x
    self.middle_side = msg.y
    self.right_side = msg.z
    if (self.left_side + self.middle_side + self.right_side) > 0:
        self.obstacle_detected = True
    else:
        self.obstacle_detected = False
    if self.obstacle_detected:
        if self.left_side > 0:
            self.get_logger().info("Obstáculo a la izquierda")
        elif self.middle_side > 0:
            self.get_logger().info("Obstáculo al centro")
        else:
            self.get_logger().info("Obstáculo a la derecha")
```



Cambios en método *aplicar_velocidad()*.

Creación del método *osbtacle_callback()*.

ACTIVIDAD 2

Dificultades de la actividad:

- 1) Falta de información de la kinect proporcionada por el simulador (corregido posteriormente con una pequeña modificación en el simulador).
- 2) Complicaciones con la decodificación de las imágenes recibidas producto de sus formatos.
- 3) Encontrar valores de thresholds adecuados para el filtrado de las imágenes y reemplazo de los valores nan.

ACTIVIDAD 3

Objetivo: Controlar al robot mediante el envío de velocidades lineales y angulares en tiempo real, y de forma remota.



Pasos generales de la solución:

- 1) Establecer comunicación entre laptops mediante la conexión a la misma red para acceder a los nodos.
- 2) A nivel de código, detectar las teclas presionadas para determinar las velocidades asociadas y aplicarlas.

ACTIVIDAD 3

Código utilizado en el nodo teleoperacion turtlebot.py:

```
pressed_keys = set() #Se almacenan las teclas apretadas actualmente

def on_press(key):
    pressed_keys.add(key.char) #Se añade tecla

def on_release(key):
    pressed_keys.discard(key.char) #Se libera tecla
```



Se verifica el pulso de las teclas mediante el uso de un *set()*. De esta manera, mediante los métodos de *add* y *discard* es fácil verificar si una tecla está siendo pulsada en el momento simplemente revisando ese *set*.

```
def bumper_callback(self, msg):
    if msg.state == BumperEvent.PRESSED:
        self.collision_detected = True
        self.vel_publisher.publish(Twist())
    else:
        self.collision_detected = False

def check_keyboard(self):
    if self.collision_detected:
        return

    cmd = Twist()
    if 'i' in pressed_keys: cmd.linear.x = 0.2
    if 'j' in pressed_keys: cmd.linear.x = -0.2
    if 'a' in pressed_keys: cmd.angular.z = 1.0
    if 's' in pressed_keys: cmd.angular.z = -1.0
    if 'q' in pressed_keys:
        cmd.linear.x = 0.2
        cmd.angular.z = 1.0
    if 'w' in pressed_keys:
        cmd.linear.x = 0.2
        cmd.angular.z = -1.0
    self.vel_publisher.publish(cmd)
```



Por otra parte, se cuenta con 2 métodos propios del nodo. El método *bumper_callback()* se encarga de verificar la colisión mediante el bumper físico del robot, enviándose un *Twist* vacío para impedir que pueda moverse. El método *check_keyboard()* se encarga de codificar las teclas pulsadas y asociarlas a sus respectivas velocidades lineales o angulares, enviándoselas al robot.

ACTIVIDAD 3



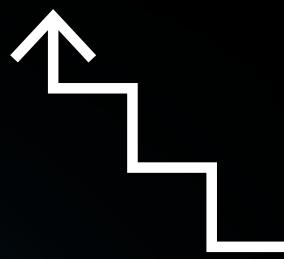
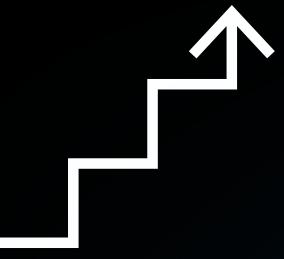
ACTIVIDAD 3

Dificultades de la actividad:

- 1) Complicaciones para lograr vincular los computadores a través de la red (en primera instancia, solucionado a través de ssh).
- 2) Dificultades para implementar el keyboard, que finalmente tuvo que ser importado desde la librería pynput (Esto a su vez también generó dificultades para importar desde los computadores root desde donde se controla el robot).
- 3) En ocasiones, dificultades para activar el bumper dependiendo de la orientación del choque.

CONCLUSIONES

- Trabajar con nodos agiliza bastante los procesos de programación y depuración, permitiendo tener una estructura más ordenada.
- El sensor kinect resulta bastante útil para dotar de percepción a un robot que se mueve por interiores, ya que existe una variedad de herramientas y técnicas para trabajar con imágenes que son relativamente fáciles de usar.
- Las no idealidades de nuestros dispositivos realmente pueden provocar errores significativos como se vio en la actividad 1, y eso considerando que eran instrucciones de movimiento pequeñas.
- El método de navegación de dead_reckoning, si bien es cierto otorga una aproximación aceptable para efectos de proyectos de desplazamientos en escalas más pequeñas, es claro que no sería conveniente utilizarlo en navegaciones en terrenos más grandes, o en donde se requiera de una gran precisión. Para este último caso, sería más conveniente utilizar métodos que posean alguna retroalimentación que minimice los errores mediante el uso de sensores, como lo es la técnica de la odometría.



LABORATORIO 1



GRUPO 1

JUAN ALARCÓN
BRUNO SCHILLING
DIEGO VILLENA

