Day 24 CS570

Review for Test 2 and work time on Unit Tests

Topics for the test on Friday

- Protocol - what are they and when to use them
- Gyros - How to use information about gyros to detect motion of the robot
- PIDControllers - what are they when to use them.
  - Understand the main methods used with PIDController `setSetpoint`, `setTolerance`, `calculate`, `Kp`,`Ki`, `Kd`
  - Understand how to manipulate `Kp`,`Ki`, `Kd` to make your robot perform better
- The purpose of a `RobotContainer`
  - How to use a SendableChooser on the SmartDashboard
- Creating Unit Tests
  - The three essential parts of any unit test *Setup*, *Action*, *Assert*
  - How to use *Fixtures* to simplify making unit tests
  - How to use *MonkeyPatch* to make methods that can return values to test parts of the code.
  - How to use *parametrize* to make multiple tests with different values

Test Expectations The test will be written on paper, and there will probably be some questions that ask you to write code. I will be looking for generally good syntax, but will be generous on syntax mistakes, but look for proper logic and use of relevant code structures (*important methods, decorators, etc.*)

I expect that the test will be about 5 or 6 free response questions.

Example questions

Protocols

1. Explain the purpose of creating the `AutonomousRoutine` protocol in our examples with Romi robots.
2. Explain the difference between a **Protocol** and **Class**.

Gyro

3. Explain how the axes x, y, and z relate to the turns that a robot makes. Feel free to draw a picture.

PIDController

4. A robot is supposed to cross a balance beam. The robot has a weight that it move from side to side that it can use to move its center of gravity via a command `balanceweight.move(x:float)`. The motion of the robot along the beam is on the x axis of the robot. Discuss how you might use write an autonomous routine similar to `climbramp` or `drivestraight` that would allow the robot to cross the beam.
5. A student is writing code to make a robot balance and their code looks something like:

```
while True:
    gyro_rate = get_gyro_reading()  # Read gyro rate in degrees per second
    if gyro_rate > 0:
        # Robot is leaning forward, adjust motors to move backward
        adjust_motors(-1)
    elif gyro_rate < 0:
        # Robot is leaning backward, adjust motors to move forward
        adjust_motors(1)
    else:
        # Robot is balanced, motors stay unchanged
        adjust_motors(0)
```

Explain why using a PIDController might be better than the code above.

RobotContainer

6. Fill in the blanks to make this **RobotContainer** work:

```
import wpilib

from autoroutine import AutoRoutine
from drivestraight import DriveStraight
from drivetrain import Drivetrain
from gyroturn import GyroTurn


class RobotContainer:

    def __init__(self) -> None:
        self.controller = wpilib.Joystick(0)
        # Create SmartDashboard chooser for autonomous routines
        self.chooser = wpilib._____________()
        self.drivetrain = Drivetrain()
        self._configure()

    def _configure(self):
        self.chooser._____________________("Twist 90 degrees", GyroTurn(self.drivetrain, 90))
        self.chooser._____________________("Go straight 2m", DriveStraight(self.drivetrain, 2))
        wpilib.SmartDashboard._________________(self.chooser)

    def get_autonomous(self) -> AutoRoutine:
        return self.chooser._____________()
```

Unit Tests

7. Looking at the code below write a test for the `set_raw_output`

```
class SparkMax(PIDMotor):
    """
    Wrapper class for the SparkMax motor controller
    """
    motor: CANSparkMax
    encoder: SparkMaxRelativeEncoder
    pid_controller: SparkMaxPIDController

    def __init__(self, can_id: int, inverted: bool = True, brushless: bool = True, config: SparkMaxConfig =
        """

        Args:
            can_id (int): The CAN ID of the motor controller
            inverted (bool, optional): Whether the motor is inverted. Defaults to True.
            brushless (bool, optional): Whether the motor is brushless. Defaults to True.
            config (SparkMaxConfig, None): The configuration for the motor controller. Defaults to None.
        """
        super().__init__()
        self._can_id = can_id
        self._inverted = inverted
        self._brushless = brushless
        self._config = config

    def init(self):
```

```
        """
        Initializes the motor controller, pid controller, and encoder
        """
        self.motor = CANSparkMax(
            self._can_id,
            CANSparkMax.MotorType.kBrushless if self._brushless else CANSparkMax.MotorType.kBrushed
        )
        self.motor.setInverted(self._inverted)
        self.pid_controller = self.motor.getPIDController()
        self.encoder = self.motor.getEncoder()
        self._set_config(self._config)

    def set_raw_output(self, x: float):
        """
        Sets the raw output of the motor controller

        Args:
            x (float): The output of the motor controller (between -1 and 1)
        """
        self.motor.set(x)

    def set_target_position(self, pos: rotations):
        """
        Sets the target position of the motor controller in rotations

        Args:
            pos (float): The target position of the motor controller in rotations
        """
        self.pid_controller.setReference(pos, CANSparkMax.ControlType.kPosition)

    def set_target_velocity(self, vel: rotations_per_second):  # Rotations per minute??
        """
        Sets the target velocity of the motor controller in rotations per second

        Args:
            vel (float): The target velocity of the motor controller in rotations per second
        """
        self.pid_controller.setReference(vel, CANSparkMax.ControlType.kVelocity)

    def get_sensor_position(self) -> rotations:
        """
        Gets the sensor position of the motor controller in rotations

        Returns:
            (rotations): The sensor position of the motor controller in rotations
        """
        return self.encoder.getPosition()

    def set_sensor_position(self, pos: rotations):
        """
        Sets the sensor position of the motor controller in rotations

        Args:
            pos (rotations): The sensor position of the motor controller in rotations
        """
        self.encoder.setPosition(pos)
```

```
    def get_sensor_velocity(self) -> rotations_per_second:
        """
        Gets the sensor velocity of the motor controller in rotations per second

        Returns:
            (rotations_per_second): The sensor velocity of the motor controller in rotations per second
        """
        return self.encoder.getVelocity()
```

8. Explain the meaning and the uses of these ideas from unit testing:

- Fixture
- Parametrize
- MonkeyPatch

9. A student is thinking of writing a unit test for the `get_value` method written below. Explain how the ideas of MonkeyPatch and Parametrize would be useful in making the test.

```
import wpilib


class LimitSwitch:
    """
    Wrapper class for I2C Limit Switches
    """

    def __init__(self, port: int, inverted: bool = True):
        """Wrapper class for I2C Limit Switches
        Args:
            port (int): I2C port of the limit switch.
            inverted (bool, optional): Return the inverted boolean of output. Defaults to True.
        """
        self.limit_switch = wpilib.DigitalInput(port)
        self.reverse = inverted

    def get_value(self):
        """Return if the limit switch is pressed or if object is detected (in the case of non-tactile senso

        Returns:
            bool: True if pressed, False if not.
        """
        if self.reverse:
            return not self.limit_switch.get()
        return self.limit_switch.get()
```

10. Write a fixture that might be used for this class in tests.

```
class SparkMax(PIDMotor):
    """
    Wrapper class for the SparkMax motor controller
    """
    motor: CANSparkMax
    encoder: SparkMaxRelativeEncoder
    pid_controller: SparkMaxPIDController

    def __init__(self, can_id: int, inverted: bool = True, brushless: bool = True, config: SparkMaxConfig =
        """
```

```
        Args:
            can_id (int): The CAN ID of the motor controller
            inverted (bool, optional): Whether the motor is inverted. Defaults to True.
            brushless (bool, optional): Whether the motor is brushless. Defaults to True.
            config (SparkMaxConfig, None): The configuration for the motor controller. Defaults to None.
        """
        super().__init__()
        self._can_id = can_id
        self._inverted = inverted
        self._brushless = brushless
        self._config = config

    def init(self):
        """
        Initializes the motor controller, pid controller, and encoder
        """
        self.motor = CANSparkMax(
            self._can_id,
            CANSparkMax.MotorType.kBrushless if self._brushless else CANSparkMax.MotorType.kBrushed
        )
        self.motor.setInverted(self._inverted)
        self.pid_controller = self.motor.getPIDController()
        self.encoder = self.motor.getEncoder()
        self._set_config(self._config)

    def set_raw_output(self, x: float):
        """
        Sets the raw output of the motor controller

        Args:
            x (float): The output of the motor controller (between -1 and 1)
        """
        self.motor.set(x)

    def set_target_position(self, pos: rotations):
        """
        Sets the target position of the motor controller in rotations

        Args:
            pos (float): The target position of the motor controller in rotations
        """
        self.pid_controller.setReference(pos, CANSparkMax.ControlType.kPosition)

    def set_target_velocity(self, vel: rotations_per_second):  # Rotations per minute??
        """
        Sets the target velocity of the motor controller in rotations per second

        Args:
            vel (float): The target velocity of the motor controller in rotations per second
        """
        self.pid_controller.setReference(vel, CANSparkMax.ControlType.kVelocity)

    def get_sensor_position(self) -> rotations:
        """
        Gets the sensor position of the motor controller in rotations

        Returns:
```

```
        (rotations): The sensor position of the motor controller in rotations
    """
    return self.encoder.getPosition()

def set_sensor_position(self, pos: rotations):
    """
    Sets the sensor position of the motor controller in rotations

    Args:
        pos (rotations): The sensor position of the motor controller in rotations
    """
    self.encoder.setPosition(pos)

def get_sensor_velocity(self) -> rotations_per_second:
    """
    Gets the sensor velocity of the motor controller in rotations per second

    Returns:
        (rotations_per_second): The sensor velocity of the motor controller in rotations per second
    """
    return self.encoder.getVelocity()
```