00 Simulation scripts

June 22, 2021

Information: *Reading notes for follow_waypoints.py and a brief summary for past and future work in the drone boat simulation project*

Written by: *Zihao Xu*

Last update date: *06.22.2021*

1 follow_waypoints.py

1.1 Import

1.1.1 Message types embedded in ROS

`geometry_msgs/Point.msg`

Type	Name
float64	x
float64	y
float64	z

- Represents the **position** of a point in free space

`geometry_msgs/Quaternion.msg`

Type	Name
float64	x
float64	y
float64	z
float64	w

- Represents an **orientation** in free space in **quaternion** form
- In short, unit quaternions provide a convenient (though not intuitive) mathematical notation for representing spatial orientations and rotations of elements in three dimensional space
- For detailed information, one available reference is the wikipedia taking about *Quaternions and spatial rotation*

`geometry_msgs/Pose.msg`

Type	Name
geometry_msgs/Point	position
geometry_msgs/Quaternion	orientation

- A representation of **pose** in free space, composed of position and orientation

`geometry_msgs/PoseWithCovariance.msg`

Type	Name
geometry_msgs/Pose	pose
float64[36]	covariance

- Represent the **pose** in free space **with uncertainty**
- The 6×6 **covariance matrix** is represented in row-major form
- Use a fixed-axis representation for the orientation
- In order, the parameters are
$$(x, y, z, R, P, Y)$$
  - R stands for *rolling*, meaning the rotation about X axis
  - P stands for *pitching*, meaning the rotation about Y axis
  - Y stands for *yawing*, meaning the rotation about Z axis

`geometry_msgs/Vector3.msg`

Type	Name
float64	x
float64	y
float64	z

- Represents a vector in free space
- It is only meant to represent a **direction**
- It does make sense to apply a translation to it
  - When applying a generic rigid transformation to a *Vector3*, only the rotation will be applied

`geometry_msgs/Twist.msg`

Type	Name
geometry_msgs/Vector3	linear
geometry_msgs/Vector3	angular

- Expresses **velocity** in free space broken into its linear and angular parts

geometry_msgs/TwistWithCovariance.msg

Type	Name
geometry_msgs/Twist	twist
float64[36]	covariance

- Represent the **velocity** in free space **with uncertainty**
- The 6×6 **covariance matrix** is represented in row-major form
- Use a fixed-axis representation for the orientation
- In order, the parameters are

$$(x, y, z, R, P, Y)$$

  - R stands for *rolling*, meaning the rotation about X axis
  - P stands for *pitching*, meaning the rotation about Y axis
  - Y stands for *yawing*, meaning the rotation about Z axis

geometry_msgs/Transform.msg

Type	Name
geometry_msgs/Vector3	translation
geometry_msgs/Quaternion	rotation

- Represent the transform between **two coordinate frames** in free space

std_msgs/Header.msg

Type	Name
uint32	seq
time	stamp
string	*frame_id*

- Generally used to communicate **timestamped** data in a **particular coordinate frame**
- *seq*: Sequence ID, consecutively increasing ID
- *stamp*: Two-integer timestamp that is expressed s:
  - *stamp.secs*: seconds (stamp secs) since epoch
  - *stamp.nsecs*: nanoseconds since stamp_secs
- *frame_id*: Frame this data is associated with

nav_msgs.msg.Odometry

Type	Name
std_msgs/Header	header
string	child_frame_id
geometry_msgs/PoseWithCovariance	pose
geometry_msgs/TwistWithCovariance	twist

- Represents an **estimate** of a **position and velocity** in free space

- *pose* should be specified in the coordinate frame given by *header.frame_id*
- *twist* should be specified in the coordinate frame given by the *child_frame_id*

trajectory_msgs/MultiDOFJointTrajectoryPoint.msg

Type	Name
geometry_msgs/Transform[ ]	transforms
geometry_msgs/Twist[ ]	velocities
geometry_msgs/Twist[ ]	*accelerations*
duration	time_from_start

- Represent a fully defined state point for a **multi-joint robot**, including **positions, velocities and accelerations** for for all joints
- *transforms*: Each multi-dof joint can specify a transform (up to 6 DOF)
- *velocities*: There can be a velocity specified for the origin of the joint
- *accelerations*: There can be an acceleration specified for the origin of the joint

trajectory_msgs/MultiDOFJointTrajectory.msg

Type	Name
std_msgs/Header	header
string[ ]	joint_names
trajectory_msgs/MultiDOFJointTrajectoryPoint[ ]	points

- The *header* is used to specify the coordinate frame and the reference time for the trajectory durations
- Use a series of fully defined state points to specify a **multi-dof joint trajectory**
- The order and length of every point must be same as the order of length as the *joint_names* array

1.1.2 Packages used in execution

rospy

- A pure Python client library for ROS, enables Python programmers to quickly interface with ROS Topics, Services and Parameters.
- For full documents, refer to ROS Wiki

numpy

- A pack used for array computation and is widely known about

tf

- A package in ROS that lets the user keep track of multiple coordinate frames over time
- *tf* maintains the relationship between coordinate frames in a tree structure buffered in time
- Lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time
- For full documents, refer to the ROS Wiki and python docs about tf

1.1.3 Final imports

Here are the reorganized imports. The first line is meant to assign python interpreters.

[ ]:
```python
#!/usr/bin/env python

# Python library for ROS
import rospy
# Necessary messages types
from geometry_msgs.msg import Transform, Twist
from nav_msgs.msg import Odometry
from trajectory_msgs.msg import MultiDOFJointTrajectoryPoint,␣
 ↪MultiDOFJointTrajectory
# Necessary Packages
import tf
import numpy as np
```

Here are the unnecessary message types and module initially imported in the scripts. For potential reference, list them here.

[ ]:
```python
""" ***** Unused message types and modules *****
import sensor_msgs.point_cloud2
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped
from sensor_msgs.msg import PointCloud2
from geometry_msgs.msg import PoseWithCovarianceStamped
from visualization_msgs.msg import Marker
from pursuit_msgs.msg import PursuitPlan
from geometry_msgs.msg import PoseStamped
from Trackers import MultiTracker
from tf.transformations import quaternion_matrix
from tf.transformations import quaternion_from_matrix
from std_srvs.srv import Empty
"""
```

1.2 Function: poly_path(...)

- **Description**: This function is not invoked in scripts I've seen so I'm not sure what it is for

```
[ ]: def poly_path(points, v0, max_vel, dt):
         """
         - Final velocity of each edge should point to next point
         - Mean velocity between edges should be max_vel

         """
         trans = points[0].reshape(-1, 3)
         vel = v0.reshape(-1, 3)
         acc = np.zeros((1, 3))
         time = np.zeros((1, 1))
         t0 = 0.0
         for i in range(1, points.shape[0]):
             s0 = points[i - 1, :]
             s1 = points[i, :]

             ds = np.sqrt(np.sum((s0 - s1)**2))
             t1 = ds / max_vel
             v1 = max_vel * (s1 - s0) / ds

             tt = np.arange(dt, t1, dt).reshape(-1, 1)

             a = (3 * s1 - 3 * s0 - 3 * t1 * v0 + t1 * v0 - t1 * v1) / t1 ** 2
             b = (2 * s0 - 2 * s1 + t1 * v0 + t1 * v1) / t1 ** 3

             ss = s0 + v0 * tt + a * tt**2 + b * tt**3
             vv = v0 + 2 * a * tt + 3 * b * tt**2
             aa = 2 * a + 6 * b * tt

             trans = np.vstack((trans, ss))
             vel = np.vstack((vel, vv))
             acc = np.vstack((acc, aa))
             time = np.vstack((time, tt + t0))

             v0 = v1
             t0 += t1

         return trans, vel, acc, time
```

1.3 Function: traj_message(...)

- **Description**:
  - Given a series of collections of position, velocity, acceleration and time, produce a list of standard ROS messages of trajectory points, which can be then used to build a standard ROS trajectory message.
- **Inputs**:
  - *trans*: A $3 \times n$ matrix, the $[x, y, z]^T$ positions of n time steps
  - *vel*: A $3 \times n$ matrix, the velocities on $[x, y, z]^T$ directions of n time steps
  - *acc*: A $3 \times n$ matrix, the accelerations on $[x, y, z]^T$ directions of n time steps
  - *time*: A vector of length n, time steps
- **Outputs**:
  - A list of *trajectory_msgs/MultiDOFJointTrajectoryPoint.msg*

[ ]:
```
def traj_message(trans, vel, acc, time):
    """
    Produce a list of trajectory points according to given arrays
    of positions, velocities, accelerations and corresponding times
    """
    # Create a blank list
    traj_points = []

    # Create the trajectory points for each time step in loop
    for i in range(time.shape[0]):
        # Create a blank trajectory point
        tpts = MultiDOFJointTrajectoryPoint()

        # Set the duration time of each trajectory point
        ''' ********** Doubt **********
        It seems that a fixed length of the time step is expected,
        but the right hand side is providing an increasing length?
        The time provided to this function is an array created by np.arange()
        '''
        tpts.time_from_start = rospy.Duration(time[i])

        # Initialize the messages needed for a trajectory point
        tpts.transforms = [Transform()]
        tpts.velocities = [Twist()]
        tpts.accelerations = [Twist()]

        # The rotation represented by a quaternion
        tpts.transforms[0].rotation.w = 1.0
        tpts.transforms[0].rotation.x = 0.0
        tpts.transforms[0].rotation.y = 0.0
        tpts.transforms[0].rotation.z = 0.0

        # The translation in three directions
        tpts.transforms[0].translation.x = trans[i, 0]
        tpts.transforms[0].translation.y = trans[i, 1]
        tpts.transforms[0].translation.z = trans[i, 2]

        # Speeds in three directions
        tpts.velocities[0].linear.x = vel[i, 0]
        tpts.velocities[0].linear.y = vel[i, 1]
        tpts.velocities[0].linear.z = vel[i, 2]

        # Accelerations in three directions
        tpts.accelerations[0].linear.x = acc[i, 0]
        tpts.accelerations[0].linear.y = acc[i, 1]
        tpts.accelerations[0].linear.z = acc[i, 2]

        # Add this point to the trajectory points
        traj_points.append(tpts)

    return traj_points
```

1.4 Class: LocalPlanner

1.4.1 ___init___(self)

[ ]:
```
class LocalPlanner(object):
    # Initiate the class when created
    def __init__(self):
        """
        Initiate the class when created
        """
        # Get the parameters from private namespace
        # If the parameters don't exist, use the default values
        # THe syntax is rospy.get_param("name","default value")
        self.dt = rospy.get_param("~dt", 0.1)
        self.max_vel = rospy.get_param("~velocity", 2.0)
        self.inertial_frame = rospy.get_param("~inertial_frame", "map")
        self.base_frame = rospy.get_param("~base_frame", "base_link")

        # Initialize the position in order of [x, y ,z]
        self.position_base = np.array([10, 0, 5.0])

        # Subscribes to the "/tf" message topic
        # Calls tf.Transformer.setTransform() with each incoming
        # transformation message
        ''' ********** Modification **********
        Currently I do not see why this is necessary while it leads to
        a flood of warnings about ignored repeated data
        Therefore, I commented this out in my simulation code
        '''
        self.listener = tf.TransformListener()

        # Set the publisher to topic "command/trajectory"
        # The publish message type is set to MultiDOFJointTrajectory
        # Queue size is set to avoid data lost due to connection issues
        self.pub_cmd = rospy.Publisher('command/trajectory',␣
 ↪MultiDOFJointTrajectory, queue_size=10)

        # Initialize the velocities and accelerationes
        self.position, self.velocity = np.zeros(3), np.zeros(3)

        # Subscribe to the "odometry" topic
        # The received message type is supposed to be Odemetry
        # The received message is sent to the function self.odom_cb
        rospy.Subscriber('odometry', Odometry, self.odom_cb)

        # Sleep for 2 seconds
        rospy.sleep(2.0)
```

1.4.2 odom_cb(self, msg)

- Input: Message of type `nav_msgs.msg.Odometry`

[ ]:
```
    def odom_cb(self, msg):
        """
        Reorganize the position and velocity information from subscription
        """
        # Get the position information
        self.position = np.array([
            msg.pose.pose.position.x,
            msg.pose.pose.position.y,
            msg.pose.pose.position.z])
        # Get the velocity information
        self.velocity = np.array([
            msg.twist.twist.linear.x,
            msg.twist.twist.linear.y,
            msg.twist.twist.linear.z])
```

1.4.3 loop(self, ...)

- Given the initial position, provides and **publishes** a trajectory which consists of a sinusoidal trajectory in y direction and a sinusoidal trajectory in z direction. Amplitudes and periods of the sinusoidal trajectories can be controlled by input arguments.
- **Inputs**:
  - *trans0*: A vector of length 3, the initial position in form of $[x, y, z]$
  - *width*: A *float* number, the amplitude of trajectory in y direction
  - *height*: A *float* number, the amplitude of trajectory in z direction
  - *period*: A *float* number, the period for the sinusoidal trajectories

[ ]:
```
    def loop(self, trans0, width, height, period):
        """
        Create and publish a trajectory described by input arguments
        """
        # Create the blank trajectory message
        cmd = MultiDOFJointTrajectory()
        # Set the joint names
        cmd.joint_names = ["robot_link"]
        # Set the header of the trajectroy message
        cmd.header.frame_id = self.inertial_frame
        cmd.header.stamp = rospy.Time.now()

        # The duration of each trajectory point is 0.1s
        time = np.arange(0, period, 0.1)

        # No movements in x direction
        x = np.zeros(time.shape) + trans0[0]
        # A sinusoidal trajectory in y direction
        y = (width / 2) * np.sin(2 * np.pi * time * 2.0 / period) + trans0[1]
        # A sinusoidal trajectory in z direction
        z = -(height / 2) * np.cos(2 * np.pi * time / period) + trans0[2]
        # Get the mixed trajectory
        trans = np.vstack((x, y, z)).T

        # The velocities seems to be the derivative of the trajectories
        vx = 0.0 * x
        vy = (width / 2) * (2 * np.pi * 2.0 / period) * np.cos(2 * np.pi * time␣
↪* 2.0 / period)
        vz = (height / 2) * (2 * np.pi / period) * np.sin(2 * np.pi * time /␣
↪period)
        vel = np.vstack((vx, vy, vz)).T

        # The accelerations seems to be the derivative of the velocities
        ax = 0.0 * x
        ay = -(width / 2) * (2 * np.pi * 2.0 / period) ** 2 \
            * np.sin(2 * np.pi * time * 2.0 / period)
        az = (height / 2) * (2 * np.pi / period) ** 2 * np.cos(2 * np.pi * time␣
↪/ period)
        acc = np.vstack((ax, ay, az)).T

        # Get the list of standard ROS messages of trajectory points
        cmd.points = traj_message(trans, vel, acc, time)

        # Publish the trajectory to topic "command/trajectory"
        self.pub_cmd.publish(cmd)
```

1.5 Main program

Commands which would be executed when directly running this script (not executed when imported)

[ ]:
```python
if __name__ == "__main__":
    # Initialize the ROS node for the process with name "local_planner"
    # The "anonymous" argument adds a random number to the end of name
    # so that the name is ensured to be unique
    rospy.init_node('local_planner', anonymous=True)

    # Create an instance from the class "LocalPlanner"
    gp = LocalPlanner()

    # Set the rate to be 0.1 hz
    r = rospy.Rate(1.0 / 10)
    while not rospy.is_shutdown():
        # Publish a trajectory needs 10 seconds to finish
        gp.loop((4.0, 0.0, 2.0), 3.0, 2.0, 10.0)
        # Wait for 10 seconds
        r.sleep()
        # The rest are similar
        gp.loop((6.0, 0.0, 2.0), 3.0, 2.0, 10.0)
        r.sleep()
        gp.loop((8.0, 0.0, 2.0), 3.0, 2.0, 10.0)
        r.sleep()
        gp.loop((6.0, 0.0, 2.0), 3.0, 2.0, 10.0)
        r.sleep()
```

2 Work Summary

2.1 Completed

- Set the ROS environment and test the simulation
- Get familiar with the current controlling scripts
  - Necessary message types
  - Subscriptions, publishers and topics related to this node
  - How the trajectory in current simulation is built

2.2 To do

2.2.1 ROS interface

- Look carefully into the launch file
  - The function of each node
  - Is it necessary to simulate the 4-rotor UAV (*hunter*) in the meantime while we're going to use the 6-rotor UAV (*firefly*) in reality?
  - Only keep the *necessary* nodes to simplify further development?
- View the topic lists to check if there are other topics controlling the movements
  - command/motor_speed
  - command/pose
  - command/trajectory
  - ...
- Think about the controlling strategy needed for obstacle avoiding and boat following(?)
  - Build a trajectory for each movement based on slow but accurate detection
  - Directly control small movements based on fast detections
  - ...

2.2.2 CNN

- Figure out the detailed requirements for the network
  - Handle the bias lying in the image datasets (most consist of rivers and boats)
  - Simple localization for the boat, segmentation according to the river shape, ...
  - Efficiency on specified devices
  - ...
- Check available datasets which can be used to train the networks

2.3 Further Plan

2.3.1 ROS interface

- Consider how to implement online classification via CNN in ROS efficiently

2.3.2 CNN Design

- Choose an appropriate network structure according to requirements and available datasets
  - A highly customized but simple network trained on custom datasets
  - CNN based on anchor boxes such as yolo v5
  - MLP with tranformer such as MLP-Mixer
  - [Possible] Adjust some requirements because of network limitations

- Train the network on some custom datasets to ensure performance
- [Possible] Try network pruning to improve efficiency
- Apply the network in ROS