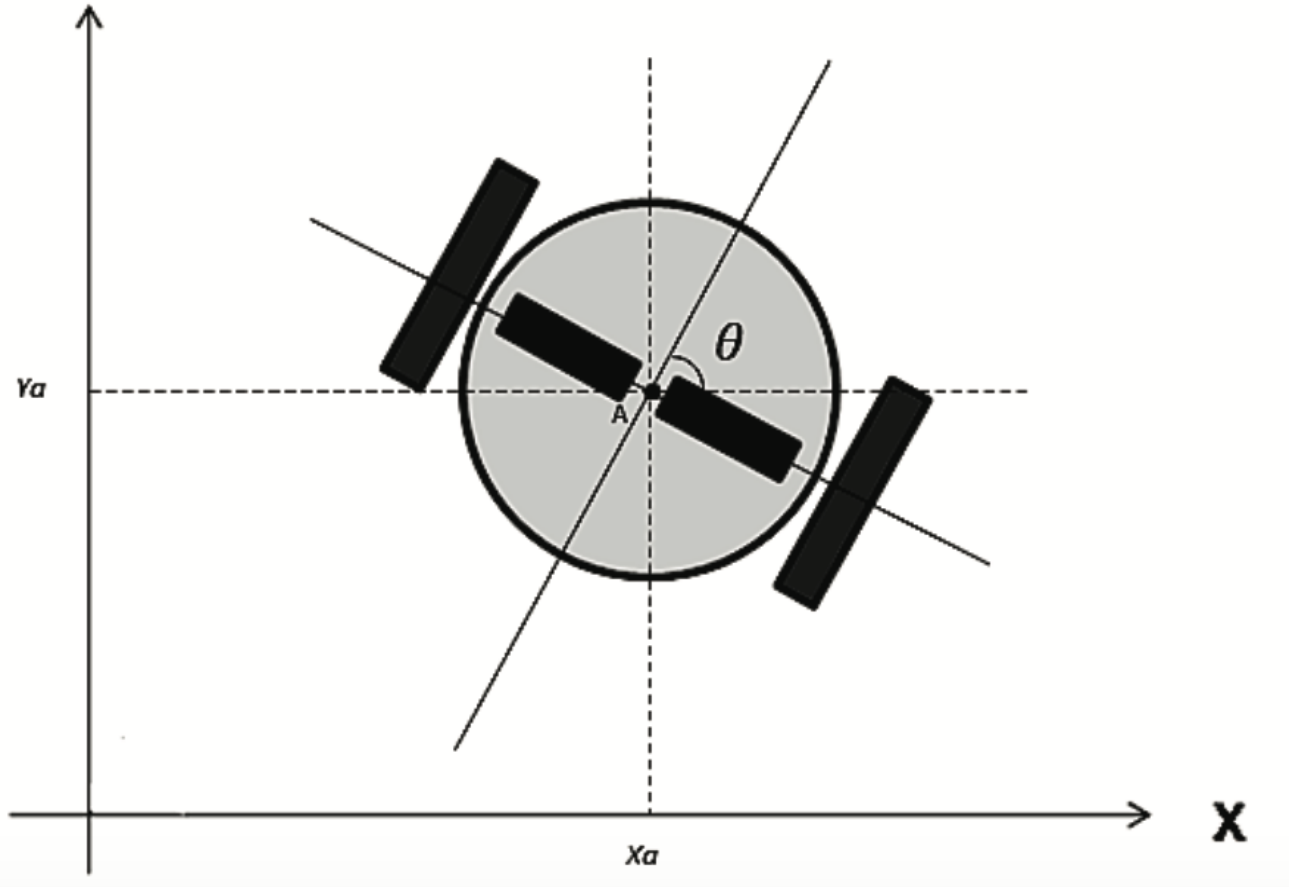
8 Проверочная работа

Пакет симуляции Turtlesim

Turtlesim - это очень простой симулятор робота, который может перемещаться в двухмерном пространстве. Он удобен для того, чтобы освоить базовые принципы управления такими роботами. Ранее мы уже сталкивались с ним, но сейчас рассмотрим его более подробно.

Роботом с дифференциальным приводом

В программе `turtlesim` мы можем эмулировать робота с так называем дифференциальным приводом. На таком роботе на одной оси установленные два мотора, которые могут вращаться в любом направлении.



Если у робот оба мотора вращаются в одном направлении, то такой робот едет прямо вперед. Если моторы вращаются на разные стороны, робот вращается на

месте.

Если скорости вращения ведущих колес разные, то робот будет выполнять поворот в ту сторону, в которой скорость вращения ведущего колеса меньше по отношению ко второму ведущему колесу.

Любой робот или другой объект в пространстве имеет шесть степеней свободы (DOF). Первые три степени свободы - это координаты робота в пространстве (x,y,z). Остальные три степени свободы относятся к ориентации робота в пространстве. Это такие значения, как крен, тангаж и рыскание.

Робот с дифференциальным приводом перемещается в двухмерной плоскости (2D) (робот не может летать), и его положение в любой момент можно описать двумя координатами X и Y, лежащими в горизонтальной плоскости. При этом курс робота обозначается как θ (theta). Этих данных вполне достаточно, чтобы описать положение робота с дифференциальным приводом.

Управление роботом

Управление перемещением роботом в пространстве в большинстве случаев используют одинаковую схему – это схема управления скоростями по степеням свободы. Мы передаем роботу скорости, которые мы хотим, чтобы он "выполнил", и робот начинает движение с заданной скоростью. При этом скорости могут как быть "линейные", то есть скорости по осям X Y Z, так и угловые для задания вращения.

Если мы передадим нулевые скорости, то робот должен остановиться.

Для такой системы управления в ROS уже создано стандартное сообщение `` `geometry_msgs/Twist ``

```
rosmsg info geometry_msgs/Twist
===
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
```

```
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Для нашего дифференциального робота актуальными являются всего два значения - это линейная скорость по оси робота X `linear.x` (скорость движения робота вперед) и угловая скорость вращения вдоль по оси Z `angular.z`. Других степеней свободы дифференциальный робота не имеет. Но, например, коптер имеет все доступные степени свободы. Попробуем управление робота.

Запусти симулятор `turtlesim`

```
rosrun turtlesim turtlesim_node
```

Найдем топик управления

```
rostopic list
===
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Это топик `/turtle1/cmd_vel`. Опубликуем в него данные со скоростями и посмотрим, что происходит.

Установим линейную скорость на 0.5 м/с.

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x: 0.5
  y: 0.0
  z: 0.0
```

```
angular:
  x: 0.0
  y: 0.0
  z: 0.0" -r 10
```

Черепашка начнет движение прямолинейно по направлению оси X робота.

Усложним пример, добавим угловую скорость.

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0" -r 10
```

Черепашка начнет движение по окружности.

Получение данных о положение

Мы смогли управлять перемещением роботом, следующий важный вопрос - это получение данных о его положении. Тут нам необходимо ввести понятие Одометрии.

Одометрия (Odometry — от греческих слов hodos («перемещение», «путешествие») и metron («мера», «измерять»)) — использование данных о движении приводов (колес) для оценки перемещения. Стандартной схемой определения одометрии робота является использование энкодеров, считывающих угол поворота колёс.

Зная радиус колес, скорость колеса и геометрию рамы робота, мы можем оценить путь пройденный роботом. Симулятор turtlesim "рассчитывает" такие данные, и мы можем их получить.

Данные о положении черепахи в симуляторе доступны в топике `/turtle1/pose` тип сообщения `turtlesim/Pose`. Выведем эти данные.

```
rostopic echo /turtle1/pose
===
x: 9.80225086212
y: 6.08695411682
theta: -1.66393887997
linear_velocity: 0.0
angular_velocity: 0.0
```

Если робот будет двигаться, то мы увидим, что данные меняются.

Программа движения робота

Мы разобрались, как управлять роботом и получать данные о его положении в симуляторе `turtlesim`. Давайте рассмотрим программу, которая заставит робота проехать 3 метра и остановиться. На базе этой программы вам будет необходимо выполнить зачетное домашнее задание.

```
import rospy, math

from turtlesim.msg import Pose
from geometry_msgs.msg import Twist

class Turtle():

    def __init__(self):

        rospy.on_shutdown(self.shutdown)
        self.pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=1
        rospy.Subscriber("turtle1/pose", Pose, self.pose_callback)
        self.pose = None

    def shutdown(self):
        self.pub.publish(Twist())

    def pose_callback(self, pose):
        self.pose = pose

    def wait_turtle(self):
        while self.pose is None:
            rospy.loginfo('Wait turtle')
            rospy.sleep(0.1)



    def move_forward(self, dist):

        start_pose = self.pose
        move_dist = 0
        cmd_vel = Twist()

        while not rospy.is_shutdown():

            move_dist = self.getMoveDist(start_pose, self.pose)

            if (move_dist < dist):
```

```
                cmd_vel.linear.x = 1
                self.pub.publish(cmd_vel)
            else:
                cmd_vel.linear.x = 0
                self.pub.publish(cmd_vel)
                return

            rospy.sleep(0.2)


    def getMoveDist(self, start_pose, current_pose):

        move_dist = math.sqrt(math.pow(start_pose.x - current_pose.x,2) +
        rospy.loginfo("Dist :%s", move_dist)
        return move_dist


rospy.init_node('move_forward')
rospy.loginfo("Start Node")

turtle = Turtle()
turtle.wait_turtle()
turtle.move_forward(3)
```

Программа выглядит уже сложнее и больше, чем мы видели раньше. Но такая программа уже больше похоже на те, какие реально работают на роботах.

Алгоритм программы по сути довольно простой. Если у нас есть координаты, то зная текущую и стартовую координаты, мы можем вычислить расстояние, которое проехала черепаха. Если расстояние меньше требуемого, то продолжать ехать. Если меньше, то остановиться и выйти. Но есть нюансы реализации.

Конструктор класса - создание Издателя для управления перемещением робота и подписка на данные Одометрии.

```
rospy.on_shutdown(self.shutdown)
self.pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
rospy.Subscriber("turtle1/pose", Pose, self.pose_callback)
self.pose = None
```

Новый для нас метод. Он нам нужен для правильного завершения программы. Мы говорим, что если мы решили отключиться от ROS, то нужно вызвать метод класса `shutdown`

```
rospy.on_shutdown(self.shutdown)
```

Вызвать такой метод необходимо, чтобы остановить робота. Иначе программа завершится, а робот может продолжить свое движение, это будет неожидаемым поведением.

Также мы инициализировали переменную, в которой собираемся хранить текущее положение робота.

```
self.pose = None
```

Функция "отключения" робота. Мы создаем пустой объект `Twist`. И отправляем его через Издателя. Пустой `Twist` содержит все нулевые скорости, поэтому робот остановится.

```
1  def shutdown(self):
2      self.pub.publish(Twist())
```

Каллбек функуия, мы просто сохраним положение в атрибуте класса `pose`, потом мы будем использовать эту переменную для расчета пройденного расстояния.

```
1  def pose_callback(self, pose):
2      self.pose = pose
```

Метод, который позволит нашей программе работать правильно.

```python
def wait_turtle(self):
    while self.pose is None:
        rospy.loginfo('Wait turtle')
        rospy.sleep(0.1)
```

По ходу выполнения нашей программы нам важно, чтобы в самом начале работы мы знали положение робота. Но у нас может быть ситуация, что программа уже запустилась и готова рассчитать расстояние, а мы еще не получили "первый" callback с данными положения робота.

Поэтому нам необходима функция, которая позволит работать программе только после того, как мы получили первые данные о положении робота.

Сама главная функция "ехать вперед на расстояние `dist` "

```python
def move_forward(self, dist):

    start_pose = self.pose
    move_dist = 0
    cmd_vel = Twist()

    while not rospy.is_shutdown():

        move_dist = self.getMoveDist(start_pose, self.pose)

        if (move_dist < dist):
            cmd_vel.linear.x = 1
            self.pub.publish(cmd_vel)
        else:
            cmd_vel.linear.x = 0
            self.pub.publish(cmd_vel)
            return

        rospy.sleep(0.2)
```

Логика работы этой функции описана в самом алгоритме программы. Мы сохраняем точку старту, и раз в `rospy.sleep(0.2)` в цикле рассчитываем расстояние `self.getMoveDist` и принимаем решение, что делать: ехать дальше или выходить из метода.

```
# Установить линейную скорость по оси робота X 1 м/с
cmd_vel.linear.x = 1
# Обнулит скорость
cmd_vel.linear.x = 0
```

Функция `getMoveDist` производит расчет пройденного расстояния робота по теореме пифагора. На вход мы передаем стартовую и текущую координату, а функция выдает расстояние между двумя этими точками.

```
    def getMoveDist(self, start_pose, current_pose):

        move_dist = math.sqrt(math.pow(start_pose.x - current_pose.x,2) +
        rospy.loginfo("Dist :%s", move_dist)
        return move_dist
```

Ну и запуск нашей программы

```
turtle = Turtle()
turtle.wait_turtle()
turtle.move_forward(3)
```

Мы создаем обьект turtle. Ожидаем получения первых данных о положении робота, едем на 3 метра.

Курсовая работа по заочному курсу

На основе примера.

Движение по квадрату

Доработать программу, чтобы при запуске программы робот черепаха двигалась по квадрату со сторонами равными 3 метрам.

В результате работы программы черепаха должны проехать все вершины квадрата и оказаться в точке старта. Ориентация черепахи в точке финиша должна совпадать с ориентацией в момент старта.

Добавить сервис

Добавить в программу управления черепахой сервис.

После запуска программы черепаха ожидает получения данных через сервис с информацией о длине стороны квадрата.

При получении данных черепаха начинает движение по заданному квадрату.

При получении новых данных о размере стороны квадрата программы должны изменить заданный размер стороны квадрата на новое и продолжить выполнение.

При получении нулевой длинны стороны квадрата черепаха должны остановиться и продолжить движение только после получения корректной длинны.

Черепаха должна двигаться по квадрату в бесконечном цикле, до остановки программы.