SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)

COIMBATORE – 641035

DEPARTMENT OF MECHATRONICS ENGINEERING

Arithmetic Operations in Scilab

```
// Define variables

a = 10;

b = 5;

// Arithmetic Operations

addition = a + b;

subtraction = a - b;

multiplication = a * b;

division = a / b;

modulus = mod(a, b); // Modulus operation

// Display Results

disp("Addition: " + string(addition));

disp("Subtraction: " + string(subtraction));

disp("Multiplication: " + string(multiplication));

disp("Division: " + string(division));

disp("Modulus: " + string(modulus));
```

Scilab Program: Basic application

```
// Scilab program to perform basic calculator operations
```

```
// Function to display a menu and get the user's choice
function choice = displayMenu()
    disp("Choose an operation:");
    disp("1. Addition");
    disp("2. Subtraction");
    disp("3. Multiplication");
    disp("4. Division");
    disp("5. Exit");
    choice = input("Enter your choice (1-5): ");
endfunction

// Main Program
while %t  // Infinite loop, will break when user chooses Exit
    choice = displayMenu();  // Display menu and get choice

    // Validate choice
    if choice == 5 then
        disp("Exiting Calculator. Goodbye!");
        break;
    elseif choice < 1 | choice > 5 then
        disp("Invalid choice! Please enter a number between 1 and 5.");
        continue;
    end
```

```
    // Get user input
    num1 = input("Enter the first number: ");
    num2 = input("Enter the second number: ");

    // Perform operations based on user's choice
    select choice
        case 1
            result = num1 + num2;
            disp("Addition: " + string(result));
        case 2
            result = num1 - num2;
            disp("Subtraction: " + string(result));
        case 3
            result = num1 * num2;
            disp("Multiplication: " + string(result));
        case 4
            if num2 ~= 0 then
                result = num1 / num2;
                disp("Division: " + string(result));
            else
                disp("Error: Division by zero is undefined.");
            end
    end
end
```

Explanation:

1. **Menu Function (displayMenu):**
    - Displays a menu of operations and takes the user's choice.
    - Returns the selected operation as an integer.
2. **Infinite Loop (while %t):**
    - Ensures the calculator runs continuously until the user chooses to exit.
3. **User Input:**
    - Prompts the user to input two numbers for arithmetic operations.
4. **Arithmetic Operations:**
    - Addition, subtraction, multiplication, and division are implemented using a select block.
    - Division includes a check to avoid dividing by zero.
5. **Exit Option:**
    - The program exits when the user selects option 5.

Sample Output:

Example 1: Addition

Choose an operation:

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your choice (1-5): 1

Enter the first number: 5

Enter the second number: 10

Addition: 15

Example 2: Division by Zero

Choose an operation:

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your choice (1-5): 4

Enter the first number: 8

Enter the second number: 0

Error: Division by zero is undefined.

Example 3: Exit

Choose an operation:

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your choice (1-5): 5

Exiting Calculator. Goodbye!

Scilab Program: DC Motor Control in Electric Vehicles

This program models the control of a DC motor used in electric vehicles (EVs) to simulate both acceleration and regenerative braking.

Program Code:

```
// Define DC Motor Parameters
R = 1;         // Armature resistance (ohms)
L = 0.5;       // Armature inductance (H)
Ke = 0.01;     // Back EMF constant (V/rad/s)
Kt = 0.01;     // Torque constant (Nm/A)
J = 0.01;      // Moment of inertia (kg.m^2)
B = 0.1;       // Friction coefficient (Nm.s/rad)

// Transfer Function: Motor Dynamics
s = poly(0, 's');  // Laplace variable
motor_tf = Kt / (J*L*s^2 + (J*R + B*L)*s + (B*R + Kt*Ke));  // DC motor model

// PID Controller for Acceleration
Kp_acc = 50;  // Proportional gain for acceleration
Ki_acc = 20;  // Integral gain for acceleration
Kd_acc = 5;   // Derivative gain for acceleration

pid_acc = Kp_acc + Ki_acc/s + Kd_acc*s;  // PID controller for acceleration
open_loop_acc = motor_tf * pid_acc;      // Open-loop transfer function
closed_loop_acc = feedback(open_loop_acc, 1);  // Closed-loop system
```

```
// PID Controller for Regenerative Braking
Kp_brk = 40;  // Proportional gain for braking
Ki_brk = 10;  // Integral gain for braking
Kd_brk = 5;   // Derivative gain for braking

pid_brk = Kp_brk + Ki_brk/s + Kd_brk*s;  // PID controller for braking
open_loop_brk = motor_tf * pid_brk;      // Open-loop transfer function
closed_loop_brk = feedback(open_loop_brk, 1);  // Closed-loop system

// Time vector for simulation
t = 0:0.01:5;  // Simulation time (0 to 5 seconds)

// Simulate Acceleration
[y_acc, x_acc] = csim('step', t, closed_loop_acc);  // Step response for acceleration

// Simulate Regenerative Braking
[y_brk, x_brk] = csim('impulse', t, closed_loop_brk);  // Impulse response for braking

// Plot Acceleration Response
subplot(2, 1, 1);
plot(t, y_acc);
xlabel('Time (s)');
ylabel('Speed (rad/s)');
title('DC Motor Control: Acceleration Response');
```

```
// Plot Regenerative Braking Response
subplot(2, 1, 2);
plot(t, y_brk);
xlabel('Time (s)');
ylabel('Speed (rad/s)');
title('DC Motor Control: Regenerative Braking Response');
```

Explanation of the Code:

1. **DC Motor Dynamics:**
    - The motor's transfer function is modeled based on its electrical and mechanical properties.
2. **Acceleration Control:**
    - A PID controller for acceleration ensures the motor achieves the desired speed quickly and efficiently.
    - Gains (Kp,Ki,KdK_p, K_i, K_d) are tuned to achieve minimal overshoot and settling time.
3. **Regenerative Braking:**
    - Another PID controller is used to simulate regenerative braking by controlling the deceleration of the motor.
    - An impulse input simulates the braking scenario.
4. **Simulation:**
    - csim('step') is used to simulate acceleration.
    - csim('impulse') is used to simulate the response for braking.
5. **Visualization:**
    - Separate plots for acceleration and braking responses allow easy analysis.

Real-World Relevance:

1. **Acceleration:**

- During acceleration, the controller ensures smooth and efficient speed buildup without sudden jerks, critical for passenger comfort and safety.

2. **Regenerative Braking:**
   - During braking, the motor acts as a generator to recover kinetic energy, storing it as electrical energy in the battery. The controller ensures smooth deceleration to avoid skidding or abrupt stops.

Applications in Electric Vehicles:

1. **Improved Efficiency:**
   - Regenerative braking reduces energy loss, improving overall vehicle efficiency.
2. **Enhanced Comfort:**
   - Smooth acceleration and deceleration improve ride quality.
3. **Battery Management:**
   - Efficient motor control helps optimize battery usage, extending the vehicle's range.

Output and Observation:

1. **Acceleration Response:**
   - The step response graph shows how the motor achieves the desired speed quickly with minimal overshoot and a short settling time.
2. **Regenerative Braking Response:**
   - The impulse response graph demonstrates controlled deceleration, recovering energy without abrupt speed drops.

Conclusion:

This program demonstrates the use of Scilab to simulate DC motor control in electric vehicles, highlighting its importance in achieving efficient acceleration and energy recovery during braking. This simulation provides insights into optimizing motor performance for real-world EV applications.

Application Question (16 Marks):

"Write a Scilab program to simulate object detection for a surveillance and security robot and explain its real-world applications."

Solution:

Overview:

Surveillance and security robots require object detection capabilities to identify threats, monitor areas, and react to anomalies. This simulation models a robot in a grid-like environment that detects objects (representing potential intrusions) within its range.

Scilab Program: Object Detection for Surveillance Robot

```
// Simulation Parameters
robot_pos = [5, 5];        // Robot's position (x, y)
object_pos = [3, 4; 8, 9; 6, 3]; // Intruder/object positions in the grid
detection_range = 4;       // Detection range of the robot (units)

// Create Environment Grid
env_size = 10;             // Environment grid size (10x10)
env = zeros(env_size, env_size); // Empty grid

// Mark Robot's Position in the Grid
env(robot_pos(1), robot_pos(2)) = 2; // 2 for the robot

// Mark Object/Intruder Positions in the Grid
for i = 1:size(object_pos, 'r')
    env(object_pos(i, 1), object_pos(i, 2)) = 1; // 1 for objects
```

```
end

// Display Initial Environment Grid
disp("Environment Grid (0: Empty, 1: Object, 2: Robot):");
disp(env);

// Object Detection Logic
detected_objects = [];  // List of detected objects
for i = 1:size(object_pos, 'r')
    distance = sqrt((object_pos(i, 1) - robot_pos(1))^2 + (object_pos(i, 2) - robot_pos(2))^2);
    if distance <= detection_range then
        detected_objects = [detected_objects; object_pos(i, :)]; // Add to detection list
    end
end

// Display Detected Objects
disp("Objects Detected Within Range:");
disp(detected_objects);

// Visualization of Environment
clf;
xgrid(); // Activate grid for plotting
plot2d(robot_pos(2), env_size - robot_pos(1), -5, "ko", " ", " ", "o"); // Robot position
for i = 1:size(object_pos, 'r')
    plot2d(object_pos(i, 2), env_size - object_pos(i, 1), -5, "r*", " ", " ", "*"); // Object positions
```

```
end

// Highlight Detected Objects
for i = 1:size(detected_objects, 'r')
    plot2d(detected_objects(i, 2), env_size - detected_objects(i, 1), -5, "g+", " ", " ", "+"); // Detected objects
end

// Annotate and Label Plot
title("Surveillance Robot: Object Detection Simulation");
xlabel("X-axis (Grid Position)");
ylabel("Y-axis (Grid Position)");
legend(["Robot", "Objects", "Detected Objects"]);
```

Explanation of the Code:

1. **Environment Grid:**
    - A 10x10 grid represents the robot's surveillance area.
    - The robot's position is marked, and potential intruders/objects are added at specified coordinates.
2. **Detection Logic:**
    - The program calculates the Euclidean distance between the robot and each object.
    - Objects within the detection range are identified as detected intrusions.
3. **Visualization:**
    - The robot's position, objects, and detected objects are plotted on a 2D grid for easy analysis.
    - Detected objects are highlighted with distinct markers.

Real-World Applications:

1. **Perimeter Monitoring:**
    - Robots patrol a specified area and detect intrusions within their sensor range.
2. **Intrusion Detection:**
    - Identifies unauthorized access or unusual activity in sensitive locations like banks, warehouses, or military zones.
3. **Public Safety:**
    - Surveillance robots in crowded areas can monitor for unattended objects or potential threats.
4. **Home Security:**
    - Domestic security robots detect movements or objects in restricted zones.

Simulation Results:

1. **Grid Display:**
    - The robot's position and object locations are displayed in the environment.
2. **Detection Report:**
    - The program outputs the coordinates of detected objects within the robot’s range.
3. **Visual Plot:**
    - A graphical representation shows the robot, objects, and detected objects, aiding in understanding the simulation.

Control Structures in Scilab

Introduction:

Control structures such as **if-else**, **for loops**, **while loops**, **break**, and **continue** are essential for programming robots to make decisions, perform actions, and navigate in a grid-like environment. These control structures allow the robot to make real-time decisions based on its current position, obstacles, and target location. In Scilab, we can simulate the robot's movement in a grid and implement decision-making logic using these control structures.

Control Structures and Their Roles:

1. **if-else Statements:**
    - The if-else control structure is used to make decisions based on certain conditions. For instance, the robot can check if it has encountered an obstacle, if it has reached the target, or if it needs to turn.
    - **Example:** If the robot is facing an obstacle, it must decide to either turn or reverse.
2. **for Loops:**
    - The for loop is used when the robot needs to iterate through a predefined number of steps or actions. It can be used to simulate the robot's movement over a set number of grid positions or check certain grid cells in a fixed sequence.
    - **Example:** A for loop can simulate the robot checking the grid's next position for obstacles or performing repeated movements.
3. **while Loops:**
    - The while loop is used when the robot's task depends on dynamic conditions, such as until it reaches a specific location or encounters an obstacle.
    - **Example:** The robot can continue moving until it detects the target or an obstacle, making it more flexible for real-time operations.
4. **break Statement:**
    - The break statement is used to exit a loop prematurely when a specific condition is met.

In this case, when the robot reaches its target or encounters a critical obstacle, it can break out of the movement loop.

- **Example:** If the robot reaches its target, the loop is terminated using break.

5. **continue Statement:**
    - The continue statement is used to skip the current iteration of a loop and continue to the next one. This is useful when the robot detects an obstacle in its path and needs to skip moving to that cell, moving instead to the next available cell.
    - **Example:** If the robot encounters an obstacle, it will skip that movement and continue checking the next position.

Scenario: Robot Moving in a Grid-Like Environment

Problem:

Consider a robot placed in a 5x5 grid. The robot must move from its starting position (0,0) to a target location (4,4). The robot can only move horizontally or vertically, and certain grid cells contain obstacles. The robot must decide its movement based on the state of the grid, avoiding obstacles and reaching the target.

Steps to Implement Using Control Structures:

1. **Initialize Grid:**
    - The grid is represented as a matrix where 0 indicates an empty space and 1 represents an obstacle.

Example:

grid = [

0, 0, 0, 0, 0;

0, 1, 0, 1, 0;

0, 0, 0, 0, 0;

1, 0, 1, 0, 0;

```
    0, 0, 0, 0, 0
];
```

2. **Decision-Making (Using if-else):**
    - At each step, the robot checks the grid cell where it intends to move. If there is an obstacle, it makes a decision to move in a different direction or stop.

Example:

```
if grid[new_row, new_col] == 1 then
    disp("Obstacle detected. Changing direction.");
    // Take action (e.g., turn or reverse)
else
    robot_position = [new_row, new_col];  // Move to the new position
end
```

3. **Movement Simulation (Using for loop):**
    - The robot can attempt to move step-by-step in a grid using a for loop, where the number of steps is fixed or dynamic.

Example:

```
for i = 1:10  // Simulate 10 steps
    if robot_position == target then
        disp("Target reached!");
        break;  // Exit loop when the target is reached
    end
    // Decision-making and movement logic here
end
```

4. **Dynamic Movement (Using while loop):**
    - The robot can keep moving in real-time until it reaches its target. The condition to check whether the robot is at the target location or needs to avoid an obstacle would be

dynamically evaluated in the while loop.

Example:

```
while robot_position != target do
   if grid[robot_position(1), robot_position(2)] == 1 then
      disp("Obstacle detected. Changing path.");
      // Adjust path or avoid obstacle
      continue;  // Skip the current iteration and check next move
   else
      // Move the robot
      robot_position = [robot_position(1)+1, robot_position(2)];  // Move right
   end
end
```

5. **Breaking Out of Loops (Using break):**
    - If the robot reaches its target, the loop is exited prematurely using the break statement.

Example:

```
if robot_position == target then
   disp("Target reached! Stopping.");
   break;  // Exit loop if target is reached
end
```

6. **Skipping Iterations (Using continue):**
    - When an obstacle is encountered, the continue statement is used to skip the current iteration and check the next position.

Example:

```
if grid[robot_position(1), robot_position(2)] == 1 then
   disp("Obstacle detected. Skipping this position.");
   continue;  // Skip to the next iteration
```

```
end
```

Scilab Code for Robot Simulation in a Grid:

```
// Define the grid (0: empty, 1: obstacle)
grid = [
    0, 0, 0, 0, 0;
    0, 1, 0, 1, 0;
    0, 0, 0, 0, 0;
    1, 0, 1, 0, 0;
    0, 0, 0, 0, 0
];

// Define starting and target positions
robot_position = [1, 1];  // Start position
target = [5, 5];  // Target position

// Simulate robot movement
while robot_position != target do
    if robot_position == target then
        disp("Target reached!");
        break;  // Exit loop when the target is reached
    end

    // Check for obstacles at the next position (right move for simplicity)
    next_position = robot_position + [0, 1];  // Moving right
```

```
    if grid(next_position(1), next_position(2)) == 1 then
        disp("Obstacle detected. Changing direction.");
        robot_position = robot_position + [1, 0];  // Move down instead
        continue;  // Skip current move and check next one
    else
        robot_position = next_position;  // Move to the next position
    end
end
disp("Robot reached the target at: " + string(robot_position));
```

Analysis of Control Structures:

1. **if-else:**
    - **Role:** Helps in decision-making by checking conditions such as obstacles or target location.
    - **Example:** If the robot encounters an obstacle, it chooses an alternate path or stops.
2. **for loop:**
    - **Role:** Useful when the number of steps or iterations is predefined. In this case, it could simulate the robot taking fixed steps until reaching a destination or stopping condition.
    - **Example:** The robot moves step-by-step until the target is reached.
3. **while loop:**
    - **Role:** Keeps the robot moving until the target is reached, making it more flexible and adaptive.
    - **Example:** The robot continuously checks if it has reached the target or if an obstacle is present, and adjusts accordingly.
4. **break:**
    - **Role:** Exits the loop once the target is reached, preventing unnecessary further checks.

- **Example:** Stops the robot from moving once it reaches its destination.

5. **continue:**
    - **Role:** Skips the current iteration when an obstacle is detected and checks the next position.
    - **Example:** Allows the robot to avoid obstacles without stopping the loop entirely.

Applications of Control Structures in Robotics:

1. **Autonomous Navigation:**
    - Robots in autonomous vehicles or drones use these control structures to avoid obstacles and reach destinations in dynamic environments.
2. **Warehouse Automation:**
    - Robots navigating in grid-like environments with fixed paths use these structures to manage their movements efficiently.
3. **Simulations:**
    - Control structures are used in simulations to mimic real-world robot behavior, allowing for virtual testing before physical