ESCOLA POLITÉCNICA DA USP

EQUIPE SKYRATS DE DRONES INTELIGENTES DA ESCOLA POLITÉCNICA DA USP

por Fernando Zolubas Preto,
Integrante da equipe Skyrats a 4 anos, ex Lider do time de Hardware
e atual Engenheiro de Controle no time de Software da equipe.

APOSTILA DE IMPLEMENTAÇÃO BÁSICA DE SISTEMA DE CONTROLE APLICADO A ROBÓTICA COM ROS (ROBOTIC OPERATIING SYSTEM)

Prof. Dr Marcelo Zuffo

SÃO PAULO - SP

2022

por Fernando Zolubas Preto,
Integrante da equipe Skyrats a 4 anos, ex Lider do time de Hardware
e atual Engenheiro de Controle no time de Software da equipe.

APOSTILA DE IMPLEMENTAÇÃO BÁSICA DE SISTEMA DE CONTROLE APLICADO A ROBÓTICA COM ROS (ROBOTIC OPERATIING SYSTEM)

PSI3442

Projeto de Sistemas Embarcados

SÃO PAULO - SP

2022

SUMÁRIO

1 Demo Package: Controlando Turtlesim com ROS e Python intuitivamente

Nesse capitulo será apresentado o *step by step* de como implementar um algorítimo em python intuitivo, mas não otimizado, de controle da tartaruga do node *turtlesim* que é o *hello world* do ROS *Robotic Operational System*

1.1 Criando um novo workspace e um novo packge

1.1.1 Criando um workspace

```
mkdir -p  workshop21_workspace/src
cd workshop21_workspace/
catkin build
nano ~/.bashrc
```

No nano (editor de texto sem mouse), utililze a seta para baixo e desça até a ultima linha e acrescente:

```
source ~/workshop21_workspace/devel/setup.bash
```

E caso não tenha acrescente também:

```
source /opt/ros/melodic/setup.bash
```

Saia do nano utilizando: ctrl + x

1.1.2 Criando um package

O comando a seguir cria um packge chamado *turtle_control_demo*1 habilitado para trabalhar com C++ (**roscpp**) e com python (**rospy**).

```
cd ~/workshop21_workspace/src
catkin_create_pkg turtle_control_demo1  rospy roscpp
```

Depois disso é preciso criar uma pasta chamada **scripts** caso ela não exista dentro do pakge criado. A sequência de comandos a seguir realiza essa operação:

Estando na home utilize:

```
cd ~/workshop21_workspace/src/turtle_control_demo1
mkdir scripts
```

Criar um arquivo **.py** chamado *turtle_control_demo*1*.py* da maneira que preferir mas dentro da pasta **scripts**.

E para torná-lo exercutável faça, estando na home:

```
cd ~/workshop21_workspace/src/turtle_control_demo1/scripts
chmod +x turtle\_control\_demo1.py
```

1.2 Node de controle: turtle_control_demo1.py

1.2.1 Estratégia

Para sair de uma posição (x_0, y_0) e chegar numa posição (x, y) qualquer do plano 2D XY será utilizada a intuição de coordenadas polares de tal modo a dividir o problema multivariável em 2 problemas univariáveis.

Esses problemas são:

- Problema do angulo

  A ideia básica é posicionar a cabeça da tartaruga na direção do ponto alvo (x, y) utilizando-se o angulo formado pelo segmento $(x - x_0, y - y_0)$ com o plano cartesiano.

- Problema da distância radial

  Uma vez posicionada a tartaruga na direção correta o próximo e ultimo paso será move-la para frente até que ela alcance a posição (x, y) desejada.

1.2.2 Código

```
#!/usr/bin/env python

#Libraries
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math

class Turtle:
    def __init__(self):
        self.pose = Pose()
        self.vel = Twist()
        self.goal_pose = Pose()

        rospy.init_node("gotogoal")
```

```
28          self.rate = rospy.Rate(10) # 10Hz
29          self.velocity_publisher = rospy.Publisher("/turtle1/
                cmd_vel", Twist, queue_size = 10)
30          self.pose_subscriber = rospy.Subscriber("/turtle1/pose
                ", Pose, self.pose_callback)
31
32      def pose_callback(self,data):
33          self.pose.x = data.x
34          self.pose.y = data.y
35          self.pose.theta = data.theta
36
37      def distance(self, pose, goal_pose):
38          return math.sqrt((goal_pose.x - pose.x) * (goal_pose.x
                - pose.x) + (goal_pose.y - pose.y) * (goal_pose.y
               - pose.y))
39
40      def angle(self, pose, goal_pose):
41          return math.atan2(goal_pose.y - pose.y, goal_pose.x -
                pose.x)
42
43      def moveToGoal(self):
44          self.goal_pose.x = float(input("Set your x goal: "))
45          self.goal_pose.y = float(input("Set your y goal: "))
46
47          while abs(self.angle(self.pose, self.goal_pose) - self
                .pose.theta) >= 0.05 and not rospy.is_shutdown():
48              self.vel.angular.z = 1
49              self.velocity_publisher.publish(self.vel)
50
51          self.vel.angular.z = 0
52          while self.distance(self.pose, self.goal_pose) >= 0.3
                and not rospy.is_shutdown():
53
54              # Set velocity
55              self.vel.linear.x = 1
56
57              self.velocity_publisher.publish(self.vel)
58              self.rate.sleep()
59
60          # When get out, is near the goal pose
61          self.vel.linear.x = 0
62          self.vel.angular.z = 0
63          self.velocity_publisher.publish(self.vel)
64          #print(vel.angular.z)
65
66      #=== end class ====
67
68  if __name__ == '__main__':
69
70      turtle = Turtle()
71      turtle.moveToGoal()
```

Observação importante: Quando esse script estiver rodando basta utilizar ctrl + c para parar sua execução. Isso funciona graças ao código

```
72 and not  rospy.is_shutdown ()
```

adicionado em cada while do programa.

1.3 Rodando o programa

Para esta secção recomenda-se utlizar o programa terminator.

Para instalar basta abrir um terminal e fazer:

```
73 sudo add-apt-repository ppa:gnome-terminator
74 sudo apt-get update
75 sudo apt-get install terminator
```

Observação:No terminator: Utilize (ctrl + shift + o) para criar uma nova aba abaixo. Utilize (ctrl + shift + e) para criar uma nova aba ao lado.

Mas com ou sem terminator, para rodar o prorgama basta seguir as instruções a seguir:

- roscore Abra um terminal e digite:

  ```
  76     roscore
  ```

  Esse comando inicializa o ROS. De fato o ROS1 é baseado numa estrutura de grafo centralizada num nó chamado roscore ou (coração do ROS) que deve sempre ser inicializado e executado ao longo de toda a utilização dos packges do ROS.

- turtlesim_node

  Abra outro terminal e digite

  ```
  77     rosrun turtlesim turtlesim_node
  ```

  Esse comando roda o node da tartaruga. Surgirá uma tartaruga na tela.

- node de controle

  Abra outro terminal e digite

  ```
  78     rosrun turtle_control_demo1 turtle_control_demo1.py
  ```

  Esse comando irá rodar o script.

Como primeira tentativa digite <1>,<enter>,<1>,<enter> e observe que a tartaruga irá girar e depois se deslocar até o pobto $(x = 1, y = 1)$.

Para verificar se tudo deu certo utilize o comando **rostopic list** para listar os tópicos disponíveis. Queremos verificar a poscição, então utilizamos o tópico **/turtle1/pose** e fazemos **rostopic echo /turtle1/pose**.

1.4 Extra: Construção passo a passo do Node de Controle

1.4.1 Estudando os nodes e tópicos a serem utilizados

Nessa secção vamos explorar o passo a passo para contruir um node em python para interagir com qualquer robô através dor ROS.

Inicialmente iniciaremos o grafo do ROS utilizando o comando **roscore**. E também, vamos ligar o sistema robótico que desejamos controlar. No nosso caso desejamos controlar a tartaruga do **turtlesim_node**. Para isso, em outro terminal iremos rodar **rosrun turtlesim turtlesim_node**.

Figura 1: Tela Turtlesim: Tartaruga a ser controlada

Com tudo ligado ao ROS vamos agora analisar as inforamções disponíveis com **rostopic list**. Além disso, com **rosnode list** verificamos que além do /rosout (node que serve para reportar logs) temos ativo o /turtlesim.

```
zolubas@zolubas-532U3C-532U4C-532U3X:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Figura 2: Listando Tópicos do ROS

De posse dos tópicos disponíveis iremos acessar a documentação do node associado a um elemento físico que desejamos controlar. Aqui temos apenas a simulação da tartaruga, mas o procedimento é análogo. Em geral essas documentações serão do seu projeto ou estarão no github caso esteja usando o código de outrém. No caso do turtlesim basta acessar: `http://wiki.ros.org/turtlesim`

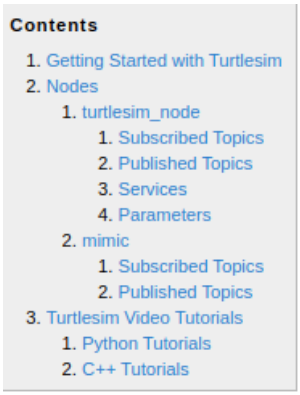


Figura 3: Documentação do Turtlesim na roswiki

A seguir analizaremos os *topics*.

- Subscribed Topics

  Documentação: turtleX/cmd_vel (geometry_msgs/Twist) The linear and angular command velocity for turtleX. The turtle will execute a velocity command for 1 second then time out. Twist.linear.x is the forward velocity, Twist.linear.y is the strafe velocity, and Twist.angular.z is the angular velocity.

  Comentário: Analisando a documentação vemos que podemos impor velocidades lineares nas direções x e y e velocidade angular em trono do eixo z. Aqui fica evidente que podemos alterar as velocidades da tartaruga, porém não podemos impor uma posição no plano XY diretamente.

- Published Topics

  Documentação: turtleX/pose (turtlesim/Pose) The x, y, theta, linear velocity, and angular velocity of turtleX.

  Comentário: Analisando a documentação vemos que é possível obter informações das posições x,y e angular θ além das velocidades lineares e angular. Do ponto de vista multivariavel pensando-se em espaço de estados é como se tivessemos acesso ao estado todo em termos de observação.

Por completude listam-se a seguir os *services* e *parameters*.

- Services

  - clear (st_srvs/Empty) Clears the turtlesim background and sets the color to the value of the background parameters.

  - reset (std_srvs/Empty) Resets the turtlesim to the start configuration and sets the background color to the value of the background.

  - kill (turtlesim/Kill) Kills a turtle by name.

  - spawn (turtlesim/Spawn) Spawns a turtle at (x, y, theta) and returns the name of the turtle. Also will take name for argument but will fail if a duplicate name.

  - turtleX/set_pen (turtlesim/SetPen) Sets the pen's color (r g b), width (width), and turns the pen on and off (off).

  - turtleX/teleport_absolute (turtlesim/TeleportAbsolute) Teleports the turtleX to (x, y, theta).

  - turtleX/teleport_relative (turtlesim/TeleportRelative) Teleports the turtleX a linear and angular distance from the turtles current position.

- Parameters

  ˜background_b (int, default: 255) Sets the blue channel of the background color.

  ˜background_g (int, default: 86) Sets the green channel of the background color.

  ˜background_r (int, default: 69) Sets the red channel of the background color.

Feita a análise já sabemos que iremos fazer um controle de posição baseado em velocidade. Note que os services de *teleport* não tem interesse prático e por tanto não serão utilizados.

Na documentação já vimos que o tipo da mensagem contida no tópico /turtle1/pose é turtlesim/pose. Vimos também que o tipo da mensagem contida no tópico /turtle1/cmd_vel é geometry_msgs/Twist.

Todavia é possível obter essa mesma informação com o comando **rostopic info <nome do tópico>**

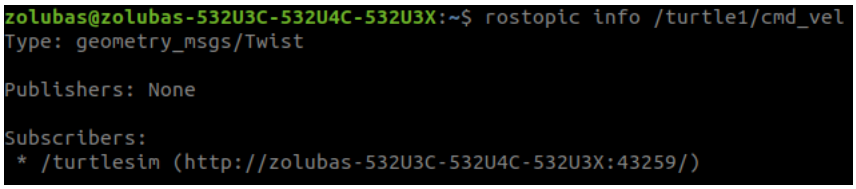


Figura 4: Informações sobre um tópico.

No item **type** é mostrado o tipo de mensagem do tópico. Podemos olhar a estrutura da mensagem usando **rosmsg show <nome da mensagem>**

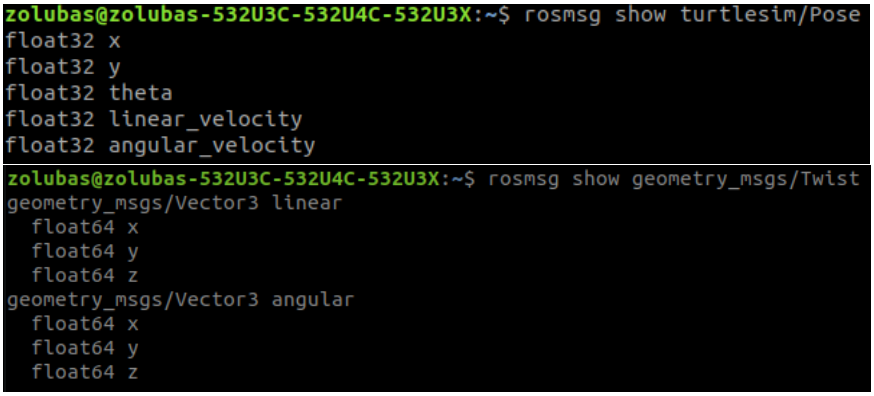


Figura 5: Detalhes das mensagens que são publicadas por *publishers* e lidas por *subscribers*.

Um outro caminho possível é explorar melhor a *ROS wiki* como mencionado anteriormente. Façamos isso no caso do turtlesim a titulo de exemplo:

Entrando em `http://wiki.ros.org/turtlesim` encontraremos a documentação do node turtlesim e explorando a página veremos:

2. Nodes

New in ROS hydro As of *Hydro* turtlesim uses the geometry_msgs/Twist message instead of its own custom one (*turtlesim/Velocity* in *Groovy* and older). Also the topic has been changed to `cmd_vel` (instead of `command_velocity` before).

2.1 turtlesim_node

`turtlesim_node` provides a simple simulator for teaching ROS concepts.

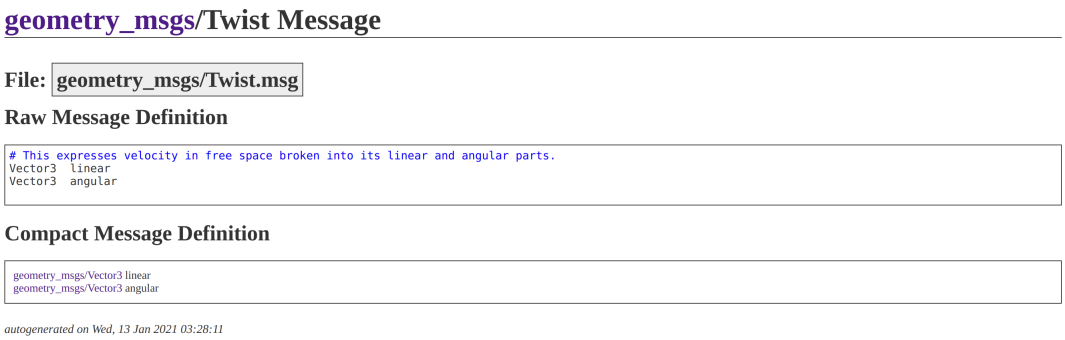
2.1.1 Subscribed Topics

`turtleX/cmd_vel` (geometry_msgs/Twist)

The linear and angular command velocity for turtleX. The turtle will execute a velocity command for 1 second then time out. Twist.linear.x is the forward velocity, Twist.linear.y is the strafe velocity, and Twist.angular.z is the angular velocity.

Figura 6: Explicação dos tópicos do ROS. Na imagem, explicação do topic chamado *cmd_vel* relacionado a inputs de velocidade no turtlesim

Vemos que esse topic possui mensagens do tipo **geometry_mgs** e do subtipo **geometry_msg /Twist** cujo link é fornecido na roswiki em azul como se vê na imagem acima. Clicando no link visualiza-se a seguinte página:



geometry_msgs/Twist Message

File: geometry_msgs/Twist.msg

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

autogenerated on Wed, 13 Jan 2021 03:28:11

Figura 7: Explicação da mensagem do tipo **geometry_msg /Twist**

Nesse arquivo podemos ver o **File** chamado **geometry_msg /Twist**.

Para importar esse a biblioteca **geometry_msg** de mensagens que contem mensagens do subtipo **/Twist** escrevemos no topo de um script em python:

```
79 import rospy
80 from geometry_msgs.msg import Twist
```

Note que a receita aplicada a outras mensagens de outros topics que não o *cmd_vel* é:

```
81 from <tipo da mensagem>.msg import <subtipo da mensagem>
```

Note também que é do **File** que retiramos as informações de qual é a mensagem e qual é o subtipo dela para colocarmos no python:

File: **geometry_msg /Twist**

File: **<tipo da mensagem> /<subtipo da mensagem>**

1.4.2 Interligando o scirpit em python com o ROS

Feito esse estudo inicial do robô que desejamos controlar, em que entendemos quais são as informações diposníveis, em quais tópicos elas estão, qual o papel de cada tópico (subscriber/publisher) e também quais e como são as mensagens de cada um dos tópicos que iremos utilizar finalmente podemos escrever o primeiro bloco de código do *turtle_control_demo*1.*py* para importar as bibliotecas a serem utilizadas.

```
82 #Habilitar ROS no python
83 import rospy
```

```
84 #Habilitar geometry_msgs/Twist
85 from geometry_msgs.msg import Twist
86 #Habilitar turtlesim/Pose
87 from turtlesim.msg import Pose
88 #Habilitar funções matemáticas
89 import math
```

Trabalhando numa estrutura da chamada Programação Orientada a Objetos criaremos uma classe chamada **Turtle**.

O construtor da classe Turtle instancia um vetor de posições lidas num objeto chamado **pose**, um vetor de posição-objetivo num objeto chamado **goal_pose** e um vetor de velocidades num objeto chamado **vel**. Além disso, o comando **rospy.init_node("gotogoal")** é o responsável por inicializar o node com o nome **gotogoal** que deve ser único. É possível chamar rospy.init_node com o argumento **anonymous=True** para que o ROS se encarregue de gerar um número aleatório adicionado no fim do nome do seu node de forma a torná-lo único.

Cria-se também a variavel **rate** para guardar a frequência de amostragem de informações vindas dos tópicos do ROS. Nesse caso escolheu-se a frequência de 10Hz.

E ainda são instanciadas objetos que tem a estrutura de tópico de modo a permitir o interfaciamento entre o node em python e os tópicos do ROS. Criaram-se nesse exemplo **velocity_publisher** e **pose_subscriber**. Em ambos os casos os argumentos são, nesta ordem, **topic name**, **Message type** e **queue_size** que define quantas mensagens são guardadas ao mesmo tempo. Quanto maior o queue_size maior a robustez e a latência da comunicação. Em geral utiliza-se um número baixo para esse argumento a menos que a frequência de publicação de mensagens seja muito elevada ou que a aplicação em questão necessite de informações do passado para funcionar.

Já o argumento **callback** é um método da classe (ou função se não estiver usando POO) que é chamado automaticamente pelo ROS sempre que uma nova mensagem é publicada num tópico no qual o node é um subscriber. O motivo pelo qual cabe a nós definirmos o callback é por que o usuário deve especificar como gostaria de receber a mensagem e em quais variváveis ou objetos gostaria de guardar as informações provenientes da mensagem recebida.

Uma explicação oficial para implementações de publihers/subscribers pode ser encontrada em

`http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29`

Nota: Em geral é comum abreviar os nomes para conjuntos de três letras de modo que velocity_publisher poderia ser apenas vel_pub.

```
90 class Turtle:
91     def __init__(self):
92         self.pose = Pose()
93         self.vel = Twist()
94         self.goal_pose = Pose()
95 
96         rospy.init_node("gotogoal")
97         self.rate = rospy.Rate(10)
98         self.velocity_publisher = rospy.Publisher("/turtle1/
               cmd_vel", Twist, queue_size = 10)
99         self.pose_subscriber = rospy.Subscriber("/turtle1/pose
               ", Pose, self.pose_callback)
```

1.4.3 Implementando o algorítimo de Controle

Implementadas todas as declarações e funções/métodos necessários para que o script em python possa se communicar com as informações diposníveis através da estrutura do ROS temos um conjunto de variáveis e objetos que contém todas as informações necessárias para implementarmos o controle. Sendo assim, basta prosseguir normalmente com a programação do script sem se preocupar com o ROS.

Geometria

A seguir implementa-se o cálculo da distância euclidiana no plano XY fazendo-se

$$d = \sqrt{(x_{goal} - x_{atual})^2 + (y_{goal} - y_{atual})^2} \tag{1}$$

```
100     def distance(self, pose, goal_pose):
101         return math.sqrt((goal_pose.x - pose.x) * (goal_pose.x
                - pose.x) + (goal_pose.y - pose.y) * (goal_pose.y
               - pose.y))
```

E também calcula-se o angulo formado entre o vetor que sai da origem e aponta para (x_{atual}, y_{atual}) e (x_{goal}, y_{goal}).

$$\theta_z = atan2((y_{goal} - y_{atual}), (x_{goal} - x_{atual})) \tag{2}$$

```
102     def angle(self, pose, goal_pose):
103         return math.atan2(goal_pose.y - pose.y, goal_pose.x -
               pose.x)
```

A figura a seguir mostra geometricamente as grandezas envolvidas nos cálculos dos dois métodos mostrados acima.

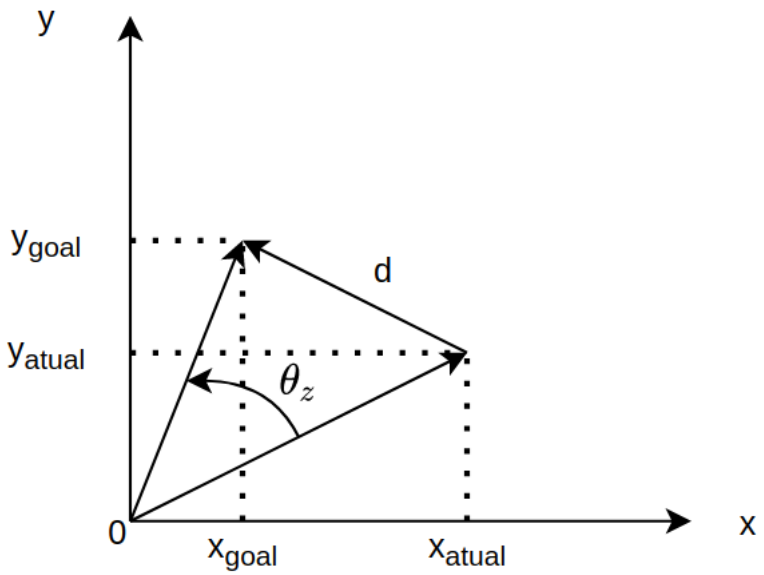


Figura 8: Geometria do problema.

Notar que d e θ_z são as grandezas calculadas, mas não foram utilizadas variáveis para guardar esses valore no código. Ao invés disso, chamam-se os métodos **distance** e **angle** sempre que se deseja saber esses valores.

O método de Controle

Essencialmente baseado na ideia de coordenadas polares como já mencionado, o método a seguir implementa a ação de rotacionar a tartaruga em torno do eixo z até que o angulo entre o vetor posição-objetivo e o vetor posição-atual seja nulo (no caso menor que 0.05 rad). E feito isso, de mover a tartaruga para frente até que a distância entreo vetor posição-objetivo e o vetor posição-atual seja nula (no caso menor que 0.3 cm).

Note que para ir para frente basta solicitar uma velocidade positiva na direção X pois ao alinhar ambos os vetores (angulo nulo), rotacionou-se o sistema de coordenadas da tartaruga de tal sorte que o vetor x desse sistema ficou paralelo com o vetor dado por $(x_{goal}, y_{goal}) - (x_{atual}, y_{atual})$.

```
104     def moveToGoal(self):
105 
106     #Pede-se a posição-objetivo
107         self.goal_pose.x = float(input("Set your x goal: "))
108         self.goal_pose.y = float(input("Set your y goal: "))
```

```
109
110     #Rotacione (anti-horario) até angulo ser nulo
111         while abs(self.angle(self.pose, self.goal_pose) - self
                .pose.theta) >= 0.05 and not rospy.is_shutdown():
112             #Set angular velocity
113             self.vel.angular.z = 1
114             #Publique a velocidade vel no tópico cmd_vel
115             self.velocity_publisher.publish(self.vel)
116
117     #Com angulo nulo, pare de rotacionar -> vel.angular = 0
118         self.vel.angular.z = 0
119
120     #Vá para frente até distancia ser nulo]a
121         while self.distance(self.pose, self.goal_pose) >= 0.3
                and not rospy.is_shutdown():
122
123             # Set velocity
124             self.vel.linear.x = 1
125             #Publique a velocidade vel no tópico cmd_vel
126             self.velocity_publisher.publish(self.vel)
127             #Espere para publicar novamente
128             self.rate.sleep()
129
130         # Quando chegar no destino zere todas as velocidades
131         self.vel.linear.x = 0
132         self.vel.angular.z = 0
133         self.velocity_publisher.publish(self.vel)
134         #print(vel.angular.z)
```

2 Controle PID

É o controle mais popular na industria. Possui a vantagem de poder ser relacionado a especificações temporais e ser derivado de operações simples da disciplina de cálculo. Dessa forma, é possível realizar um ajuste intuitivo de seus parâmetros empriricamente já que é fácil compreender o efeito de cada um deles no sistema, mas também é possível fazer projetos sofisticados levando-se em conta especificações no domínio do tempo e até mesmo no domínio da frequência.

De acordo com o padrão ISA o PID é especificado pela formula a seguir:

$$\frac{V_m(s)}{E(s)} = K_p \times (1 + \frac{1}{T_I \times s} + \frac{T_D \times s}{1 + \frac{T_D \times s}{N}}) \tag{3}$$

E tem três parâmetros a serem ajustados: K_p, T_I e T_D. Por questões de ruído em regal a implementação conta com um quarto parâmetro $N \in [3, 20]$.

Adicionalmente um ponto que é fundamental é que na prática de robótica existem varios efeitos não lineares em que comprometem a precisão máxima que um controlador pode ter ao controlar a posição de um robô qualquer que seja. Nesse caso isso não é considerado por conta da simplicidade do turtlesim_node.

2.1 Controle Proporcional Polar

Nesta secção implementou-se um controlador proporcional utilizando-se a mesma estratégia do capitulo 1. Ou seja, o problema bidimensional foi dividido em 2 problemas unidimensionais. O primeiro problema consiste num ajuste da posição angular e o segundo consiste no ajuste da posição radial de modo que a tartaruga alcance uma posição (x, y) desejada.

2.1.1 O controlador

O controle proporcional foi implementado nestas secções do código:

Primeiramente implementa-se o controle proporcional para acertar o angulo

$$\begin{cases} e_k = angle(pose, goal\_pose) - pose.theta \\ \omega_z = e_k \end{cases}$$

No código é importante notar que a variável **ros_zero** teve que ser multiplicada por 1000 dado que a precisão máxima da velocidade angular é inferior a precisão máxima da velocidade linear. Com isso, há um erro que não pode ser corrigido inserido no controle em razão da estratégia polar adotada.

```
135 while abs(self.angle(self.pose, self.goal_pose) - self.pose.
    theta) >= 1000*self.ros_zero and not rospy.is_shutdown():
136     # Set angular velocity by proportional control   rad/s
137     self.vel.angular.z = self.angle(self.pose, self.goal_pose)
            - self.pose.theta
138     self.velocity_publisher.publish(self.vel)
139 self.vel.angular.z = 0
```

E em seguida, uma vez que o versor $\hat{x}$ está orientado na direção de (y_{goal}, x_{goal}) a tartaruga é direcionada para frente através do loop a seguir que implementa o segundo controlador proporcional.

$$\begin{cases} e_k = |goal\_pose.x - pose.x| \\ v_x = e_k \end{cases}$$

```
140 while abs(self.distance_x(self.pose, self.goal_pose)) > self.
    ros_zero and not rospy.is_shutdown():
141 
142     # Set velocity by proportional control 10[(m/s)/m]*
        distnace()[m] = vel [m/s]
143     self.vel.linear.x = 1*self.distance_x(self.pose, self.
        goal_pose)
144     self.velocity_publisher.publish(self.vel)
145     self.rate.sleep()
146 
147 # When get out, is near the goal pose
148 self.vel.linear.x = 0
149 self.vel.angular.z = 0
150 self.velocity_publisher.publish(self.vel)
```

2.1.2 O código completo: turtle_proportional_angularControl.py

No código vale ressaltar que a função **distance_x** implementa a medida do módulo da distância na direção de $\hat{x}$. E como mencionado a precisão angular é 1000 vezes inferior do que a precisão radial, isto é, quando a tartaruga se move em relação ao eixo $\hat{x}$.

```python
#!/usr/bin/env python

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

class Turtle:
    def __init__(self):
        self.pose = Pose()
        self.vel = Twist()
        self.goal_pose = Pose()

        rospy.init_node("gotogoal")
        self.rate = rospy.Rate(10)
        self.velocity_publisher = rospy.Publisher("/turtle1/
           cmd_vel", Twist, queue_size = 10)
        self.pose_subscriber = rospy.Subscriber("/turtle1/pose
           ", Pose, self.pose_callback)

    def pose_callback(self,data):
        self.pose.x = data.x
        self.pose.y = data.y
        self.pose.theta = data.theta

    def distance(self, pose, goal_pose):
        return math.sqrt((goal_pose.x - pose.x) * (goal_pose.x
            - pose.x) + (goal_pose.y - pose.y) * (goal_pose.y
           - pose.y))

    def distance_x(self, pose, goal_pose):
        return  abs(self.goal_pose.x - self.pose.x )

    def angle(self, pose, goal_pose):
        return math.atan2(goal_pose.y - pose.y, goal_pose.x -
           pose.x)

    def moveToGoal_proportionalRotational(self):
        #Similar then moveToGoal but works with P controler on
            angular position

        #Error tolerance using ROS precision
        self.ros_zero = 0.000000001 #5.544444561

        self.goal_pose.x = float(input("Set your x goal: "))
        self.goal_pose.y = float(input("Set your y goal: "))

        while abs(self.angle(self.pose, self.goal_pose) - self
           .pose.theta) >= 1000*self.ros_zero and not rospy.
           is_shutdown():
```

```
193             # Set angular velocity by proportional control   rad/
                    s
194                 self.vel.angular.z = self.angle(self.pose, self.
                        goal_pose) - self.pose.theta
195                 self.velocity_publisher.publish(self.vel)
196 
197             self.vel.angular.z = 0
198             while abs(self.distance_x(self.pose, self.goal_pose))
                    > self.ros_zero and not rospy.is_shutdown():
199 
200                 # Set velocity by proportional control 10[(m/s)/m
                        ]*distnace()[m] = vel [m/s]
201                 self.vel.linear.x = 1*self.distance_x(self.pose,
                        self.goal_pose)
202                 self.velocity_publisher.publish(self.vel)
203                 self.rate.sleep()
204 
205             # When get out, is near the goal pose
206             self.vel.linear.x = 0
207             self.vel.angular.z = 0
208             self.velocity_publisher.publish(self.vel)
209 
210         #=====
211 
212 if __name__ == '__main__':
213 
214     turtle = Turtle()
215     turtle.moveToGoal_proportionalRotational()
```

2.1.3 Resultados

A tartaruga começa por padrão na posição $(x, y) = (5.5, 5.5)$. Chamamos a rotina turtle_proportional_angularControl.py 3 vezes. Solicitamos (2,7), (8,6) e (3,3).

A seguir é mostrada a trajetória realizada e os status de posição após cada chamada de turtle_proportional_angularControl.py.

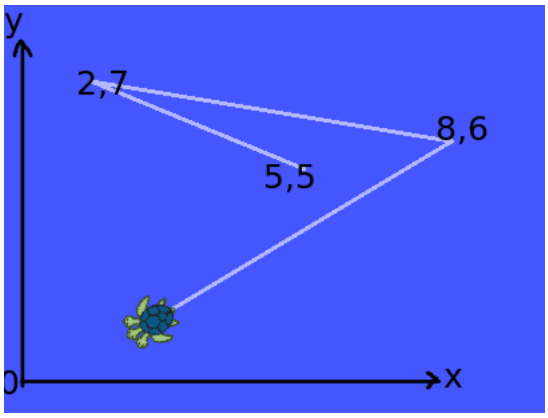


Figura 9: Trajetória Controle Polar Proporcional

- goal (2,7)

  Result:

  x: 2.0

  y: 7.0000038147

  theta: 2.75193119049

  linear_velocity: 0.0

  angular_velocity: 0.0

- goal (8,6)

  Result:

  x: 8.0

  y: 6.00000619888

  theta: -0.165148302913

  linear_velocity: 0.0

  angular_velocity: 0.0

- goal (3,3)

  Result:

  x: 3.0

  y: 2.99999284744

  theta: -2.60117125511

  linear_velocity: 0.0

angular_velocity: 0.0

O resultado principal a ser observado é que a escolha da estratégia polar sempre leva a um erro não nulo na coordenada y pois a precisão máxima que se consegue em termos angulares é inferior a precisão máxima nas direções de $\hat{x}$ e de $\hat{y}$. Note que o erro acontece na coordenada y pois o controle radial é feito por meio da coordenada x que se torna exatamente a coordenada radial ρ de (ρ, θ) e que consegue por tanto utilizar a precisão máxima permitida pelo ROS.

Além disso, fazendo-se outros testes percebeu-se que esse controlador não é robusto e pode levar a problemas de posicionamento da tartaruga em razão de erros numéricos que interferem na convergência do algorítimo. Uma sugestão de otimização é trabalhar com tolerâncias maiores (por exemplo 1×10^{-4}) ou implementar outras restrições para impedir quebra de convergência do algorítimo. Nessa implementação foi utilizada uma tolerância de 1×10^{-9} para posição radial e de 1×10^{-6} para posição angular.