

Human-Robot Interaction and Autonomous Systems

AI 616

Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

Lecture 6





Course Outlines

Module	Content	Simulation Lab
1. Introduction to Robotics	- Fundamentals of Robot Control - Classification of Robotics - Exploration of Robot Types	- Basic Robot Control - Modeling Robot Classes - Simulating Various Robot Types
2. Advanced HRI Studies	- Introductory to Advanced Concepts in HRI	- Basic to Complex HRI Scenarios
3. Social Dynamics in Robotics	- Architectures for Social Interaction - Case Studies and History of HRI	- Social Interaction Simulations - Historical Evolution in HRI
4. Strategic HRI System Design	- Core Principles and Communication Strategies in HRI	- Designing Interfaces - Communication Protocol Simulations
5. HRI Efficiency and Effectiveness	- Metrics for Mission Effectiveness and Human Factors	- Implementing and Analyzing HRI Metrics
6. Innovative HRI System Design	- Exploring Innovative Designs and Real-World Applications	- Testing Innovative Designs - Simulating Diverse Scenarios
7. HRI System Classification	- Developing Comprehensive Classifications	- Classification Application and Case Studies
8. Human Factors in HRI	- Analyzing Ergonomics and Human Factors	- Ergonomic Simulations - Scenario-Based Human Factor Analysis
9. Robotics and AI Integration	- Theoretical Concepts and Practical AI Applications	- AI Algorithm Implementation - AI-Driven HRI Modeling
10. HRI Case Studies	- Mobile Assistive Robot Studies	- Designing and Testing Mobile Assistive Robots





Webots

robot simulation

Tutorial 1: Moving the Robot Forward

Objective: Learn how to initialize your robot and make it move forward.

Step 1: Setting Up Your Webots Project

- Open Webots and create a new project.
- Add an e-puck robot to your scene from the Webots library.

Step 2: Creating Your Controller

- Navigate to **File > New > New Robot Controller...**
- Name your controller **move_forward**.
- Select Python as the programming language and confirm.

Step 3: Writing the Controller Code

Open the **move_forward.py** file generated by Webots and follow along:

```
# Import the Robot class to interact with the robot
from controller import Robot

# Create a robot instance to control it
robot = Robot()

# Define the simulation step time
TIME_STEP = 64

# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

This code initializes the robot but doesn't yet move it. Let's proceed to add movement.

```
# Import the Motor class to control the wheels
from controller import Robot, Motor

# Initialization steps as before

# Retrieve motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set the motors to velocity control mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
# Define a forward speed
FORWARD_SPEED = 6.28
# Apply the speed to both motors to move forward
leftMotor.setVelocity(FORWARD_SPEED)
rightMotor.setVelocity(FORWARD_SPEED)
```

```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Example 1:

```
# Import the Robot class to interact with the robot
from controller import Robot
```

```
# Create a robot instance to control it
robot = Robot()
```

```
# Define the simulation step time
TIME_STEP = 64
```

```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Example 2:

```
# Import the Motor class to control the wheels
from controller import Robot, Motor
```

```
# Initialization steps as before
```

```
# Retrieve motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set the motors to velocity control mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
# Define a forward speed
FORWARD_SPEED = 6.28
# Apply the speed to both motors to move forward
leftMotor.setVelocity(FORWARD_SPEED)
rightMotor.setVelocity(FORWARD_SPEED)
```

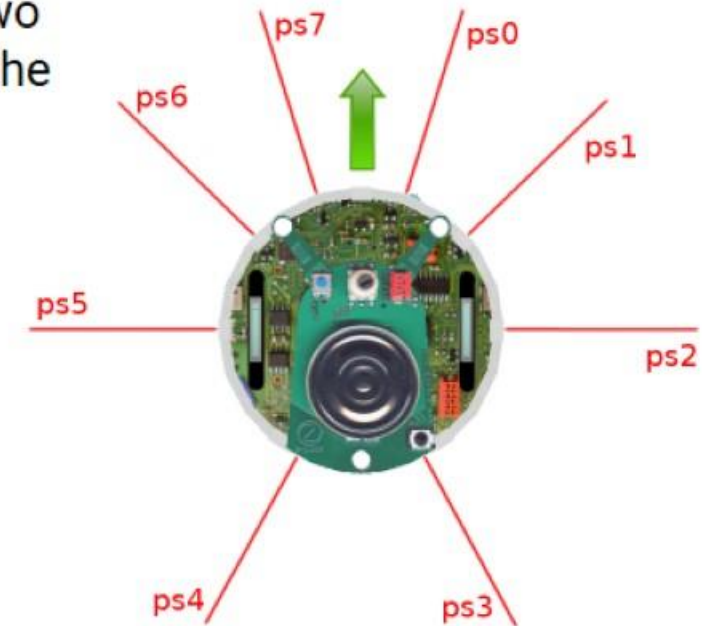
```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Objective: Modify the robot's behavior to stop when an obstacle is detected.

Step 1: Enhancements to the Controller

Build upon the **move_forward** controller by adding sensor functionality to detect obstacles.

Controller programming requires some information related to the e-puck model. In order to create the collision avoidance algorithm, we need to read the values of its **8 infra-red distance sensors located around** its turret, and we need to actuate its two wheels. The way that the distance sensors are distributed around the turret and the e-puck direction are depicted in **this figure**.



```
# Import necessary classes
```

```
from controller import Robot, Motor, DistanceSensor
```

```
# Initialization steps as before
```

```
# Initialize the front distance sensor
```

```
frontSensor = robot.getDevice('ps0')
```

```
frontSensor.enable(TIME_STEP)
```

```
# Control loop
```

```
while robot.step(TIME_STEP) != -1:
```

```
    # Read the sensor value
```

```
    distance = frontSensor.getValue()
```

```
    # Check for obstacles
```

```
    if distance < 800: # Adjust based on your environment
```

```
        # An obstacle is detected, stop the robot
```

```
        leftMotor.setVelocity(0)
```

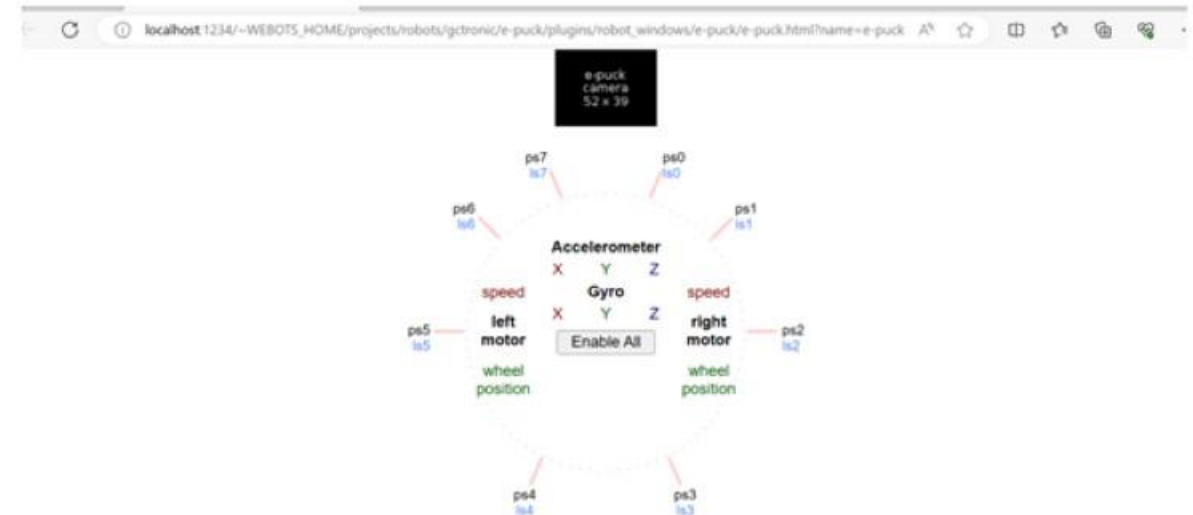
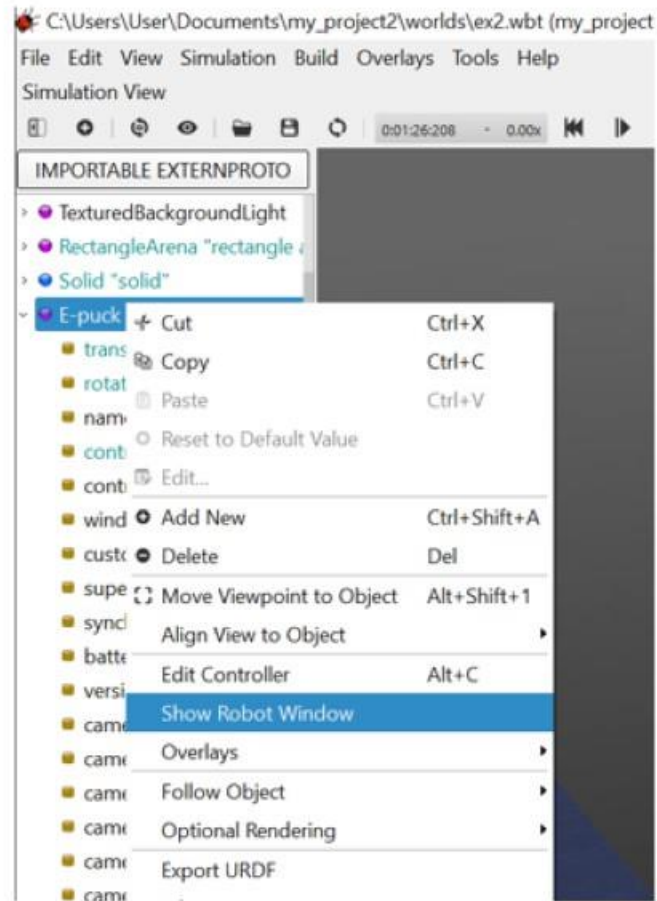
```
        rightMotor.setVelocity(0)
```

```
    else:
```

```
        # No obstacle, move forward
```

```
        leftMotor.setVelocity(FORWARD_SPEED)
```

```
        rightMotor.setVelocity(FORWARD_SPEED)
```



```
from controller import Robot, Motor, DistanceSensor

TIME_STEP = 64
MAX_SPEED = 6.28 # Adjusting MAX_SPEED for realism,
                  # assuming it's in radians/sec which is typical in
                  # simulations

# create the Robot instance.
robot = Robot()

# get a handler to the motors and set target position to
infinity (speed control)
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))

# Initialize and enable the distance sensor
distanceSensor = robot.getDevice('ps0') # Assuming
'sp0' is the correct sensor name
distanceSensor.enable(TIME_STEP)
```

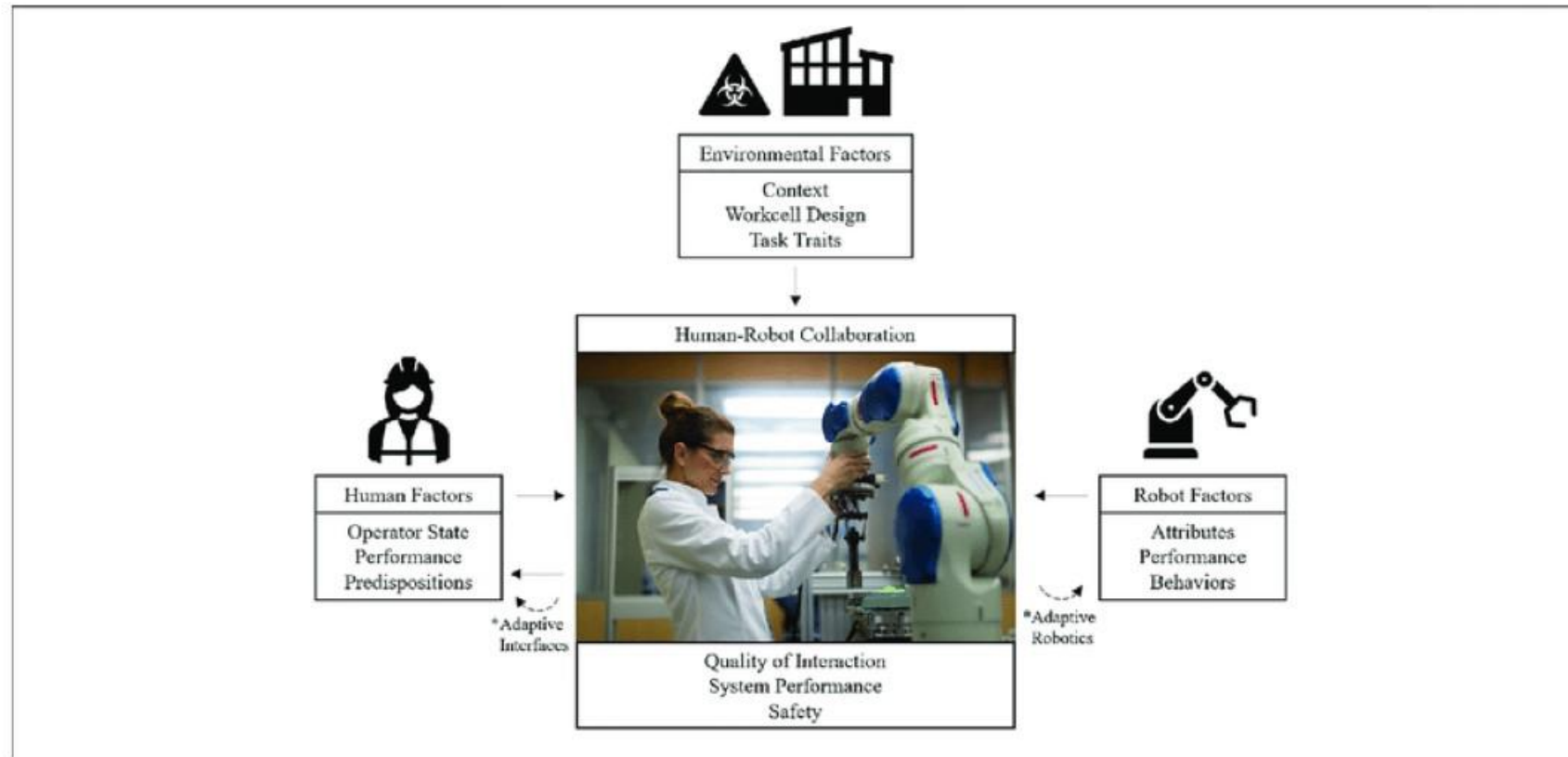
```
# set up the motor speeds at 10% of the MAX_SPEED.
leftMotor.setVelocity(0.1 * MAX_SPEED)
rightMotor.setVelocity(0.1 * MAX_SPEED)

while robot.step(TIME_STEP) != -1:
    # Read the sensor value
    distance = distanceSensor.getValue()
    print(distance)

    # Check for obstacles - assuming a threshold value for when to
    stop
    # You may need to adjust the threshold based on your
    simulation environment
    if distance < 62: # Example threshold value, adjust based on
        your sensor's range and sensitivity
        # An obstacle is detected, stop the robot
        leftMotor.setVelocity( MAX_SPEED)
        rightMotor.setVelocity(0)
    else:
        # No obstacle, continue moving forward
        leftMotor.setVelocity(0.1 * MAX_SPEED)
        rightMotor.setVelocity(0.1 * MAX_SPEED)
```



The concepts of HRI efficiency and effectiveness metrics for mission effectiveness and human factors.



How Fast Robots and Humans Complete Tasks **Together**:

Explanation: Checking if a robot and human can finish a job quickly, like cleaning up a room.

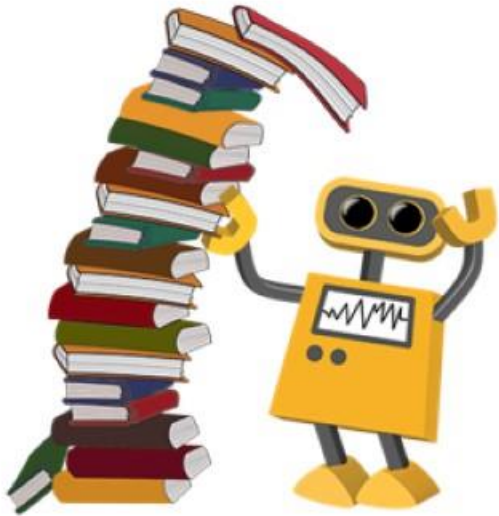
Example: If a robot helps a human clean a room and they finish in 30 minutes instead of an hour, that's good and means they work well together.



Doing the Right Job:

Explanation: Making sure the robot does its **part of the job correctly**, like a robot that sorts books by color without mistakes.

Example: A robot helps sort red and blue books into separate boxes. If it puts all the books in the right boxes without mixing them up, it's doing a great job.



Robots Working on Their Own:

Explanation: Seeing if the robot can do tasks by itself without needing a human to tell it what to do every time.

Example: A robot vacuum cleans the floors by itself every day without someone having to start it each time.



Changing Plans When Needed:

- Explanation:** The robot can change what it's doing based on new situations, like avoiding spilled juice on the floor it didn't know was there.
- Example:** If a robot is delivering mail indoors and finds a hallway blocked, it figures out a different way to go on its own.



Keeping the Robot Busy:

- **Explanation:** Making sure the robot has tasks to do so it's not just sitting around unused.
- **Example:** A robot in a store helps customers find items, clean up, and restock shelves all day instead of standing idle.



How Happy People Are with the Robot:

- Explanation:** After working with the robot, people feel good about it and think it's helpful.
- Example:** After using a robot assistant in class, students feel it's fun and helps them learn better.



Thinking Hard?:

- **Explanation:** Checking if using the robot makes people feel stressed or tired from thinking too much.
- **Example:** If a teacher uses a robot to grade tests and finds it easy to use without getting a headache, that's great.



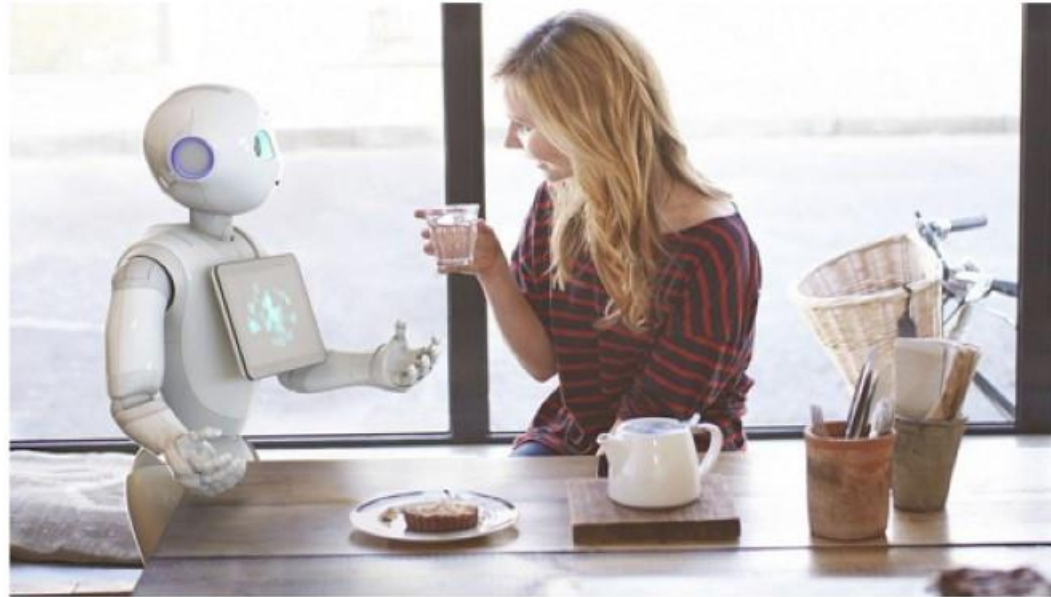
Trusting the Robot:

- **Explanation:** People believe the robot will do its job well and not make mistakes.
- **Example:** A family trusts their robot pet sitter to feed and play with their cat correctly when they are not home.



Learning to Use the Robot:

- **Explanation:** Finding out if it's easy and quick for someone to learn how to work with the robot.
- **Example:** If someone can start using a robot to make coffee after a quick lesson, that means it's easy to learn.

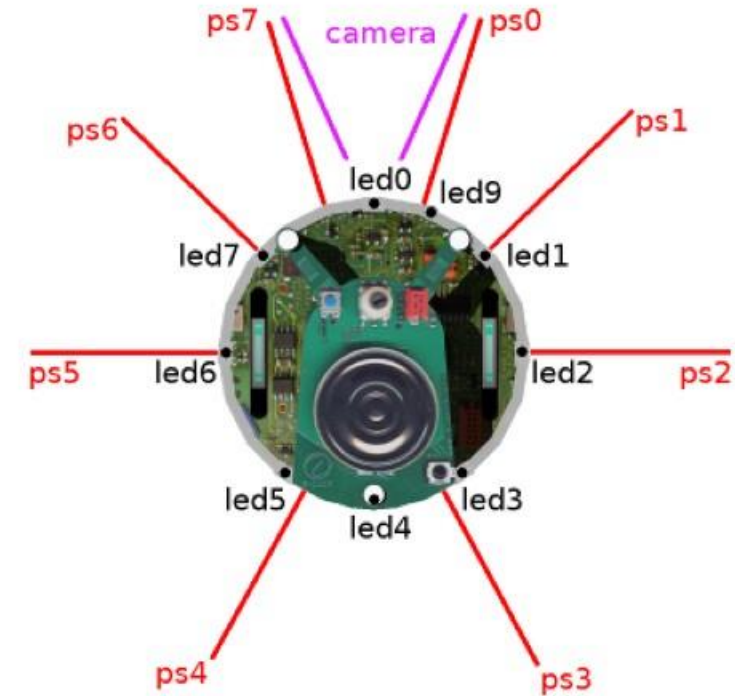
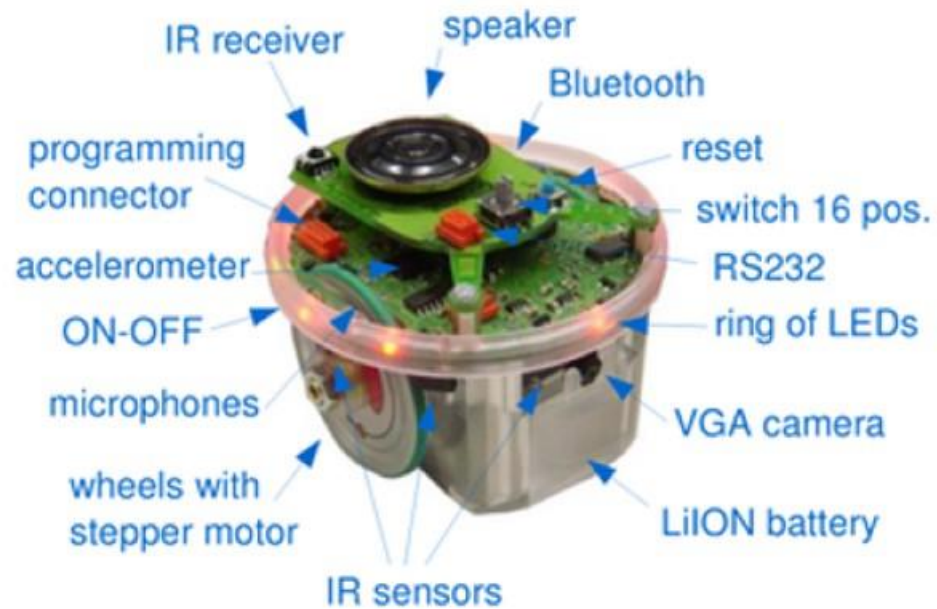


Making Human Jobs Easier:

- **Explanation:** The robot takes over tasks that are hard or boring, so people can do other important things.
- **Example:** A robot takes over cleaning the floors at a restaurant, so the staff can spend more time making meals and helping customers.





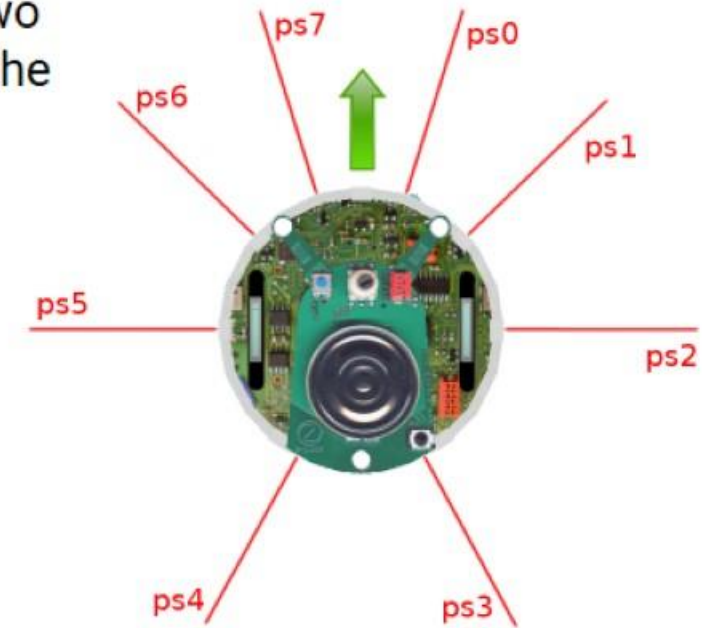


Objective: Expand obstacle detection to utilize multiple infrared sensors for more nuanced navigation.

Step 1: Controller Expansion for Multiple Sensors

Enhance the controller to read from multiple sensors and react accordingly.

Controller programming requires some information related to the e-puck model. In order to create the collision avoidance algorithm, we need to read the values of its 8 infra-red distance sensors located around its turret, and we need to actuate its two wheels. The way that the distance sensors are distributed around the turret and the e-puck direction are depicted in [this figure](#).



1. **Start:** Begin the robot control program.
2. **Initialize Robot and Motors:** Set up the robot, left and right motors, including setting positions to infinite and initial velocities to 0.
3. **Define Constants:** Specify the time step and forward speed.
4. **Initialize Sensors:** For each sensor ('ps0', 'ps1', 'ps2', 'ps7'), enable it with the defined time step and add it to the sensor objects list.
5. **Control Loop Begins:** Enter the main control loop that runs at each time step.
 - **Read Sensor Values:** Get distance readings from all initialized sensors.
 - **Obstacle Detection:** Check if there's an obstacle in front or to the sides based on sensor values.
 - If an obstacle is detected in front or front-right, **Turn Left**.
 - If an obstacle is detected in front-left, **Turn Right**.
 - If no obstacles are detected, **Move Forward**.
6. **Repeat Control Loop:** Continue the loop until the program ends.
7. **End:** The program concludes when manually stopped or when the robot's task is completed.

```
# Import necessary classes for the robot control
from controller import Robot, Motor, DistanceSensor
```

```
# Initialize the Robot instance
robot = Robot()
```

```
# Define the time step of the robot's control loop
TIME_STEP = 64
```

```
# Initialize motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set to infinite position mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
```

```
# Initial velocity
leftMotor.setVelocity(0.0)
rightMotor.setVelocity(0.0)
```

```
# Speed constants
FORWARD_SPEED = 5.0
```

```
# Initialize multiple distance sensors by their names
sensors = ['ps0', 'ps1', 'ps2', 'ps7']
sensorObjects = []
for sensorName in sensors:
```

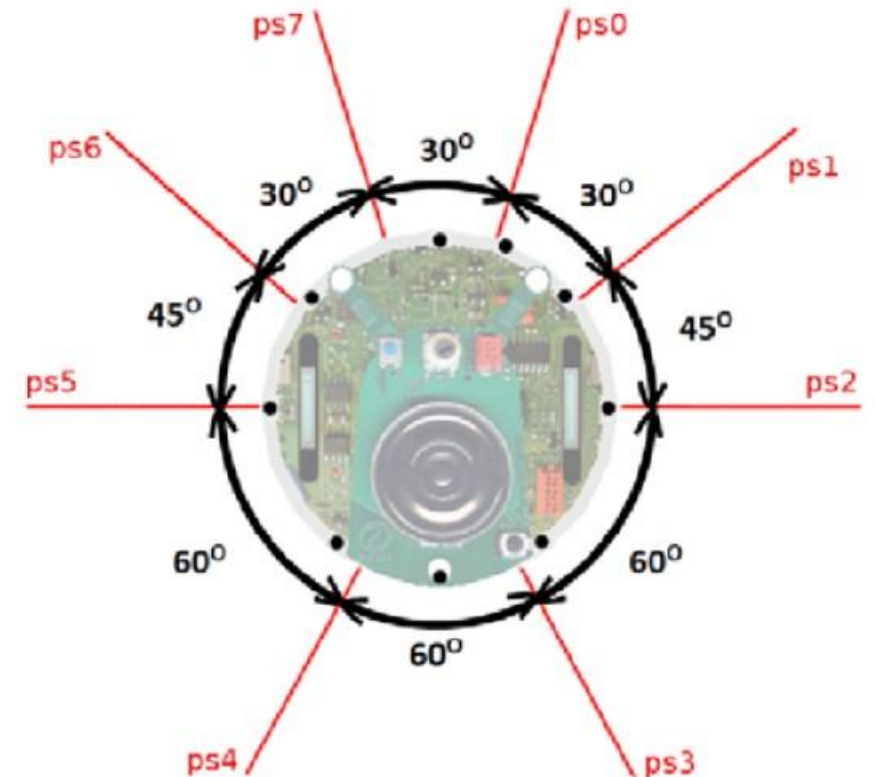
```
    sensor = robot.getDevice(sensorName) # Get the sensor device
    sensor.enable(TIME_STEP) # Enable the sensor with the defined time step
    sensorObjects.append(sensor) # Append the sensor object to the list
```

```
# Control loop: execute at each simulation step
while robot.step(TIME_STEP) != -1:
    # Read sensor values into a list
    sensorValues = [sensor.getValue() for sensor in
                    sensorObjects]
```

```
# Basic obstacle detection and navigation strategy
if sensorValues[0] < 70 or sensorValues[1] < 70:
    # Obstacle detected in front or front-right: turn left
    leftMotor.setVelocity(-FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
elif sensorValues[3] < 800:
    # Obstacle detected in front-left: turn right
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(-FORWARD_SPEED)
else:
    # No obstacle detected: move forward
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
```

Initializing multiple distance sensors on a robot

1. **Define Sensor Names:** Start by listing the names of the sensors you're going to use. For instance, ``sensors = ['ps0', 'ps1', 'ps2', 'ps7']`` includes four sensors located at different positions on the robot.
2. **Create a List for Sensor Objects:** Prepare a list to store the sensor objects. This is done with ``sensorObjects = []``.
3. **Loop Through Each Sensor Name:**
 - Use a ``for`` loop to iterate over each sensor name in the ``sensors`` list.
4. **Get the Sensor Device:**
 - Inside the loop, use ``robot.getDevice(sensorName)`` to get each sensor based on its name. This function connects your code to the specific hardware sensor on the robot.
5. **Enable the Sensor:**
 - Call ``sensor.enable(TIME_STEP)`` to activate the sensor, allowing it to start collecting data. The ``TIME_STEP`` specifies how often (in milliseconds) the sensor should update its data.
6. **Store Sensor Objects:**
 - Add the initialized and enabled sensor to the ``sensorObjects`` list with ``sensorObjects.append(sensor)``. This step is crucial for easily accessing each sensor's data later in your code.



```
sensorValues = [sensor.getValue() for sensor in sensorObjects]
```

1. **List Comprehension:** This line uses a Python feature called "list comprehension" to create a new list. It's a concise way to apply an operation (in this case, `sensor.getValue()`) to each item in an existing list (`sensorObjects`).
2. **`sensorObjects` List:** `sensorObjects` is a list that contains all the distance sensor objects you previously initialized. Each `sensor` in this list is an individual distance sensor object.
3. **`sensor.getValue()`:** For each `sensor` in the `sensorObjects` list, `.getValue()` is called. This method retrieves the current reading from the sensor, which is a numerical value representing the distance to the nearest object.
4. **Creating `sensorValues` List:** The numerical values retrieved by `sensor.getValue()` for each sensor are collected into a new list named `sensorValues`. This list now contains the current distance reading from each of the sensors in `sensorObjects`.
5. **Why This Matters:** By gathering sensor readings into `sensorValues`, you can easily check and react to the robot's surroundings. For example, you can use these values to decide if the robot should continue moving forward, turn left, or turn right to avoid obstacles.

This approach simplifies accessing and using sensor data, making your robot's control logic clearer and easier to manage, especially for tasks like obstacle avoidance.

The obstacle detection and navigation strategy uses sensor readings to control the robot's movement. When sensors 0 or 1 detect an obstacle (values below 800), indicating something in front or to the front-right, the robot turns left by setting a negative velocity for the left motor and a positive velocity for the right motor. If sensor 3 detects an obstacle, implying an obstruction is to the front-left, it turns right by reversing these velocities. If no obstacles are detected, both motors propel the robot forward at the same speed. This basic logic helps the robot navigate by avoiding collisions.

```
# Basic obstacle detection and navigation strategy
if sensorValues[0] < 800 or sensorValues[1] < 800:
    # Obstacle detected in front or front-right: turn left
    leftMotor.setVelocity(-FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
elif sensorValues[3] < 800:
    # Obstacle detected in front-left: turn right
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(-FORWARD_SPEED)
else:
    # No obstacle detected: move forward
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
```

```

# Import necessary classes for the robot control
from controller import Robot, Motor, DistanceSensor

# Initialize the Robot instance
robot = Robot()

# Define the time step of the robot's control loop
TIME_STEP = 64

# Initialize motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf')) # Set to infinite position mode
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0.0) # Initial velocity
rightMotor.setVelocity(0.0)

# Speed constants
FORWARD_SPEED = 5.0

# Initialize multiple distance sensors by their names
sensors = ['ps0', 'ps1', 'ps2', 'ps7']
sensorObjects = []
for sensorName in sensors:
    sensor = robot.getDevice(sensorName) # Get the sensor device
    sensor.enable(TIME_STEP) # Enable the sensor with the defined time step
    sensorObjects.append(sensor) # Append the sensor object to the list

```

```

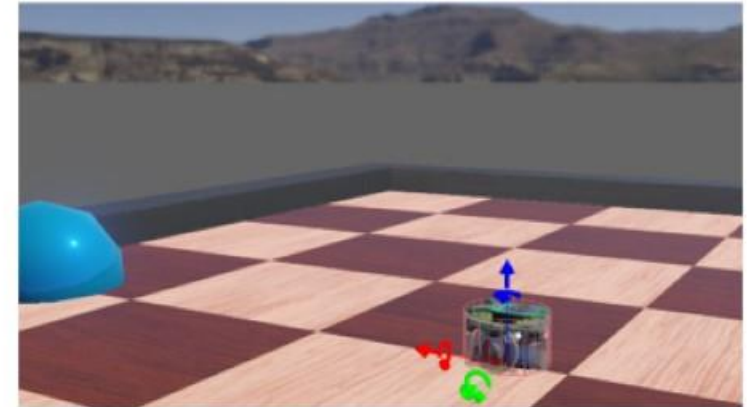
# Control loop: execute at each simulation step
while robot.step(TIME_STEP) != -1:
    # Read sensor values into a list
    sensorValues = [sensor.getValue() for sensor in sensorObjects]

    # Basic obstacle detection and navigation strategy
    if sensorValues[0] < 800 or sensorValues[1] < 800:
        # Obstacle detected in front or front-right: turn left
        leftMotor.setVelocity(-FORWARD_SPEED)
        rightMotor.setVelocity(FORWARD_SPEED)
    elif sensorValues[3] < 800:
        # Obstacle detected in front-left: turn right
        leftMotor.setVelocity(FORWARD_SPEED)
        rightMotor.setVelocity(-FORWARD_SPEED)
    else:
        # No obstacle detected: move forward
        leftMotor.setVelocity(FORWARD_SPEED)
        rightMotor.setVelocity(FORWARD_SPEED)

```

Robot that flashes its LED red and white three times before turning right when it detects an obstacle.

1. **Start:** Begin the robot program.
2. **Initialize the Robot:** Set up the robot's motors to an infinite position and velocity to 0.
3. **Define Time Step:** Set the simulation time step (64 ms in this case).
4. **Initialize Sensors:** Enable distance sensors (ps0 and ps7) for obstacle detection.
5. **Initialize LED:** Set up an LED for visual feedback.
6. **Enter Control Loop:** Start the main loop that runs every simulation step.
 - **Read Sensor Values:** Check the distance values from sensors ps0 and ps7.
 - **Detect Obstacle:** If both sensors detect an obstacle (values less than 70),
 - **Flash LED:** Turn the LED on and off three times to signal obstacle detection.
 - **Turn Right:** Adjust motor velocities to turn the robot right.
 - **Else,**
 - **Move Forward:** Set both motors to move the robot forward.
7. **Repeat Control Loop:** Continue the loop until the program ends.
8. **End:** Terminate the program.



```
from controller import Robot, LED, DistanceSensor
```

```
# Initialize the robot  
robot = Robot()
```

```
# Define the time step  
TIME_STEP = 64
```

```
# Initialize the LEDs on the robot  
led = robot.getDevice('led0') # Assuming the robot has an LED named  
'led0'
```

```
# Initialize the distance sensors  
sensor0 = robot.getDevice('ps0')  
sensor7 = robot.getDevice('ps7')  
sensor0.enable(TIME_STEP)  
sensor7.enable(TIME_STEP)
```

```
# Initialize motors, assuming the robot has motors named 'left wheel  
motor' and 'right wheel motor'  
leftMotor = robot.getDevice('left wheel motor')  
rightMotor = robot.getDevice('right wheel motor')  
leftMotor.setPosition(float('inf'))  
rightMotor.setPosition(float('inf'))  
leftMotor.setVelocity(0)  
rightMotor.setVelocity(0)
```

```
# Function to flash LED three times  
def flashLED(led):
```

```
    for _ in range(3):  
        led.set(1) # Turn LED red  
        robot.step(TIME_STEP)  
        led.set(0) # Turn LED off (or white if your robot supports it)  
        robot.step(TIME_STEP)
```

```
# Main control loop
```

```
while robot.step(TIME_STEP) != -1:
```

```
    # Read sensor values  
    value0 = sensor0.getValue()  
    value7 = sensor7.getValue()
```

```
    # Check if there is an obstacle in front  
    if value0 < 70 and value7 < 70:
```

```
        flashLED(led) # Flash LED if obstacle is detected
```

```
        # Turn right
```

```
        leftMotor.setVelocity(5) # Adjust velocity to turn the robot
```

```
        rightMotor.setVelocity(-5)
```

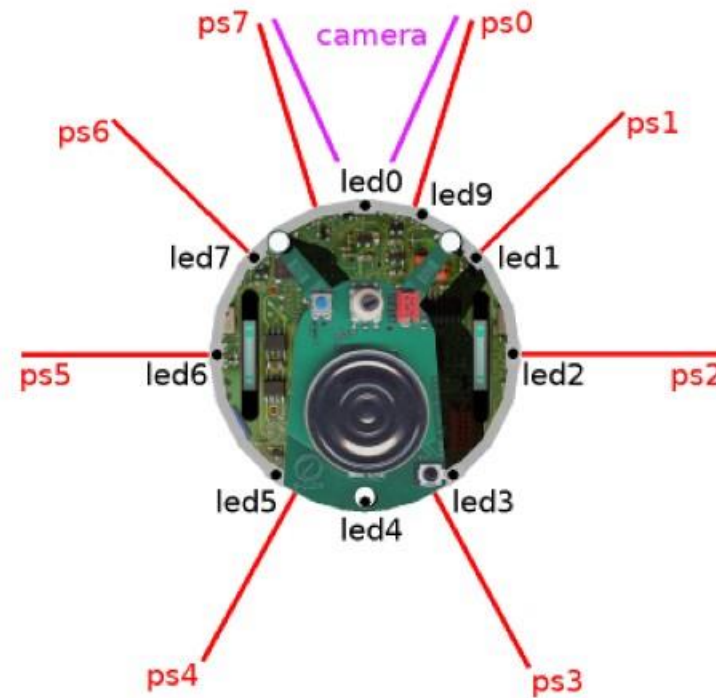
```
    else:
```

```
        # Move forward if no obstacle
```

```
        leftMotor.setVelocity(5)
```

```
        rightMotor.setVelocity(5)
```

E-puck robot can use its **camera** to capture an image and analyze the color of the central pixel in the Webots simulation environment.



```
# Import necessary modules from the Webots library.
# Robot is the main class for robot control.
# Camera is for accessing the robot's camera.
from controller import Robot, Camera

# Define the simulation time step in milliseconds.
# This is how often the simulation updates the robot's sensors and actuators.
TIME_STEP = 64

# Create an instance of the Robot class.
# This will be our main handle to interact with the robot.
robot = Robot()

# Get a handle to the camera device on the robot.
# 'camera' is the name of the camera device specified in the robot's model.
# If the camera has a different name in your project, change 'camera' accordingly.
camera = robot.getDevice('camera')

# Enable the camera to start capturing images.
# The camera will update its image every TIME_STEP milliseconds.
camera.enable(TIME_STEP)

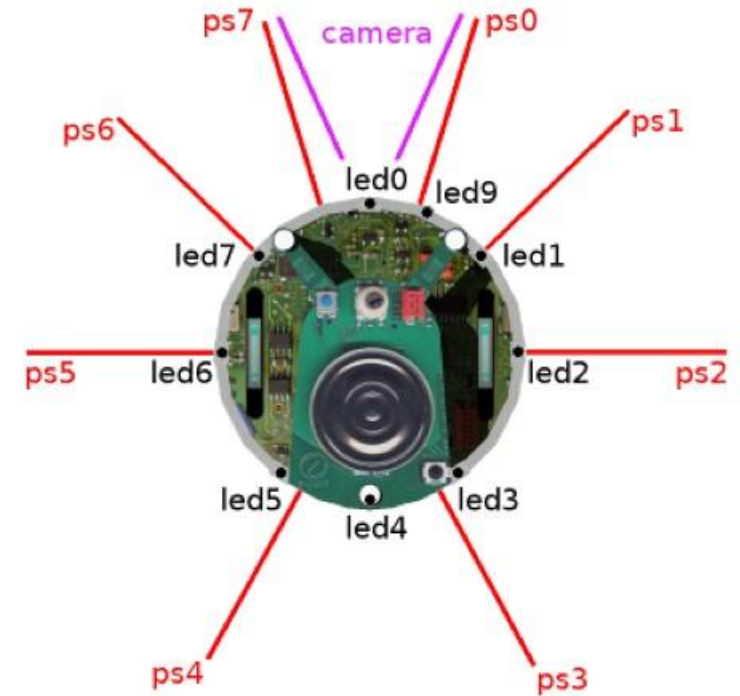
# Main loop: Execute actions repeatedly until the simulation is stopped.
while robot.step(TIME_STEP) != -1:
    # Capture an image from the camera.
    # The getImage() method returns the latest image captured by the camera.
    image = camera.getImage()
```

```
# Retrieve the width and height of the camera image.
# These values are set in the Webots world file and can be used to process the
image.
cameraWidth = camera.getWidth()
cameraHeight = camera.getHeight()
print(f'Captured an image of size: {cameraWidth}x{cameraHeight}')

# Calculate the position of the central pixel in the image.
# We divide the width and height by 2 to find the center.
centerX = cameraWidth // 2
centerY = cameraHeight // 2

# Get the RGB color values of the central pixel.
# imageGetRed, imageGetGreen, and imageGetBlue methods are used to
extract
# the color components of a pixel at a specified position.
red = camera.imageGetRed(image, cameraWidth, centerX, centerY)
green = camera.imageGetGreen(image, cameraWidth, centerX, centerY)
blue = camera.imageGetBlue(image, cameraWidth, centerX, centerY)

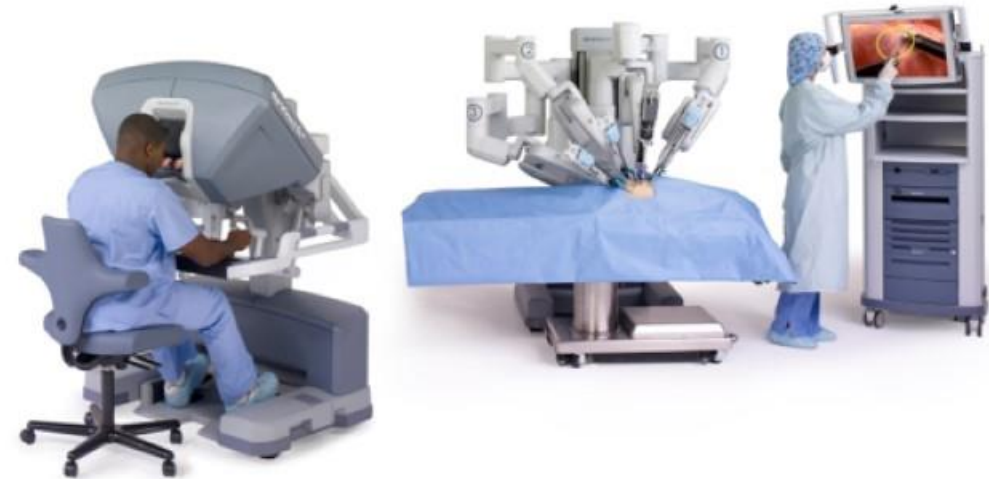
# Print the RGB color values of the central pixel.
# This can be useful for tasks like color detection or analysis.
print(f'Color of the central pixel: R={red}, G={green}, B={blue}')
```



"Innovative Designs and Real-World Applications of Robotics and AI"



1. Robotic Surgery: Surgical robots, like the da Vinci Surgical System, allow surgeons to perform complex procedures with more precision, flexibility, and control than traditional techniques. These robots offer minimally invasive options, reducing recovery time and improving patient outcomes.



2.Care Robots: Robots designed for care settings assist with tasks ranging from patient monitoring to delivering therapy, helping to alleviate the workload on human caregivers and enhancing patient care.



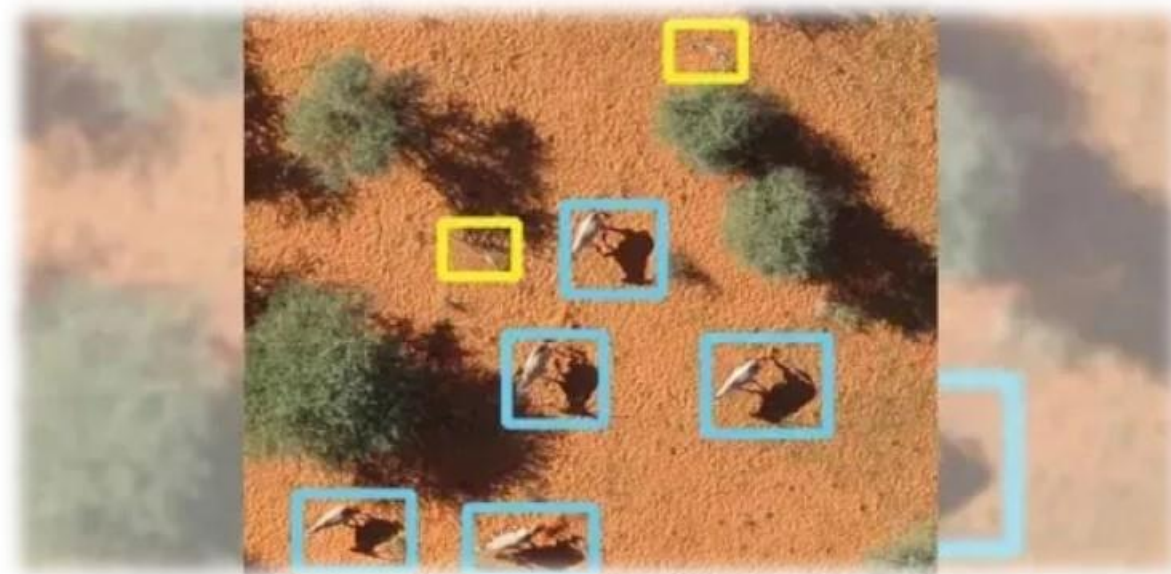
Agricultural Drones and Robots: These technologies revolutionize farming practices by providing detailed aerial views that help in crop monitoring, health assessment, and targeted pesticide spraying. Ground-based robots automate tasks like weeding, planting, and harvesting.



Precision Agriculture: AI-driven analysis of data collected from sensors and drones enables farmers to make informed decisions about water management, crop rotation, and optimal planting strategies, leading to increased crop yield and sustainability.



Wildlife Conservation: AI algorithms analyze images and data from camera traps to monitor wildlife, track poaching activities, and study biodiversity, contributing to conservation efforts.



Ocean Cleanup: Innovative robotic systems are deployed to collect plastic and other waste from oceans and rivers, addressing the critical issue of water pollution.



Educational Robots: Robots serve as teaching assistants or learning platforms, offering interactive and personalized learning experiences. They help in teaching coding, STEM subjects, and languages, making education more engaging.



Traffic Management: AI-driven traffic control systems analyze real-time data to optimize traffic flow, reduce congestion, and improve road safety.



Waste Management: Smart waste collection systems use sensors to monitor trash levels and optimize collection routes, improving efficiency and cleanliness.





Thank You

