Wombat Mecanum Wheel Control System: Python Implementation and Omnidirectional Motion Algorithms

Oscar Yan, **Muyao Zhang**, **Oliver Yan**

As the new Botball utilizes the new Mecanum Wheel System compared to the regular wheels, it is crucial to understand and know how to use its functions. Our paper aims to share our knowledge and help others regarding the new system.

What is Omnidirectional Movement?

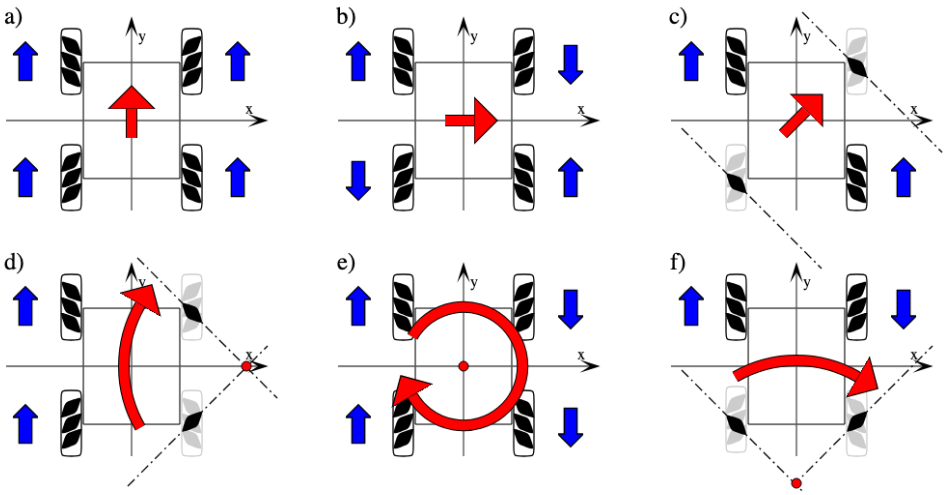
Omnidirectional movement refers to the ability to move in any direction forward, backward, sideways, and diagonally without needing to turn first.

Mecanum wheels are a special type of wheel that enable this by using rollers set at a 45-degree angle around each wheel. By spinning the wheels at different speeds and directions, the robot can move in any direction smoothly.

This mechanic is very useful not only in Botball as it allows for more efficiency and prevention in clashing into obstacles but also in many real life scenarios such as in the military or everyday transport.

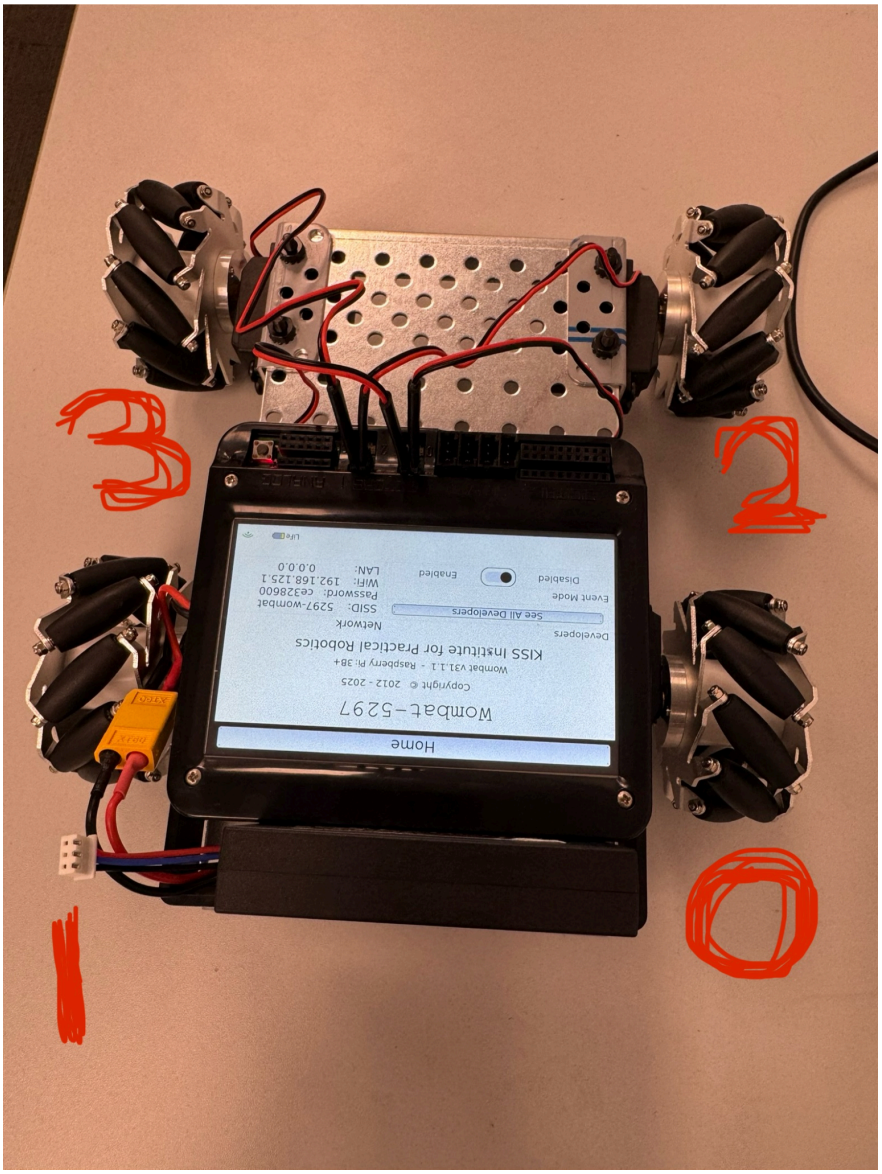
Initial Move

Our initial move was to test each motor to see if they were working, it was really important to have matching, corresponding wires so that the code was executed correctly. For each motor, we tested simple code that allowed it to move, either forwards or backwards. After all of them were working, we found this diagram which helped us a lot in finding how to code for new movements.



This diagram clearly displays all the necessary ports to move to activate a certain movement, for example going right diagonal would need the top left motor to go forward and bottom right to go forward.

Our Port Setting



Here is the following code for figure a - f in the diagram
a)

```
def forward(speed=80, duration=800):
    k.motor(0, -speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, speed)
    k.msleep(duration)
    stop_motors()
```

b)

```
def left(speed=80, duration=800):
    k.motor(0, speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, -speed)
    k.msleep(duration)
    stop_motors()
```

c)

```
def right_diagonal():
    k.motor(0, 300)
    k.motor(1, 0)
    k.motor(2, 0)
    k.motor(3, 300)
    k.msleep(3200)
```

d)

```
def right_turn():
    k.motor(0, 300)
    k.motor(1, 0)
    k.motor(2, 0)
    k.motor(3, 300)
    k.msleep(3200)
```

e)

```
def whole_right():
    k.motor(0, -300)
    k.motor(1, 300)
    k.motor(2, -300)
    k.motor(3, 300)
    k.msleep(3200)
```

f)

```
def whole_right():
    k.motor(0, 0)
    k.motor(1, 0)
    k.motor(2, -300)
    k.motor(3, 300)
    k.msleep(3200)
```

Common Issues

We have faced some issues that hindered the performance of robots such as, as mentioned before, not matching wires and ports that do not correspond with the code.

Additionally, debris may be caught up in the mecanum wheel mechanism which does not make it work well.

Other smaller issues such as the robot not turning or sliding well required fine-tuning skills. This is slightly tweaking the arguments for certain functions and observing the output.

Our First Code

```
#!/usr/bin/python3
import os, sys
import math
sys.path.append("/usr/lib")
import _kipr as k

# Basic movement functions
def forward(speed=80, duration=800):
    k.motor(0, -speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, speed)
    k.msleep(duration)
    stop_motors()

def backward(speed=80, duration=800):
    k.motor(0, speed)
    k.motor(1, -speed)
    k.motor(2, speed)
    k.motor(3, -speed)
    k.msleep(duration)
    stop_motors()

def left(speed=80, duration=800):
    k.motor(0, speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, -speed)
    k.msleep(duration)
    stop_motors()

def right(speed=80, duration=800):
    k.motor(0, -speed)
    k.motor(1, -speed)
    k.motor(2, speed)
    k.motor(3, speed)
    k.msleep(duration)
```

```
    stop_motors()

# Omnidirectional slide function
def slide_move(angle, speed=80, duration=800):
    rad = math.radians(angle)

    vx = math.cos(rad)
    vy = math.sin(rad)

    motor0_speed = (-vx - vy) * speed
    motor1_speed = (vx - vy) * speed
    motor2_speed = (-vx + vy) * speed
    motor3_speed = (vx + vy) * speed

    motor0_speed = max(-100, min(100, motor0_speed))
    motor1_speed = max(-100, min(100, motor1_speed))
    motor2_speed = max(-100, min(100, motor2_speed))
    motor3_speed = max(-100, min(100, motor3_speed))

    k.motor(0, int(motor0_speed))
    k.motor(1, int(motor1_speed))
    k.motor(2, int(motor2_speed))
    k.motor(3, int(motor3_speed))

    k.msleep(duration)
    stop_motors()

# Rotation function
def turn(angle, speed=80):
    time_per_degree = 7500.0 / 360.0 * (80.0 / speed)
    duration = int(abs(angle) * time_per_degree)

    if angle > 0:  # Right turn (clockwise)
        k.motor(0, -speed)
        k.motor(1, -speed)
        k.motor(2, -speed)
        k.motor(3, -speed)
    else:  # Left turn (counter-clockwise)
        k.motor(0, speed)
        k.motor(1, speed)
        k.motor(2, speed)
        k.motor(3, speed)

    k.msleep(duration)
    stop_motors()

# Stop all motors
def stop_motors():
```

```
    k.motor(0, 0)
    k.motor(1, 0)
    k.motor(2, 0)
    k.motor(3, 0)

def main():
    # Test basic movements
    '''
    forward(80, 1000)
    k.msleep(500)

    backward(80, 1000)
    k.msleep(500)

    left(80, 1000)
    k.msleep(500)

    right(80, 1000)
    k.msleep(500)

    # Test diagonal movements
    slide_move(45, 80, 1000)
    k.msleep(500)

    slide_move(135, 80, 1000)
    k.msleep(500)

    slide_move(225, 80, 1000)
    k.msleep(500)

    slide_move(315, 80, 1000)
    k.msleep(500)

    # Test various angle movements
    slide_move(30, 80)      # Right-forward 30 degrees
    k.msleep(400)

    slide_move(-30, 80)     # Left-forward 30 degrees
    k.msleep(400)

    slide_move(60, 80)      # Right-forward 60 degrees
    k.msleep(400)

    slide_move(-60, 80)     # Left-forward 60 degrees
    k.msleep(400)
    '''
    # Test rotations
    turn(90, 80)
```

```
    k.msleep(500)

    turn(-90, 80)
    k.msleep(500)

    turn(45, 80)
    k.msleep(500)

    turn(-45, 80)
    k.msleep(500)

    turn(180, 60)
    k.msleep(500)

    # Complex movement patterns
    forward(100, 500)
    turn(90, 80)
    forward(100, 500)
    turn(90, 80)
    forward(100, 500)
    turn(90, 80)
    forward(100, 500)
    k.msleep(1000)

    # Circle pattern with slide moves
    for i in range(8):
        slide_move(i * 45, 80, 300)
        k.msleep(200)

    # Figure-8 pattern
    slide_move(45, 80, 1000)
    turn(90, 60)
    slide_move(315, 80, 1000)
    turn(-90, 60)
    slide_move(45, 80, 1000)
    turn(-90, 60)
    slide_move(315, 80, 1000)

    stop_motors()

if __name__ == "__main__":
    main()
```

Explanation

First we tested for basic movements such as going forward, backwards left and right. Although simple, It took quite some attempts to nail it down.

We implemented the Omnidirection function to achieve full omnidirectional control. This function accepts an angle in degrees and uses trigonometric functions to calculate the x and y velocity components (vx and vy) based on that angle. These components are then used to compute the individual speeds for each motor, taking into account the 45-degree roller orientation of mecanum wheels. This calculation allows the robot to move smoothly in any direction, including diagonals and custom angles, while maintaining its orientation.

To add rotation, we created the turn function. This rotates the robot around its center by setting all four motors to spin in the same direction. The duration of the turn is calculated based on the desired angle and a time-per-degree ratio, which we adjusted based on motor speed. This function allows the robot to perform precise angle turns without any linear displacement.

Additionally, we used the circle slide_move function in a loop to perform a circular pattern by sliding at 45-degree intervals around a full 360 degrees. This demonstrated the robot's ability to follow curved paths by combining directional vectors. It also confirmed the effectiveness of our angle-based motion control using mecanum wheels.

Upgraded Code

However, to ensure more safety and efficiency, we have upgraded our previous code. The newer code incorporates an acceleration mechanic and deceleration mechanic to ensure the bot moves efficiently, lowering the risk of accidental crashes or failure to halt in time. These mechanics will be extremely beneficial for teams who are trying to use the mecanum wheels to the max.

Here is our code with the acceleration and deceleration:

```
#!/usr/bin/python3
import os, sys
import math
sys.path.append("/usr/lib")
import _kipr as k

# Basic movement functions
def forward(speed=80, duration=800):
    k.motor(0, -speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, speed)
    k.msleep(duration)
    stop_motors()

def backward(speed=80, duration=800):
    k.motor(0, speed)
    k.motor(1, -speed)
```

```
    k.motor(2, speed)
    k.motor(3, -speed)
    k.msleep(duration)
    stop_motors()

def left(speed=80, duration=800):
    k.motor(0, speed)
    k.motor(1, speed)
    k.motor(2, -speed)
    k.motor(3, -speed)
    k.msleep(duration)
    stop_motors()

def right(speed=80, duration=800):
    k.motor(0, -speed)
    k.motor(1, -speed)
    k.motor(2, speed)
    k.motor(3, speed)
    k.msleep(duration)
    stop_motors()

# Omnidirectional slide function
def slide_move(angle, speed=80, duration=800):
    rad = math.radians(angle)

    vx = math.cos(rad)
    vy = math.sin(rad)

    motor0_speed = (-vx - vy) * speed
    motor1_speed = (vx - vy) * speed
    motor2_speed = (-vx + vy) * speed
    motor3_speed = (vx + vy) * speed

    motor0_speed = max(-100, min(100, motor0_speed))
    motor1_speed = max(-100, min(100, motor1_speed))
    motor2_speed = max(-100, min(100, motor2_speed))
    motor3_speed = max(-100, min(100, motor3_speed))

    k.motor(0, int(motor0_speed))
    k.motor(1, int(motor1_speed))
    k.motor(2, int(motor2_speed))
    k.motor(3, int(motor3_speed))

    k.msleep(duration)
    stop_motors()

# Smooth slide with acceleration and deceleration
def slide_move_smooth(angle, max_speed=80, duration=800):
```

```
rad = math.radians(angle)
vx = math.cos(rad)
vy = math.sin(rad)

# Time segments
accel_time = int(duration * 0.25)  # 25% acceleration
const_time = int(duration * 0.5)   # 50% constant speed
decel_time = int(duration * 0.25)  # 25% deceleration

steps = 10  # Number of steps for smooth transition

# Acceleration phase
for i in range(steps):
    current_speed = max_speed * (i + 1) / steps

    motor0_speed = (-vx - vy) * current_speed
    motor1_speed = (vx - vy) * current_speed
    motor2_speed = (-vx + vy) * current_speed
    motor3_speed = (vx + vy) * current_speed

    k.motor(0, int(motor0_speed))
    k.motor(1, int(motor1_speed))
    k.motor(2, int(motor2_speed))
    k.motor(3, int(motor3_speed))

    k.msleep(accel_time // steps)

# Constant speed phase
motor0_speed = (-vx - vy) * max_speed
motor1_speed = (vx - vy) * max_speed
motor2_speed = (-vx + vy) * max_speed
motor3_speed = (vx + vy) * max_speed

k.motor(0, int(motor0_speed))
k.motor(1, int(motor1_speed))
k.motor(2, int(motor2_speed))
k.motor(3, int(motor3_speed))

k.msleep(const_time)

# Deceleration phase
for i in range(steps):
    current_speed = max_speed * (steps - i) / steps

    motor0_speed = (-vx - vy) * current_speed
    motor1_speed = (vx - vy) * current_speed
    motor2_speed = (-vx + vy) * current_speed
    motor3_speed = (vx + vy) * current_speed
```

```
        k.motor(0, int(motor0_speed))
        k.motor(1, int(motor1_speed))
        k.motor(2, int(motor2_speed))
        k.motor(3, int(motor3_speed))

        k.msleep(decel_time // steps)

    stop_motors()

# PID-based smooth movement with distance control
def slide_move_pid(angle, distance_mm, max_speed=80):
    # PID constants - tune these for your robot
    Kp = 0.8   # Proportional gain
    Ki = 0.1   # Integral gain
    Kd = 0.3   # Derivative gain

    # Convert angle to radians
    rad = math.radians(angle)
    vx = math.cos(rad)
    vy = math.sin(rad)

    # Timing
    dt = 50  # milliseconds per loop

    # PID variables
    error_sum = 0
    last_error = distance_mm

    # Estimated distance traveled
    distance_traveled = 0
    speed_to_distance = 0.5  # Calibrate this value for your robot

    while abs(distance_mm - distance_traveled) > 5:  # 5mm tolerance
        # Calculate error
        error = distance_mm - distance_traveled

        # PID calculation
        P = Kp * error
        I = Ki * error_sum
        D = Kd * (error - last_error) / dt

        # Calculate speed
        speed = P + I + D
        speed = max(-max_speed, min(max_speed, speed))

        # Apply smooth limits near target
        if abs(error) < 50:  # Within 50mm of target
```

```
            speed = speed * (abs(error) / 50)  # Linear reduction

        # Set motor speeds
        motor0_speed = (-vx - vy) * speed
        motor1_speed = (vx - vy) * speed
        motor2_speed = (-vx + vy) * speed
        motor3_speed = (vx + vy) * speed

        k.motor(0, int(motor0_speed))
        k.motor(1, int(motor1_speed))
        k.motor(2, int(motor2_speed))
        k.motor(3, int(motor3_speed))

        # Update distance estimate
        distance_traveled += abs(speed) * speed_to_distance * dt / 1000.0

        # Update PID variables
        error_sum += error * dt
        last_error = error

        k.msleep(dt)

    stop_motors()

# Rotation function
def turn(angle, speed=80):
    time_per_degree = 7500.0 / 360.0 * (80.0 / speed)
    duration = int(abs(angle) * time_per_degree)

    if angle > 0:  # Right turn (clockwise)
        k.motor(0, -speed)
        k.motor(1, -speed)
        k.motor(2, -speed)
        k.motor(3, -speed)
    else:  # Left turn (counter-clockwise)
        k.motor(0, speed)
        k.motor(1, speed)
        k.motor(2, speed)
        k.motor(3, speed)

    k.msleep(duration)
    stop_motors()

# Stop all motors
def stop_motors():
    k.motor(0, 0)
    k.motor(1, 0)
    k.motor(2, 0)
```

```
    k.motor(3, 0)

def main():
    # Test basic movements
    forward(80, 1000)
    k.msleep(500)

    backward(80, 1000)
    k.msleep(500)

    left(80, 1000)
    k.msleep(500)

    right(80, 1000)
    k.msleep(500)

    # Test smooth movements
    slide_move_smooth(0, 80, 1500)     # Forward with smooth accel/decel
    k.msleep(500)

    slide_move_smooth(45, 80, 1500)    # Diagonal with smooth motion
    k.msleep(500)

    slide_move_smooth(90, 80, 1500)    # Right with smooth motion
    k.msleep(500)

    # Test PID movements (distance-based)
    slide_move_pid(0, 300, 80)         # Forward 300mm
    k.msleep(500)

    slide_move_pid(180, 300, 80)       # Backward 300mm
    k.msleep(500)

    slide_move_pid(45, 200, 60)        # Diagonal 200mm at lower speed
    k.msleep(500)

    # Comparison test: normal vs smooth
    slide_move(90, 80, 1000)           # Normal right movement
    k.msleep(1000)

    slide_move_smooth(90, 80, 1000)    # Smooth right movement
    k.msleep(1000)

    # Test rotation
    turn(90, 40)
    k.msleep(500)

    turn(-90, 40)
```

```
    k.msleep(500)

    stop_motors()

if __name__ == "__main__":
    main()
```

The updated version of our code keeps many of the same core functions as the original, such as moving forward, backward, left, and right.

The biggest difference in the new version is the introduction of smoother and more controlled movement. The slide_move_smooth function replaces the sudden start-and-stop motion of the original slide_move with gradual acceleration and deceleration. This function splits the movement into three parts: the first part slowly increases speed, the middle part keeps the robot moving at full speed, and the last part slows the robot down until it stops. These changes make the robot move more naturally and help prevent slipping, bouncing, or sudden shifts that could throw it off course. It's especially useful for moving over longer distances or when accuracy is important.

Another new feature in the updated code is slide_move_pid, which allows the robot to move a set distance in millimeters using a control method called PID(Proportional, Integral, and Derivative). This function constantly checks how far the robot is from its target and adjusts the speed to match. If the robot is far away, it moves quickly. As it gets closer, it slows down to avoid overshooting. This makes the movement more accurate and consistent, which is really useful when the robot needs to stop exactly at a certain point.

The turning function remains the same in both versions. It rotates the robot on the spot by spinning all four wheels in the same direction. The robot turns clockwise or counter-clockwise depending on whether the input angle is positive or negative. The speed and time are calculated based on the angle, which gives the robot control over how much it rotates.

Ultimately, the newer code adds smarter and smoother movement. The acceleration and deceleration make the robot feel more controlled and realistic, while the PID function adds accuracy when traveling specific distances. These features improve how the robot handles tasks and make it more reliable during competitions or real-world use. Compared to the original version, the robot is now better at avoiding errors, moving smoothly, and stopping exactly where it needs to.

References:

Wikipedia contributors. (2025, July 2). Mecanum wheel. In Wikipedia, The Free Encyclopedia. Retrieved July 8, 2025, from https://en.wikipedia.org/wiki/Mecanum_wheel

Game Manual 0 Contributors. (n.d.). *Mecanum TeleOp*. In *Game Manual 0* software tutorials. Retrieved July 8, 2025, from https://gm0.org/en/latest/docs/software/tutorials/mecanum-drive.html