1 PID Controller Lambda Binding Design Pattern

1.1 Functional Programming Design Pattern

The PID controller implementation employs functional programming design patterns through lambda expressions for sensor reading and output conversion functions, achieving complete decoupling between control algorithms and hardware interfaces.

Algorithm 1 Hardware Interface Binding through Function Pointers

Require: PID controller instance, sensor interface, actuator interface
Ensure: Configured PID controller with bound I/O functions

```
Define sensor reading function:
read_sensor ← λ() → double
    return INA226_getCurrent_mA()

Define output conversion function:
convert_output ← λ(output: double) → double
    MCP4725_setVoltage(output)
    return output

Bind functions to PID controller:
controller.read_sensor ← read_sensor
controller.convert_output ← convert_output
```

1.2 Architecture Advantages

This functional programming approach provides several advantages:

- **Hardware abstraction**: Control logic is independent of specific sensor/actuator implementations
- **Enhanced testability**: Mock functions can be easily substituted for unit testing
- **Improved maintainability**: Hardware changes require only function binding modifications
- **Runtime flexibility**: I/O functions can be dynamically reconfigured during operation

1.3 Implementation Example

Algorithm 2 Lambda Function Implementation for Sensor Interface

Require: Sensor hardware interface, data processing requirements
Ensure: Bound lambda functions for PID controller

Current sensor reading lambda:
current_reader $\leftarrow \lambda()\{$
  $raw\_value \leftarrow$ INA226.getCurrent_raw()
  $calibrated\_value \leftarrow raw\_value \times cal\_factor + offset$
  return $calibrated\_value$
$\}$

Voltage output lambda:
voltage_writer $\leftarrow \lambda(output\_voltage)\{$
  $dac\_value \leftarrow (output\_voltage/V_{ref}) \times 4095$
  MCP4725.setVoltage(dac_value)
  return $output\_voltage$
$\}$

Controller binding:
pid_controller.bind_input($current\_reader$)
pid_controller.bind_output($voltage\_writer$)

1.4 Design Pattern Benefits

1.4.1 Decoupling and Modularity

The lambda binding pattern creates a clear separation between:

- Control algorithm logic (PID computation)
- Hardware interface details (sensor/actuator communication)
- Data processing operations (calibration, scaling)

1.4.2 Testing and Validation

Mock functions can be easily substituted for hardware interfaces:

Algorithm 3 Mock Function Binding for Testing

Require: Test scenarios, simulation parameters
Ensure: Testable PID controller configuration

Mock sensor function:
mock_sensor $\leftarrow \lambda()\{$
    return $test\_value + noise\_generator()$
$\}$

Mock actuator function:
mock_output $\leftarrow \lambda(value)\{$
    $test\_log.append(value)$
    return $value$
$\}$

Test configuration:
pid_controller.bind_input($mock\_sensor$)
pid_controller.bind_output($mock\_output$)