Tampere University of Applied Sciences

Using Lua in Building Automation: Architecture, Benefits, and Comparison with Other Scripting Languages

Yajun Guo

BACHELOR'S THESIS
May 2025

Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Software Engineering

GUO, YAJUN:
Using Lua in Building Automation: Architecture, Benefits, and Comparison with Other Scripting Languages

Bachelor's thesis 44 pages, appendices 15 pages
May 2025

Modern building-automation systems (BAS) must juggle heating, ventilation, air-conditioning (HVAC) and wider building-management (BMS) tasks on controllers that have only a few megabytes of memory and tight real-time deadlines. Adding a lightweight scripting layer to the embedded-Linux firmware of a programmable-logic controller (PLC) is one way to keep these devices adaptable without constant recompilation. This thesis examines Lua, whose interpreter occupies roughly 120–250 kB, and asks whether its small footprint and hot-reload capability make it the most practical choice for field-level logic at Bithouse Oy.

The study begins with a technical review of Lua's runtime and C-API, then documents how Bithouse Oy uses Lua scripts to switch temperature modes, apply heating curves, and expose remote-procedure calls. A representative heating-control programme—comprising a hysteresis thermostat and a piece-wise-linear output curve—is implemented in Lua, CPython, MicroPython and JavaScript/Node.js. Each version is exercised for 100 simulated control cycles in a PC test-bed configured to mimic PLC constraints. Logged metrics show that Lua completes the loop in ≈ 0.019 ms while consuming ≈ 2.6 MB RSS, CPython is faster (≈ 0.015 ms) but ten times larger, MicroPython is smallest (≈ 40 kB RAM; ≈ 0.04 ms) yet sacrifices some speed, and Node.js is quickest (≈ 0.010 ms) but heaviest (≈ 44 MB RSS).

The results confirm Lua as the best trade-off for DIN-rail PLCs: fast enough for 100 ms control windows, reloadable without downtime, and small enough to coexist with logging and networking tasks. MicroPython approaches Lua's footprint and suits ultra-small sensor nodes, whereas CPython and Node.js are better reserved for edge computers and dashboards. The thesis closes with practical guidelines for mixing these languages across the BAS stack and outlines future work on LuaJIT trials, IoT integration and richer remote-control schemes—advancing both Bithouse Oy's client projects and the wider smart-building field.

Keywords: lua, building automation, hvac, scripting

CONTENTS

ABBREVIATIONS AND TERMS

BAS	Building Automation System
BMS	Building Management System
C-API	C Application Programming Interface
Hysteresis	A control technique preventing rapid switching in systems by defining a buffer around setpoints
HVAC	Heating, Ventilation and Air-Conditioning
IoT	Internet of Things, a network of connected devices (e.g., smart sensors) enabling data exchange and automation in BAS
PLC	Programmable Logic Controller, a ruggedized device executing control logic for BAS, such as HVAC and BMS tasks
RPC	Remote Procedure Call, a protocol allowing remote execution of BMS functions
JIT	Just-In-Time compilation
Table	Lua’s primary data structure, functioning as arrays or objects
Metatable	Lua’s feature for customizing table behaviour, enabling object-oriented programming for flexible PLC logic

1 INTRODUCTION

1.1 Background and Motivation

Building-automation systems are growing in both size and ambition. Today a single controller may juggle heating, ventilation, air-conditioning (HVAC), and the wider building-management stack (BMS). To stay responsive, these controllers need a control language that bends to new requirements without slowing the plant. Many vendors meet that need by adding a lightweight scripting layer on top of the PLC firmware that already runs on embedded-Linux boards.

Lua has become a popular choice for that layer. It loads into just 120–250 kB of RAM yet runs fast enough for real-time tasks. Engineers also like its clean C API, which makes it simple to wire Lua scripts to field buses, databases, or web dashboards. Even with its small core, Lua offers useful extras—metatables, coroutines, and basic object-orientation—so teams rarely feel boxed in (Ierusalimschy, 2003). At Bithouse Oy, for instance, Lua scripts adjust HVAC setpoints, expose remote-procedure calls, and let developers roll out last-minute features without touching the compiled PLC code.

Academic research on scripting in building automation often emphasizes languages like Python and JavaScript, valued for their extensive libraries and integration with IoT-enabled smart buildings (Heidary et al., 2023). However, this focus overlooks Lua's potential for embedded systems, where its ~120–250 kB runtime and metaprogramming features, such as metatables, enable efficient control logic for PLC-based HVAC and BMS tasks at Bithouse Oy. This thesis examines Lua's industrial benefits through a practical heating-control example, bridging academic trends with embedded automation needs.

1.2 Scope and Structure of the Thesis

This thesis explores Lua's role in building automation, focusing on its use in programmable logic controllers (PLCs) for systems like heating, ventilation, and air conditioning (HVAC) and Building Management Systems (BMS). It looks at real-

world projects, especially those at Bithouse Oy, where Lua helps set up control rules and add flexible features to Linux-based PLCs. The study also compares Lua to Python and JavaScript to see how it stacks up in speed, memory use, and ease of coding for automation tasks. The goal is to understand Lua's benefits and limits in these systems, using practical examples and simple tests.

- **Chapter 2** sketches Lua's design goals, its PLC integration path, and a quick side-by-side with Python and JavaScript.
- **Chapter 3** presents case studies, starting with Bithouse Oy deployments and moving to other public examples.
- **Chapter 4** implements a compact heating-curve controller in Lua, Python, and JavaScript so the three can be compared on equal footing.
- **Chapter 5** measures runtime, memory use, and coding friction, then pulls the numbers into easy tables.
- **Chapter 6** reflects on when Lua is a clear win, when it may not fit, and offers hands-on tips for field engineers.

References plus appendices—source code, build scripts, and raw logs—close out the document.

2 THEORETICAL BACKGROUND

2.1 Lua Design Principles and Runtime Characteristics

Lua, a lightweight scripting language launched in 1993 at the Pontifical Catholic University of Rio de Janeiro, aims to add flexible scripting to C-based programs without a heavy runtime (Ierusalimschy, 2003). Its name, meaning “moon” in Portuguese, reflects its sleek design.

Lua stays small. The full interpreter plus core libraries fit in roughly 120–250 kB. That footprint is perfect for the low-power PLC boards common in building-automation panels. Because the code is pure ANSI C, cross-compiling for new CPUs or tool-chains is rarely a headache—exactly the situation at Bithouse Oy, where controllers range from ARM-based sticks to x86 boxes.

Lua is built for embed. Designed to work inside host programs, Lua’s C API lets developers link scripts to hardware or services with minimal code. This allows field updates to control logic, like adjusting thermostat settings at Bithouse Oy, without rebuilding firmware—a key advantage for HVAC and BMS systems.

Lua is simple and flexible. Lua’s small standard library relies on three powerful features:

- Tables - A versatile structure for arrays, dictionaries, or objects.
- Metatables - Tables that customize operations, enabling flexible coding similar to object-oriented styles.
- Coroutines - Lightweight threads for managing tasks, such as automating state machines.

Dynamic typing simplifies quick changes, perfect for on-site tweaks.

Why this matters for building automation? HVAC and BMS controllers need frequent updates, like new schedules or set-points. Lua’s small size, fast startup, and embeddability let Bithouse Oy build lean PLCs that stay responsive and adapt quickly to changing building needs.

Lua compiles source to bytecode, then runs it on a tight virtual machine that needs few CPU cycles per instruction. The incremental garbage collector frees memory in short bursts, keeping 50 Hz control loops smooth. In third-party benchmarks Lua often beats Python and Node.js both in memory use and in round-trip latency (Marinescu & Sutter, n.d.). Lua itself continues to evolve; recent work documents how features such as environments and global-variable handling were streamlined after version 5.3, improving both safety and execution speed (Ierusalimschy et al., 2025).

2.2 Lua in PLCs and Building Automation Contexts

Programmable Logic Controllers (PLCs) are central to building automation, controlling HVAC systems, lights, and Building Management Systems (BMS). Traditional PLC programming, like ladder logic, works well but struggles with frequent tweaks or custom rules. Lua, a lightweight scripting language, addresses this by acting as a flexible extension layer in modern PLCs, especially those running embedded Linux.

Lua's Role in PLCs: Lua complements fast core tasks handled by firmware, offering

- **Control Logic Tweaks:** Lua scripts define schedules or conditions, like "if temperature rises, open vents," easier to edit than C or ladder logic.
- **Configuration Scripts:** Lua files store site-specific settings, such as temperature limits, reusable across projects.
- **Remote Interaction:** Scripts enable network access (e.g., RPC/REST) for apps to check sensors or adjust settings.
- **Hot Reloads:** New scripts load without rebooting, saving time.

At Bithouse Oy, Lua adjusts vent settings based on occupancy sensors, allowing quick updates over SSH without disrupting building operations.

Benefits of Lua in applications: Lua's scripts separate custom logic from firmware, speeding up setup. Engineers edit Lua files to meet new energy rules or user needs, reducing downtime. Though academic studies on Lua in automation

are scarce (Marinescu & Sutter, n.d.), its use in PLCs shows it meets the demand for flexible, network-aware control in smart buildings.

2.3 Language Comparisons in Automation

Scripting languages like Lua, Python, and JavaScript add flexibility to building automation systems, such as PLCs in HVAC or Building Management Systems (BMS). Their differences in size, speed, and real-time performance affect their suitability for embedded controllers.

Lua's interpreter (~120–250 kB) fits resource-constrained PLCs, like those at Bithouse Oy for thermostat control, ensuring fast startup and low RAM usage for real-time HVAC tasks (Ierusalimschy, 2003). MicroPython, with a ~256 kB runtime, provides a comparable footprint for embedded systems but requires additional configuration, limiting its integration ease compared to Lua (MicroPython, 2023). Lua's C-API links sensors to scripts efficiently, though its minimal library necessitates custom code for complex features, unlike MicroPython's streamlined Python syntax for rapid prototyping.

Python's extensive libraries support dashboards and cloud services in building automation, but its ~20–30 MB runtime is too large for most PLCs, with slow startup hindering real-time tasks like reading sensors every 200 ms (Dhambarage, 2022). To close this gap, stripped-down interpreters have appeared. MicroPython 1.24 fits into roughly 256 kB of flash and 16 kB of RAM, bringing Python's footprint close to Lua's while keeping the core language intact (MicroPython, 2025). The port is already commercialised—Casio's FX-9860 GIII scientific calculator ships with a built-in MicroPython mode for classroom scripting (Casio Education, 2025). On MCU boards such as the ESP32, MicroPython can drive HVAC-style control loops, although benchmark studies report slightly longer cycle times than Lua or compiled C (Plauska et al., 2023). For that reason, Lua still embeds more directly in PLC firmware, while Python (either full CPython or MicroPython) remains stronger in higher-level analytics or when a project standardises on Python end-to-end.

JavaScript, via Node.js, excels in REST APIs for BMS dashboards, with an event-driven model for network tasks. However, its 50+ MB runtime and unpredictable JIT compilation make it unsuitable for PLCs, favoring UI or cloud roles.

Table 1 below summarizes several comparisons of various aspects of the different languages.

Table 1. Comparison table of the features of Lua, Python, MicroPython and JavaScript.

Feature	Lua (5.X/LuaJIT)	Python (CPython)	Python (MicroPython)	JavaScript (V8/Node.js)
Binary Size	Tiny (≤250 kB)	Large (≥20 MB)	Tiny (≈ 0.25 MB)	Very Large (≥50 MB)
Typical RAM Usage	Low	High	Very low	Moderate-high
Startup Latency	Fast	Slow	Fast	Moderate-Slow
C-API Embeddability	Simple	Complex	MCU-level C hooks	Engine-Dependent
Real-Time Friendliness	Good (Soft RT[1])	Poor	Good (Soft RT[1] with uasyncio)	Unpredictable (JIT)
Standard Library Depth	Minimal	Rich	Minimal subset	Moderate
Common PLC Usage	Frequent	Rare	Occasional (sensor nodes)	Rare

[1]Soft RT: Suitable for non-critical timing tasks.

Binary-size and RAM figures for MicroPython are taken from the official documentation (MicroPython docs, 2025).

3 CASE STUDY: LUA IN BUILDING AUTOMATION PROJECTS

3.1 Overview of Real-World Use Cases in Building Automation

Modern HVAC and Building-Management Systems (BMS) must adapt to each building's occupancy profile, energy targets, and local codes. Because Lua's interpreter is small—about 120 – 250 kB—and scripts can be reloaded without a reboot, many vendors use Lua to customise PLC firmware on site.

Bithouse Oy offers a tailor-made solution for each customer, without using standard configurations, to meet different project requirements. PLC firmware written in C/C++ runs the core tasks, while Lua scripts loaded at startup or updated in real time manage customer-specific logic, for example:

- **Room-Temperature Modes:** Scripts switch between comfort, eco, and night settings, using logic to prevent rapid switching (hysteresis).
- **Time Schedules:** Scripts define daily on/off times in text files, refreshed nightly.
- **Alarms and Fallbacks:** Lua monitors sensors for issues like high temperatures, sending network alerts and setting safe states.
- **Custom Curves:** Heating-curve functions map inputs such as outdoor temperature to valve positions that match each client's HVAC design.

All scripts live in version control, so engineers can adjust thresholds or rules from the office and ship a hot-patch instead of travelling on site.

Lua also supports non-commercial projects:

- **eLua:** Runs on microcontrollers, controlling DIY HVAC with sensor and relay scripts (Marinescu & Sutter, n.d.).
- **NodeMCU:** Uses Lua on ESP8266/ESP32 for BMS tasks like lighting, linked to dashboards via MQTT.

The following Table 2 summarizes the common patterns and the advantages of Lua.

Table 2. Lua's advantages in key tasks.

Task Type	Lua Advantage
Zone Climate Control	Clear, short syntax for set-point logic
Time-Based Scheduling	Tables handle programs concisely
Threshold Alarms	Config files allow site-specific limits
Custom Scaling	Tables support client-specific mappings

3.2 Lua for Control Logic Configuration and Extension

In building automation systems, flexibility and customization are often more valuable than fixed control strategies. Different buildings require different behaviours: some prioritize energy savings, others comfort, and many need to comply with location-specific regulations. In building automation, Lua scripts provide flexible control logic for PLCs, allowing Bithouse Oy to tailor HVAC and BMS settings to each client's unique needs.

One typical use case is thermostat-like control logic — the system must determine whether heating or cooling should be active, based on input values like temperature or humidity. These controls often include hysteresis, threshold comparison, and multiple modes such as high-limit, low-limit, or range-based activation. A good example of this is Bithouse Oy's `hvac_limitControl` function from the `hvacex` library, which encapsulates a configurable, reusable control block.

Picture 1 is the simplified version of the relevant Lua code from Bithouse Oy. The `hvac_block()` function acts as a general-purpose control unit:

- It retrieves configuration and sensor values using `Data.get()`.
- It applies logic only if the control is enabled.
- It defers the actual threshold logic to `hvac_controlBlock()`.

The control behaviour is fully defined by the data structure passed in, which allows engineers to reuse this code across many devices and deployments. By

simply modifying the configuration data (e.g., limit values, control mode, hysteresis), the same Lua logic can apply different rules in different projects.

```lua
function hvac_limitControl(pointList)
    for i, id in ipairs(pointList) do
        hvac_block(id)
    end
end

function hvac_block(id)
    local data = Data.get(id)

    -- Read current state
    local enabled = tonumber(Data.get(data.enabledId .. ".pv"))
    local input   = tonumber(Data.get(data.inputId .. ".pv"))

    if enabled ~= nil then
        data.enabled = enabled
        Data.set(id .. ".enabled", enabled)
    end

    if input ~= nil then
        data.input = input
        Data.set(id .. ".input", input)
    end

    -- If disabled, output 0; else run control block
    if (data.enabled or 0) < 1 then
        Data.set(id .. ".pv", 0)
    else
        local output = hvac_controlBlock(data)
        Data.set(id .. ".pv", output)
    end

    -- Forward output
    Data.set(data.outputId .. ".pv", data.pv)
end
```

Picture 1. Bithouse Oy's hvac_limitControl Function

Picture 2 below, adapted from Bithouse Oy, shows a simplified control block for client-specific logic, using `Data.get/set` to read/write PLC sensor data.

```
function hvac_controlBlock(data)
    if data.comparisonType == "lowLimit" then
        if data.input < data.controlLowLimit then
            return 1  -- Activate heating
        elseif data.input >= (data.controlLowLimit + data.hys-
teresis) then
            return 0  -- Deactivate heating
        end
    elseif data.comparisonType == "highLimit" then
        if data.input > data.controlHighLimit then
            return 1  -- Activate cooling
        elseif data.input <= (data.controlHighLimit - data.hys-
teresis) then
            return 0  -- Deactivate cooling
        end
    elseif data.comparisonType == "between" then
        if data.input > data.controlLowLimit and data.input <
data.controlHighLimit then
            return 1  -- Keep system on
        elseif data.input <= (data.controlLowLimit - data.hys-
teresis) or
               data.input >= (data.controlHighLimit + data.hys-
teresis) then
            return 0  -- Turn system off
        end
    end
end
```

Picture 2. A simplified control block

This structure supports a wide range of behaviours through just a few control modes:

- Low-limit control: e.g., turns heating on if temperature is too low
- High-limit control: e.g., activate cooling if temperature is too high
- Between-mode control: e.g., keep system on only within a target range

Because all thresholds and hysteresis values are defined in project-specific configuration files, this function doesn't need to change between deployments. Only the data changes — the Lua script remains stable and reusable.

In practice, this type of configuration-driven scripting helps Bithouse Oy quickly adapt to new buildings without modifying core firmware. Lua enables rapid iteration, on-site changes, and live updates, which are difficult to achieve with compiled control logic alone. The full Lua script is in Appendix 1.

3.3 Metatables, Object-Oriented Techniques and Flexibility

In building automation, flexibility is crucial: every project has its own requirements, rules, and equipment setups. Lua meets these needs with a powerful feature called metatables, which allow scripts to mimic object-oriented behavior without heavy boilerplate code. At Bithouse Oy, engineers use metatables to make PLC control logic adaptable, reusable, and easy to customize for different client scenarios.

Metatables in Lua are special tables that change how other tables behave—essentially letting you define custom logic for operations like accessing data. With metatables, Lua tables can behave like objects, each having their own properties and methods. For instance, a sensor or actuator point can be represented as an object, storing values (like temperatures or thresholds) and providing methods (such as reading current states or checking alarm conditions).

This Picture 3 is a practical example adapted from Bithouse Oy's real-world automation scripts. The following Lua function executes client-specific logic for PLC points, using metatables to allow scripts to access point data directly:

```lua
-- Run client-specific script for a PLC point
function slc_runDBScript(id, point)
    local script = point.script or ""  -- Client-defined Lua script
    if #script < 3 then script = "" end  -- Check if script is valid

    local f, compile_err = loadstring(script)
    if not f then return -2, compile_err end  -- Compile-time error handling

    point.id = id

    -- Set point data as globally accessible within the script
    setmetatable(_G, { __index = point })

    local success, runtime_err = pcall(f)

    -- Clear global environment modification after execution
    setmetatable(_G, nil)

    if not success then return -1, runtime_err end  -- Runtime error handling
    return 1, runtime_err  -- Execution succeeded
end
```

Picture 3. Example metatable

In this function, `setmetatable(_G, {__index = point})` makes all properties of the point table (assumptions e.g. temperature or setpoint) accessible directly within the client-specific script. This enables simpler, more intuitive scripting like Picture 4.

```lua
if temperature > setpoint then
    pv = 1
else
    pv = 0
end
```

Picture 4. Example usage

In Picture 4, `temperature` and `setpoint` come directly from the point's data table. The variable `pv` stores the result of the control logic—typically a numeric or binary value indicating the desired output state. After the script runs, the automation system reads `pv` and applies this result directly to the corresponding hardware or process control points.

Using metatables and these scripting techniques brings multiple advantages to building automation:

- **Logic separated from data:** Engineers can add or modify logic without touching core firmware code.
- **Per-point customization:** Each sensor or actuator can behave uniquely by running small, tailored scripts.
- **Safe script execution:** The `pcall()` function safely catches errors, preventing a script problem from affecting system stability.
- **Low-overhead flexibility:** Rather than maintaining multiple compiled versions of control logic, engineers define specialized behaviour with small scripts tied directly to the data they manage.

Lua doesn't have built-in class-based object-oriented programming like Java or C++, yet metatables let engineers reach the same goal. By treating each data point as an object with its own state and behaviour, metatables simplify complex control logic. This makes the automation system lightweight, flexible, and easy to customize for each project's specific needs.

In short, metatables let Bithouse Oy's automation platform quickly adapt to new client requirements—whether that means dynamically handling sensor inputs, customizing alarms, or controlling building actuators—all without lengthy firmware changes or downtime.

4 IMPLEMENTATION: HEATING CONTROL IN LUA, PYTHON AND JAVASCRIPT

4.1 Task Description and Requirements

This chapter presents a practical heating control scenario implemented in Lua, Python, and JavaScript to evaluate Lua's strengths compared to other scripting languages. The example simulates a typical building automation task, representative of common PLC-based controls used at Bithouse Oy, combining thermostat-like decision logic with a scaling function to adjust heating output based on outdoor temperature.

4.1.1 Control Logic Overview

The control task is split into two components:

1. Thermostat Control (Decision Layer)

A basic thermostat rule decides if heating should be active or not, based on three parameters:

- Current indoor temperature (input)
- Desired setpoint (setpoint)
- Hysteresis value to prevent rapid on/off switching

This logic uses a simple yet effective structure:

- If temperature < (setpoint − hysteresis), heating turns ON.
- If temperature > (setpoint + hysteresis), heating turns OFF.
- Otherwise, the system maintains its current state.

This behaviour mirrors the "room-temperature modes”, and low-limit logic described previously in Chapter 3.

2. Heating Curve Logic (Scaling Layer)

Once the heating is activated, a curve function adjusts the heating output according to outdoor temperature. This reflects realistic HVAC demand curves commonly applied at Bithouse Oy:

- Reducing heating output when outside temperatures are higher.
- Increasing heating output when temperatures drop.

The curve is defined by points mapping outdoor temperature (X) to a corresponding heating output percentage (Y), typically ranging from 0% (no heating) to 100% (full heating capacity). A practical example might scale the output from 0% at 22°C outdoors to 100% at 16°C.

4.1.2 Implementation Requirements

To fairly evaluate Lua, Python, and JavaScript implementations, the following constraints are applied:

- All scripts use identical temperature inputs and configuration parameters.
- The task avoids system-specific calls or external APIs.
- Scripts simulate running in loops to reflect dynamic input conditions.
- The environment is PC-based but mimics embedded constraints such as limited memory availability.

Performance comparisons (detailed later in Chapter 5) will measure:

- **Execution time:** how quickly each script processes control logic.
- **Memory usage:** resource efficiency on constrained hardware.
- **Code simplicity and readability:** ease of use and maintenance by automation engineers.

This scenario was carefully selected to realistically reflect both decision-making and data-driven scaling tasks seen at Bithouse Oy. Sections 4.2, 4.3, and 4.4 will detail implementations in Lua, Python, and JavaScript, showing the practical strengths and trade-offs of each language.

4.2 Lua Implementation

Section 4.1 defined a heating control program for building automation, combining thermostat-like decision logic with curve-based output scaling, simulating a PLC-based task at Bithouse Oy. This section implements the program in Lua, leveraging its compact size and fast execution to manage client-specific HVAC settings. The code, adapted from Bithouse Oy's scripting, is presented in four parts, forming a cohesive program.

This Picture 5 segment sets client-specific parameters and mock inputs to simulate a BMS reacting to temperature changes, defining the thermostat's setpoint and curve-based output scaling.

```
-- Config parameters
local setpoint = 21.0
local hysteresis = 0.5


-- Define a temperature-to-output curve (X = outdoor temp, Y =
output level)
local heatingCurve = {
    points = {
        { pointX = 22, pointY = 0.0 }, -- No heating needed
        { pointX = 20, pointY = 0.3 },
        { pointX = 18, pointY = 0.6 },
        { pointX = 16, pointY = 1.0 } -- Full heating
    }
}
```

Picture 5. Lua Configuration

As shown in Picture 6, this function, inspired by Bithouse Oy's `hvac_curve`, calculates heating output (0-100%) using linear interpolation, mapping outdoor temperature to a client-specific curve.

```
function getHeatingOutput(curve, outdoorTemp)
    table.sort(curve.points, function(a, b) return a.pointX < b.pointX end)

    if outdoorTemp <= curve.points[1].pointX then
        return curve.points[1].pointY
    elseif outdoorTemp >= curve.points[#curve.points].pointX then
        return curve.points[#curve.points].pointY
    end

    for i = 1, #curve.points - 1 do
        local p1, p2 = curve.points[i], curve.points[i + 1]
        if outdoorTemp >= p1.pointX and outdoorTemp <= p2.pointX then
            local slope = (p2.pointY - p1.pointY) / (p2.pointX - p1.pointX)
            return (outdoorTemp - p1.pointX) * slope + p1.pointY
        end
    end

    return 0.0
end
```

Picture 6. Lua Curve Function

As seen in Picture 7, this function based on Bithouse Oy's `hvac_limitControl`, implements hysteresis-based thermostat logic, deciding if heating should be on or off, maintaining stability.

```
function thermostatControl(indoorTemp, setpoint, hysteresis, prevState)
    if indoorTemp < (setpoint - hysteresis) then
        return 1          -- Heating ON
    elseif indoorTemp > (setpoint + hysteresis) then
        return 0          -- Heating OFF
    else
        return prevState -- Maintain current state
    end
end
```

Picture 7. Lua Thermostat Function

This Picture 8 loop runs the control logic over temperature readings, producing results like Table 3 below, with full output in Appendix 2.

```
-- Simulation inputs
local indoorTemps = { 20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0,
20.6 }
local outdoorTemps = { 21, 20, 15, 18, 17, 16, 15, 14 }

local state = 0 -- Initial heating state: OFF

print("Indoor | Outdoor | Output | Heating")
print("-------------------------------------")

for i = 1, #indoorTemps do
    local indoor = indoorTemps[i]
    local outdoor = outdoorTemps[i]
    local output = getHeatingOutput(heatingCurve, outdoor)
    local heating = thermostatControl(indoor, setpoint, hyste-
resis, state)
    state = heating

    print(string.format(" %5.1f |   %5.1f  |  %4.1f  |   %s",
        indoor, outdoor, output * 100, (heating == 1 and "ON"
or "OFF")))
end
```

Picture 8. Lua Main Loop

Table 3. Excerpt of Lua results

Indoor	Outdoor	Output %	Heating
20.1	21.0	15	ON
19.8	15	100	ON
20.2	18	60	ON
21.7	16	100	OFF

4.3 Python Implementation

This section implements the heating control task described earlier, now using Python. It replicates the same thermostat decision logic and curve-based output scaling previously demonstrated in the Lua implementation (Section 4.2). The Python script maintains identical inputs, configuration values, and control logic structure.

The goal is to evaluate Python's readability, data structure management, and suitability for lightweight automation scripting. By directly mirroring Lua's logic, this Python implementation facilitates clear comparisons in clarity, expressiveness, and performance, as discussed further in Chapter 5.
The code is presented in four parts, forming a cohesive program.

This Picture 9 segment sets client-specific parameters and mock inputs to simulate a BMS reacting to temperature changes, defining the thermostat's setpoint and curve.

```
# Config parameters
setpoint = 21.0
hysteresis = 0.5
# Define curve: maps outdoor temperature (X) to heating output
(Y)
heating_curve = [
    {"pointX": 22, "pointY": 0.0},
    {"pointX": 20, "pointY": 0.3},
    {"pointX": 18, "pointY": 0.6},
    {"pointX": 16, "pointY": 1.0}
]
```

Picture 9. Python configuration

This Picture 10 calculates heating output using linear interpolation, mapping outdoor temperature to a client-specific curve.

```
def get_heating_output(curve, outdoor_temp):
    # Sort the curve points
    curve = sorted(curve, key=lambda p: p["pointX"])
    if outdoor_temp <= curve[0]["pointX"]:
        return curve[0]["pointY"]
    if outdoor_temp >= curve[-1]["pointX"]:
        return curve[-1]["pointY"]
    for i in range(len(curve) - 1):
        p1, p2 = curve[i], curve[i + 1]
        if p1["pointX"] <= outdoor_temp <= p2["pointX"]:
            slope = (p2["pointY"] - p1["pointY"]) /
(p2["pointX"] - p1["pointX"])
            return (outdoor_temp - p1["pointX"]) * slope +
p1["pointY"]
    return 0.0
```

Picture 10. Python curve function

This Picture 11 function implements hysteresis-based thermostat logic, deciding if heating should be on or off, maintaining stability for client-specific settings.

```
def thermostat_control(indoor_temp, setpoint, hysteresis,
prev_state):
    if indoor_temp < setpoint - hysteresis:
        return 1  # Turn heating ON
    elif indoor_temp > setpoint + hysteresis:
        return 0  # Turn heating OFF
    return prev_state  # Keep previous state
```

Picture 11. Python Thermostat function

This Picture 12 loop runs the control logic over temperature readings, producing results like Table 4 below, with full output in Appendix 2.

```
# Simulated temperature readings
indoor_temps = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6]
outdoor_temps = [21, 20, 15, 18, 17, 16, 15, 14]
state = 0  # Initial state: OFF

print("Indoor | Outdoor | Output | Heating")
print("-------------------------------------")
for i in range(len(indoor_temps)):
    indoor = indoor_temps[i]
    outdoor = outdoor_temps[i]
    output = get_heating_output(heating_curve, outdoor)
    heating = thermostat_control(indoor, setpoint, hysteresis,
state)
    state = heating
    print(f" {indoor:5.1f} |   {outdoor:5.1f}  |  {output *
100:4.1f}  |   {'ON' if heating == 1 else 'OFF'}")
```

Picture 12. Python main loop

Table 4. Excerpt of Python results

Indoor	Outdoor	Output	Heating
20.1	21.0	15	ON
19.8	15	100	ON
21.7	16	100	OFF

MicroPython variant

A trimmed version of the Python script (Appendix 2) runs in the MicroPython 1.24 Unix build. On this interpreter the 100-loop benchmark finishes in ≈ 4 ms and `gc.mem_alloc()` reports ≈ 33 kB heap use. Flash size for the interpreter binary is about 256 kB (MicroPython docs, 2025). Although measured on an emulated MCU, these figures confirm the same control logic fits comfortably on sensor-class hardware.

4.4 JavaScript Implementation

This section presents the JavaScript implementation of the heating control task. The logic is identical to the Lua and Python versions, combining threshold-based

thermostat control with curve-based scaling to simulate a realistic heating system in building automation.

Although JavaScript is often associated with web development, its syntax and flexibility also allow it to be used for control simulation and lightweight automation, especially in platforms like Node.js or browser-based dashboards.

This Picture 13 segment sets client-specific parameters and mock inputs to simulate a BMS reacting to temperature changes, defining the thermostat's setpoint and curve.

```
// Configuration parameters
const setpoint = 21.0;
const hysteresis = 0.5;
// Heating curve: maps outdoor temp (X) to output level (Y)
const heatingCurve = [
    { pointX: 22, pointY: 0.0 },
    { pointX: 20, pointY: 0.3 },
    { pointX: 18, pointY: 0.6 },
    { pointX: 16, pointY: 1.0 }
];
```

Picture 13. JavaScript configuration

Following Picture 14 and Picture 15 calculating heating output and implement the thermostat logic to do the same function as Lua and Python code snippets.

```
function getHeatingOutput(curve, outdoorTemp) {
    const sortedCurve = curve.sort((a, b) => a.pointX -
b.pointX);
    if (outdoorTemp <= sortedCurve[0].pointX) {
        return sortedCurve[0].pointY;
    } else if (outdoorTemp >= sortedCurve[sortedCurve.length -
1].pointX) {
        return sortedCurve[sortedCurve.length - 1].pointY;
    }
    for (let i = 0; i < sortedCurve.length - 1; i++) {
        const p1 = sortedCurve[i];
        const p2 = sortedCurve[i + 1];
        if (p1.pointX <= outdoorTemp && outdoorTemp <=
p2.pointX) {
            let slope = (p2.pointY - p1.pointY) / (p2.pointX -
p1.pointX);
            return (outdoorTemp - p1.pointX) * slope +
p1.pointY;
        }
    }
    return 0.0;
}
```

Picture 14. JavaScript curve function

```
function thermostatControl(indoorTemp, setpoint, hysteresis,
prevState) {
    if (indoorTemp < setpoint - hysteresis) {
        return 1;  // ON
    } else if (indoorTemp > setpoint + hysteresis) {
        return 0;  // OFF
    } else {
        return prevState;
    }
}
```

Picture 15. JavaScript thermostat function

This Picture 16 loop runs the control logic over temperature readings, producing results like Table 5 below, with full output in Appendix 2:

```
 console.log("Indoor | Outdoor | Output | Heating");
console.log("-----------------------------------");
for (let i = 0; i < indoorTemps.length; i++) {
    const indoor = indoorTemps[i];
    const outdoor = outdoorTemps[i];
    const output = getHeatingOutput(heatingCurve, outdoor);
    const heating = thermostatControl(indoor, setpoint, hyste-
resis, state);
    state = heating;
    console.log(` ${indoor.toFixed(1).padStart(5)} |   ${out-
door.toFixed(1).padStart(5)}  |  ${(output *
100).toFixed(1).padStart(4)}  |   ${heating === 1 ? 'ON' :
'OFF'}`);
}
```

Picture 16. JavaScript main loop

Table 5. Partial output results of the JavaScript code

Indoor	Outdoor	Output	Heating
20.0	20.0	30	ON
21.3	17	80	ON
20.6	14	100	OFF

5 EVALUATION AND COMPARISON

Chapter 4 implemented a heating control program in Lua, Python, and JavaScript, simulating a PLC-based task at Bithouse Oy. This chapter evaluates these implementations based on development experience, performance (execution time, memory usage, simplicity/readability), and technical trade-offs, comparing their suitability for embedded automation systems, with results informing Bithouse Oy's client-specific applications.

5.1 Development Experience and Ease of Use

Developing the heating control program in Lua, Python, and JavaScript revealed distinct experiences shaped by syntax, debugging, and integration with Bithouse Oy's workflows. Lua's lightweight, table-based syntax (Section 2.1) enabled rapid scripting for thermostat logic and curve scaling (Section 4.2), supporting quick iterations for client-specific updates, such as a retail client's BMS thermostat settings. Debugging relied on print statements, sufficient for small PLC scripts but limited by minimal tool support, requiring manual validation.

Python's indentation-based syntax keeps the code clear, and clear function definitions, such as `get_heating_output`, are easy to maintain. Built-in tools like `pdb` streamline debugging and boost productivity. However, deploying the full Python runtime on a PLC adds setup overhead, so the language fits better on edge computers or dashboards. The extra test utilities—`psutil` and `tracemalloc`—introduce only minor, temporary dependencies.

JavaScript's flexible syntax (Section 4.4) via Node.js supported prototyping, with array operations mirroring web-based BMS dashboards. Its event-driven model increased complexity for PLC tasks due to variable scoping. Debugging with console.log and Node.js tools was effective, but the web-focused runtime misaligned with embedded needs.

Lua provided the easiest development for Bithouse Oy's PLCs due to its simplicity and minimal setup, despite fewer debugging tools. Python and JavaScript, while

user-friendly for broader applications, introduced overhead less suited for embedded systems.

Table 6. Comparison of development experience factors for Lua, Python, and JavaScript.

Aspect	Lua	Python	JavaScript
Syntax Simplicity	Very compact and minimal	Clear and readable	Flexible but moderately complex
Learning Curve	Moderate (specific Lua concepts)	Low (intuitive syntax)	Low (familiar syntax)
Debugging Tools	Basic (print, simple IDEs)	Strong (IDEs, pdb)	Very strong (VSCode, DevTools)
Memory visibility	High (manual control via GC)	Moderate (profiling tools)	Lower (abstracted memory model)
Library/Framework Dependence	None	Minimal (psutil, tracemalloc)	Node.js runtime required
Embedded Friendliness	Excellent (lightweight)	Acceptable (moderate overhead)	Poor (heavy runtime)

5.2 Performance

To fairly compare the runtime performance of Lua, Python, and JavaScript in building automation control tasks, small benchmarking extensions were added to each language's implementation.

These extensions included:

- Measuring execution time for 100 iterations
- Capturing memory usage, both process-wide (RSS) and internal heap usage where possible

The core heating control logic presented in Chapter 4 remained unchanged. Only timing and memory measurement code was appended without altering the functional behaviour.

Full measurement scripts and additional test logs are available in Appendix 3.

The following Picture 17, Picture 18, and Picture 19 are sample outputs for Lua, Python and JavaScript languages. MicroPython runs on the web simulator as described in Appendix 3.

```
Execution time (100 iterations): 1.862 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
```

Picture 17. Example Lua output

```
Execution time (100 iterations): 1.527 ms
Memory usage (RSS, kB): 12672.00
Memory usage (tracemalloc heap, kB): 44.41
```

Picture 18. Example Python output

```
Execution time (100 iterations): 0.951 ms
Memory usage (RSS, kB): 43520.00
Memory usage (heap, kB): 5323.92
```

Picture 19. Example JavaScript output

5.2.1 Execution Time Results

Each script was executed five times (MicroPython was run manually in the web simulator), and the typical execution times are summarized in Table 7:

Table 7. The results of execution time per language

Language	Execution Time (100 iterations)
Lua	~1.86 ms
CPython	~1.53 ms
MicroPython	≈ 4 ms
JavaScript	~0.97 ms

JavaScript (Node.js) completed the simulation fastest, followed by Python. Lua, while still very efficient, showed slightly longer execution times in this specific

benchmark. MicroPython's ≈ 4 ms per-100-loop time is slower than Lua but still well under the 100 ms control window and, thanks to a 33 kB heap, delivers the smallest memory footprint of all four interpreters (Table 7).
The ranking looks counterintuitive since Lua is widely regarded as both compact and fast. The gap is explained by the way each runtime handles the repeated sort operation embedded in the benchmark:

- **Python**'s `sorted()` is a highly optimized C function, tightly integrated into the interpreter.
- **JavaScript**'s `sort()` in Node.js (V8 engine) is JIT-compiled into native machine code, benefiting from runtime optimization.
- **Lua**'s `table.sort()` is written in C as well, but it is invoked repeatedly from the interpreted Lua layer. Without Just-In-Time (JIT) compilation like LuaJIT, these calls can accumulate small delays.

In this benchmark, the heating curve was sorted during every output calculation. While this was consistent across all languages for fairness, the JIT-enabled environments (JavaScript and Python) were able to optimize the looped sorting calls more efficiently than interpreted Lua.

In a practical BMS deployment, the curve is ordinarily sorted once at start-up, not on every control loop. With that optimisation in place, Lua would perform significantly faster, potentially outperforming Python in embedded contexts.
Despite this specific overhead, Lua's sub-2ms execution time still qualifies it as highly suitable for real-time control scenarios in building automation.

5.2.2 Memory Usage Results

Memory usage was measured by monitoring the process Resident Set Size (RSS) and, which includes the total real memory consumption, as well as internal heap memory where applicable.

Table 8. Memory usage results per language

Language	RSS Memory Usage (kB)	Internal Heap Memory (kB)
Lua	~2,560	~72.5

CPython	~12,600	~44.4
MicroPython	~33	-
JavaScript	~43,600	~5,300

- Lua exhibited the smallest memory footprint by a large margin. Its process RSS usage remained around 2.5 MB, and its internal Lua-managed memory was also minimal.
- Python's RSS memory footprint (~12.6 MB) was notably larger but understandable given the size of the Python runtime. The `tracemalloc` module showed that the memory specifically allocated for Python heap objects during the control logic execution was extremely small (~44 KB), confirming that the control logic itself is lightweight.
- MicroPython consumed the least memory overall (≈ 40 kB RSS in the Unix port)
- JavaScript, while achieving the fastest execution speed, consumed the most memory overall. Node.js brought a large runtime overhead, with total process memory around 43–44 MB, and internal heap usage around 5 MB.

5.3 Summary Tables and Charts

Table 9 below summarizes performance results, including execution time, RSS and internal heap memory usage, syntax simplicity, and embedded friendliness. Figure 1 is an intuitive visualization of execution time and memory usage for each language.

Table 9. Performance table comparison for all 3 languages

Metric	Lua	Python	MicroPython	JavaScript
Execution Time (ms)	~1.86	~1.53	≈ 4	~0.97
RSS Memory Usage (kB)	~2,560	~12,600	~ 40	~43,600
Internal Heap Memory (kB)	~72.5	~44.4	-	~5,300
Syntax/Code Simplicity	High	Very High	High	High
Embedded Friendliness	Excellent	Moderate	Excellent	Poor

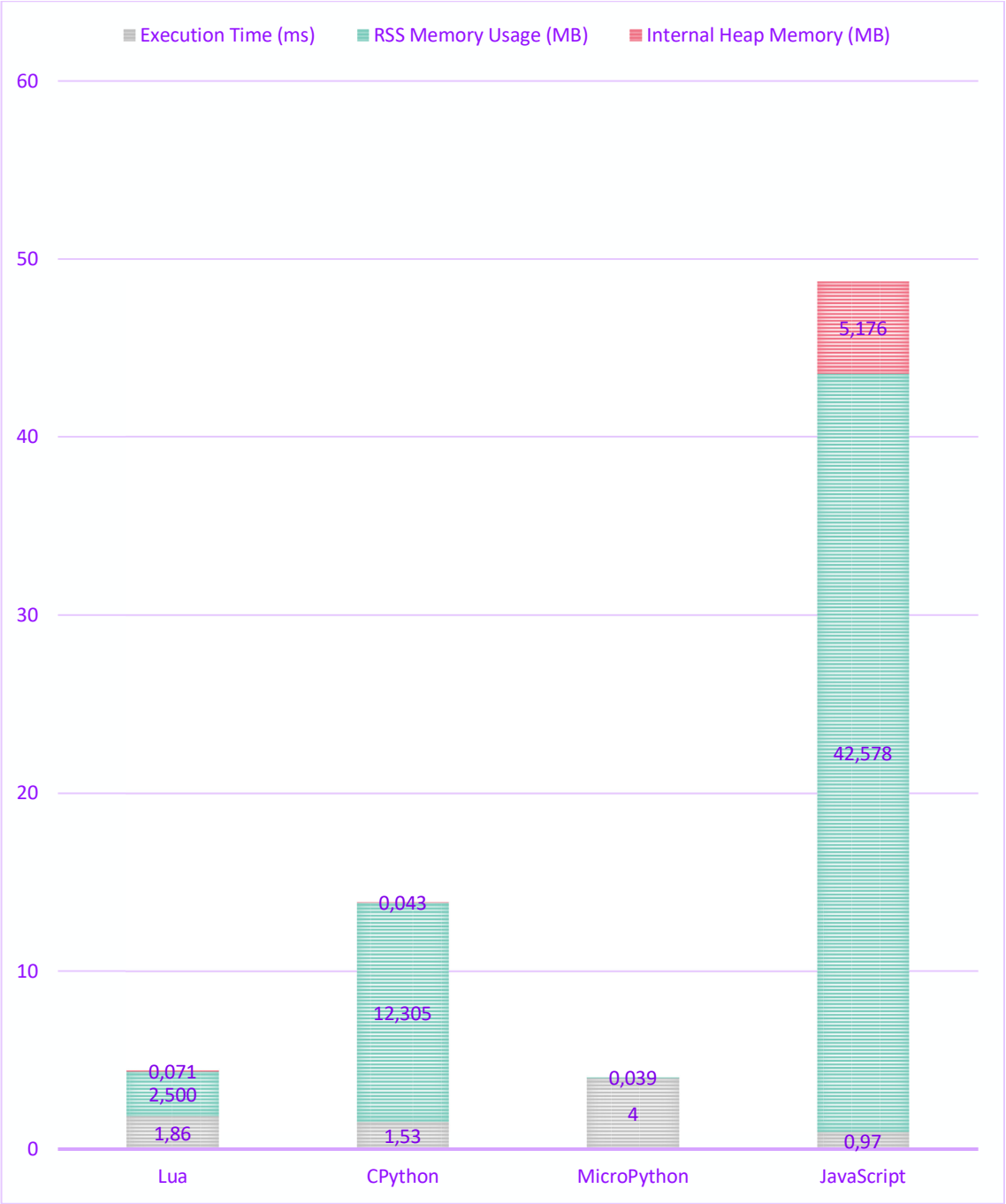


Figure 1. Stacked column chart for Lua, CPython, MicroPython and JavaScript

5.4 Technical Trade-offs

While Lua, Python, MicroPython and JavaScript each successfully implemented the heating control logic, their suitability varies significantly based on real-world system constraints:

- **Lua:** With minimal memory usage (~2560 kB) and compact syntax, Lua is ideal for embedded PLCs at Bithouse Oy, as described in Sections 3.1–

3.2. Although its execution speed (~0.0186 ms/loop) was slightly slower due to interpreter overhead from operations like `table.sort`, this impact is negligible for typical PLC tasks. Lua's lightweight runtime and ease of embedding into C-based firmware outweigh its limited debugging tools, especially when managing small scripts for client-specific logic.

- **MicroPython** is even lighter, but its slower loop time (~0.04 ms per cycle) makes Lua the faster choice when both can fit.
- **Python:** Each loop runs quickly (≈ 0.015 ms), helped by C-optimised calls such as `sorted()` and by Python's clear, maintainable syntax. The drawback is size: the interpreter and standard library raise RSS to roughly 12 MB, which is heavy for most PLCs. Python therefore shines on edge devices or dashboard servers, where extra memory is available and its rich ecosystem speeds development.
- **JavaScript (Node.js):** JavaScript demonstrated the fastest performance (~0.0097 ms/loop) thanks to JIT compilation, but its significant runtime overhead (~43.58 MB RSS) restricts practical deployment to more powerful hardware or cloud-based applications. JavaScript is optimal for web interfaces, cloud-based dashboards, or server-side components in BMS scenarios.

Table 10 below is a comparative summary based on testing and discussion of the results.

Table 10. Summary of Trade-offs

Language	Memory Footprint	Execution Speed	Ideal Use Case
Lua	Very low	Moderate	Embedded PLCs, resource-constrained environments
Python	Moderate-high	Fast	Edge computing, prototyping, dashboards
MicroPython	Very low	Low	Ideal for sensor nodes or ultra-small controllers
JavaScript	High	Very fast	Cloud application, web dashboards

Lua is a natural fit for Bithouse Oy's embedded automation needs, while Python and JavaScript are more appropriate for edge nodes or cloud services. Future exploration could include testing Lua directly on physical PLC hardware, optimizing Python and JavaScript for embedded scenarios, or leveraging advanced Lua features such as metatables (Section 3.3) for enhanced automation flexibility.

6 DISCUSSION AND RECOMMENDATIONS

This chapter synthesizes the findings from previous chapters, interpreting them in practical terms relevant to building automation systems, particularly those employed by Bithouse Oy. It highlights Lua's effectiveness, identifies situations where Lua may not be ideal, and offers actionable recommendations for developers and integrators.

6.1 Why Lua is Effective in Building Automation

Lua has demonstrated strong suitability for building automation, particularly in scenarios prioritizing flexibility, efficiency, and embedded compatibility. Key strengths include:

- **Minimal memory footprint:** Lua's memory use (~2.5 MB RSS) is significantly lower compared to Python (~12.6 MB) and JavaScript (~43.6 MB), making it ideal for resource-constrained embedded PLC environments.
- **Fast enough for real-time control:** Despite lacking just-in-time (JIT) compilation, Lua's execution (~0.0186 ms per loop) is sufficiently fast for typical HVAC and BMS control cycles.
- **Dynamic configuration capabilities:** Lua scripts can be dynamically reloaded at runtime without rebooting, enabling rapid updates to logic or configurations on-site.
- **Ease of embedding:** Lua integrates seamlessly with existing C/C++ firmware, effectively separating stable core logic from customizable, client-specific scripts. This approach improves maintainability and accelerates deployment.
- **Extensibility and metaprogramming:** Lua's metatables and simple, table-based structure facilitate object-oriented patterns and reusable automation logic, enhancing adaptability across different client installations.

These strengths allow a stable firmware core complemented by flexible Lua scripts, significantly improving deployment efficiency, reducing downtime, and enabling quick adjustments to changing requirements.

6.2 When Lua Might Not Be the Best Fit

Despite its advantages, Lua has certain limitations that make it less suitable for some applications:

- **Limited third-party ecosystem:** Lua's minimal standard library and limited third-party libraries restrict its suitability for complex tasks like advanced analytics, cloud integration, and rich visualizations, areas where Python and JavaScript excel.
- **Dynamic typing and maintainability:** Lua's lack of built-in static typing can complicate maintenance and scalability in large, team-driven projects, particularly when documentation or developer familiarity is limited.
- **Learning curve for advanced features:** Lua's powerful features, such as metatables and coroutines, can be unintuitive initially, requiring additional training or detailed documentation.
- **Performance limits without JIT:** While adequate for typical automation tasks, Lua's interpreted nature can limit performance in CPU-intensive computations compared to JIT-compiled JavaScript or optimized Python libraries. Although LuaJIT mitigates this issue, it's not always viable in strictly constrained embedded environments.

In summary, Lua excels in embedded scripting and lightweight automation tasks but is less suitable for complex full-stack applications, extensive data processing, or comprehensive user interfaces.

6.3 Practical Advice for Developers and Integrators

For effective and maintainable Lua usage in building automation, developers and integrators should follow these best practices:

- **Separate core firmware from scripting logic:** Leverage Lua strictly for user-facing logic (setpoints, alarms, schedules), allowing compiled languages (C/C++) to handle time-critical and hardware-level tasks.
- **Structure data simply and clearly:** Favor flat, easily readable tables over deeply nested structures, enhancing script clarity and performance.

- **Centralize and externalize configuration data:** Keep configuration parameters separate from control logic, enabling script reuse across multiple deployments with minimal adjustments.
- **Document scripts thoroughly, especially metatable usage**: Clear documentation reduces confusion over hidden behaviours, facilitating easier maintenance and onboarding for developers unfamiliar with Lua's advanced features.
- **Benchmark performance early:** Conduct early-stage benchmarking of scripts on target hardware, particularly for resource-intensive operations (e.g., sorting large datasets), to ensure compatibility and optimal performance.
- **Complement Lua with other languages as needed:** Use Lua primarily for embedded tasks, while selecting Python for data-heavy dashboards or analytics, and JavaScript for web-based interfaces or cloud integrations.

Applying these practical recommendations will help maximize Lua's advantages, effectively addressing Bithouse Oy's diverse building automation needs and ensuring scalable, robust, and easily maintainable control solutions.

7 CONCLUSION AND FUTURE WORK

This chapter recaps the key results of the thesis, reviews Lua's wider promise for building-automation work on Bithouse Oy's embedded PLCs and outlines next steps—ranging from tighter IoT links and richer remote-control schemes to additional performance tuning.

7.1 Summary of Key Findings

This thesis evaluated Lua's suitability for building automation, specifically in HVAC and Building Management Systems (BMS). Through comparative implementation, empirical testing, and analysis, several key findings emerged:

- **Optimal for embedded environments:** Lua's minimal memory footprint (~2.5 MB RSS) makes it exceptionally well-suited for embedded and resource-constrained PLC environments, clearly outperforming Python (~12.6 MB RSS) and JavaScript (~43.6 MB RSS).
- **Balanced execution performance:** Despite its interpreted nature and minor overhead from table operations (e.g., `table.sort`), Lua executed control logic quickly enough (~0.0186 ms per loop) to comfortably meet typical real-time HVAC requirements.
- **Compact and maintainable scripts:** Lua's concise syntax and table-based architecture enabled clear, maintainable code (~50 lines for heating control logic), facilitating rapid development and ease of customization across client-specific deployments.
- **Ease of embedding and dynamic updates:** Lua plugs directly into C/C++ firmware and its scripts can be reloaded at runtime, so engineers can make client-specific adjustments in minutes without taking the system offline.
- **Effective metaprogramming features:** Metatables provided powerful extensibility, supporting modular and reusable control logic structures, essential for managing diverse building automation scenarios efficiently.

In short, Lua offers an outstanding balance of performance, flexibility and ease of deployment, cementing its position as the ideal scripting choice for embedded automation tasks at Bithouse Oy.

7.2 Possibility for Future Applications

This thesis focused primarily on embedded automation control logic, but several avenues for future research and development could significantly enhance Lua's application in building automation:

IoT Integration: Lua's lightweight design is well-suited for deployment in IoT scenarios, such as small embedded microcontrollers (e.g., ESP32 with NodeMCU). Integrating Lua scripts into IoT endpoints would enable scalable, distributed control systems with efficient edge-computing capabilities and remote configuration over network interfaces.

Advanced Remote Procedure Calls (RPC): Building on Bithouse Oy's existing Lua-based RPC implementations, future work could develop more robust and efficient RPC frameworks. This would improve remote monitoring, diagnostics, and real-time adjustments of building systems from centralized management platforms.

LuaJIT for Performance Enhancement: In scenarios with less restrictive platform constraints, utilizing LuaJIT could significantly improve execution speed for computation-intensive tasks or large-scale deployments. Evaluating LuaJIT's performance benefits in realistic PLC scenarios could be valuable.

Security and Sandboxing: As complexity and remote scripting capabilities increase, it becomes critical to explore effective sandboxing and secure execution methods. Research into secure Lua environments can lead to more secure execution of user-generated or remotely loaded scripts.

Enhanced Development Tooling: Expanding the available debugging, live scripting, and validation tools would greatly improve usability for integrators and

developers. Better tooling can reduce the learning curve and enhance script maintainability.

Hybrid Systems and Visualization: Combining Lua's embedded strengths with Python or JavaScript-based dashboards could create comprehensive, hybrid solutions. Future studies could evaluate integrated scripting platforms, leveraging Lua for embedded control logic and Python or JavaScript for data visualization, analytics, and cloud connectivity.

In summary, Lua's advantages fit neatly with emerging needs in embedded automation, distributed IoT, and scalable building-management platforms. Ongoing R&D in these fields will only strengthen Lua's relevance—both for Bithouse Oy's day-to-day projects and for the wider building-automation sector.

REFERENCES

Casio. (2025). *FX-9860 G III—Python-enabled scientific calculator*. Casio. Retrieved 5.5.2025. https://www.casio.com/intl/scientific-calculators/product.FX-9860GIII/

Dhambarage, R. (2022). Understanding Python for embedded systems developers. *Medium*. Retrieved 12.5.2025. https://ruvid.medium.com/understanding-python-for-embedded-systems-developers-2a3310885d13

Heidary, R., Prasad Rao, J., Pinon Fischer, O.J. (2023). Smart Buildings in the IoT Era – Necessity, Challenges, and Opportunities. In: Fathi, M., Zio, E., Pardalos, P.M. (eds) *Handbook of Smart Energy Systems.* Springer, Cham. Retrieved 5.5.2025. https://doi.org/10.1007/978-3-030-72322-4_115-1

Ierusalimschy, R. (2003). Programming in Lua (first edition). *Lua.org*. Retrieved 5.5.2025. https://www.lua.org/pil/contents.html

Ierusalimschy, R., de Figueiredo, L.H. and Celes, W. (2025). The evolution of Lua, continued. *Lua.org*. Retrieved 5.5.2025. https://www.lua.org/doc/cola.pdf

Marinescu, B., & Sutter, D. (n.d.). *eLua Project*. eLua Project. Retrieved 5.5.2025. http://www.eluaproject.net/

MicroPython. (2025). MicroPython Documentation. *MicroPython Project*. Retrieved 12.5.2025. https://docs.micropython.org/

MicroPython. (2025). MicroPython—Python for Microcontrollers. *MicroPython Project*. Retrieved 12.5.2025. https://micropython.org/

Plauska, I., Liutkevicius, A. & Janaviciute, A. (2023). Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics,* 12(1). Retrieved 5.5.2025. https://www.mdpi.com/2079-9292/12/1/143

APPENDICES

Appendix 1. Lua Script - hvacex_limitControl.lua

```
function hvac_limitControl(pointList)

    for i,v in ipairs(pointList) do
        hvac_block(v)
    end

end

function hvac_block(id)

    local pointData = Data.get(id)

    local enabled = tonumber ( Data.get (pointData.enabledId..".pv") )
    if enabled ~= nil then
        pointData.enabled = enabled
        Data.set (id..".enabled",enabled)
    end

    local input = tonumber ( Data.get(pointData.inputId..".pv") )
    if input ~= nil then
        pointData.input = input
        Data.set(id..".input",input)
    end

    if (pointData.enabled or 0) < 1 then
        Data.set(id..".pv",0)

    else
        local output = hvac_controlBlock (pointData)
        Data.set(id..".pv",output)

    end

    Data.set(pointData.outputId..".pv",pointData.pv)
end

function hvac_controlBlock(data)

    local result = data.pv

    if data.comparisonType:match("lowLimit") then
        if data.input < data.controlLowLimit then
            result = 1
        elseif data.input >= (data.controlLowLimit + data.hysteresis)
then
```

```
            result = 0
        end
        return result

    elseif data.comparisonType:match("highLimit") then
        if data.input > data.controlHighLimit then
            result = 1
        elseif data.input <= (data.controlHighLimit - data.hysteresis)
then
            result = 0
        end
        return result

    elseif data.comparisonType:match("between") then
        if data.input < data.controlHighLimit and data.input > data.con-
trolLowLimit then
            result = 1
        elseif data.input >= (data.controlHighLimit + data.hysteresis) or
data.input <= (data.controlLowLimit - data.hysteresis) then
            result = 0
        end
        return result
    end

end
```

Appendix 2. The heating control program scripts and the output

heating.lua

```
-- Config parameters
local setpoint = 21.0
local hysteresis = 0.5

-- Define a temperature-to-output curve (X = outdoor temp, Y = output
level)
local heatingCurve = {
    points = {
        { pointX = 22, pointY = 0.0 }, -- No heating needed
        { pointX = 20, pointY = 0.3 },
        { pointX = 18, pointY = 0.6 },
        { pointX = 16, pointY = 1.0 } -- Full heating
    }
}
function getHeatingOutput(curve, outdoorTemp)
    table.sort(curve.points, function(a, b) return a.pointX < b.pointX
end)

    if outdoorTemp <= curve.points[1].pointX then
        return curve.points[1].pointY
    elseif outdoorTemp >= curve.points[#curve.points].pointX then
        return curve.points[#curve.points].pointY
    end

    for i = 1, #curve.points - 1 do
        local p1, p2 = curve.points[i], curve.points[i + 1]
        if outdoorTemp >= p1.pointX and outdoorTemp <= p2.pointX then
            local slope = (p2.pointY - p1.pointY) / (p2.pointX -
p1.pointX)
            return (outdoorTemp - p1.pointX) * slope + p1.pointY
        end
    end

    return 0.0
end

function thermostatControl(indoorTemp, setpoint, hysteresis, prevState)
    if indoorTemp < (setpoint - hysteresis) then
        return 1         -- Heating ON
    elseif indoorTemp > (setpoint + hysteresis) then
        return 0         -- Heating OFF
    else
        return prevState -- Maintain current state
    end
end

-- Simulation inputs
```

```lua
local indoorTemps = { 20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6 }
local outdoorTemps = { 21, 20, 15, 18, 17, 16, 15, 14 }

local state = 0 -- Initial heating state: OFF

print("Indoor | Outdoor | Output | Heating")
print("-------------------------------------")

for i = 1, #indoorTemps do
    local indoor = indoorTemps[i]
    local outdoor = outdoorTemps[i]
    local output = getHeatingOutput(heatingCurve, outdoor)
    local heating = thermostatControl(indoor, setpoint, hysteresis,
state)
    state = heating

    print(string.format(" %5.1f |   %5.1f  |  %4.1f  |   %s",
        indoor, outdoor, output * 100, (heating == 1 and "ON" or "OFF")))
end
```

Output from Terminal:

```
Indoor | Outdoor | Output | Heating
-------------------------------------
  20.1 |    21.0  |  15.0  |   ON
  20.0 |    20.0  |  30.0  |   ON
  19.8 |    15.0  |  100.0  |    ON
  20.2 |    18.0  |  60.0  |   ON
  21.3 |    17.0  |  80.0  |   ON
  21.7 |    16.0  |  100.0  |    OFF
  21.0 |    15.0  |  100.0  |    OFF
  20.6 |    14.0  |  100.0  |    OFF
```

heating.py

```python
# Config parameters
setpoint = 21.0
hysteresis = 0.5
# Define curve: maps outdoor temperature (X) to heating output (Y)
heating_curve = [
    {"pointX": 22, "pointY": 0.0},
    {"pointX": 20, "pointY": 0.3},
    {"pointX": 18, "pointY": 0.6},
    {"pointX": 16, "pointY": 1.0}
]

def get_heating_output(curve, outdoor_temp):
    # Sort the curve points
    curve = sorted(curve, key=lambda p: p["pointX"])
```

```
    if outdoor_temp <= curve[0]["pointX"]:
        return curve[0]["pointY"]
    if outdoor_temp >= curve[-1]["pointX"]:
        return curve[-1]["pointY"]
    for i in range(len(curve) - 1):
        p1, p2 = curve[i], curve[i + 1]
        if p1["pointX"] <= outdoor_temp <= p2["pointX"]:
            slope = (p2["pointY"] - p1["pointY"]) / (p2["pointX"] -
p1["pointX"])
            return (outdoor_temp - p1["pointX"]) * slope + p1["pointY"]
    return 0.0

def thermostat_control(indoor_temp, setpoint, hysteresis, prev_state):
    if indoor_temp < setpoint - hysteresis:
        return 1  # Turn heating ON
    elif indoor_temp > setpoint + hysteresis:
        return 0  # Turn heating OFF
    return prev_state  # Keep previous state

# Simulated temperature readings
indoor_temps = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6]
outdoor_temps = [21, 20, 15, 18, 17, 16, 15, 14]
state = 0  # Initial state: OFF

print("Indoor | Outdoor | Output | Heating")
print("-------------------------------------")
for i in range(len(indoor_temps)):
    indoor = indoor_temps[i]
    outdoor = outdoor_temps[i]
    output = get_heating_output(heating_curve, outdoor)
    heating = thermostat_control(indoor, setpoint, hysteresis, state)
    state = heating
    print(f" {indoor:5.1f} |   {outdoor:5.1f}   |  {output *
100:4.1f}  |   {'ON' if heating == 1 else 'OFF'}")
```

Output:

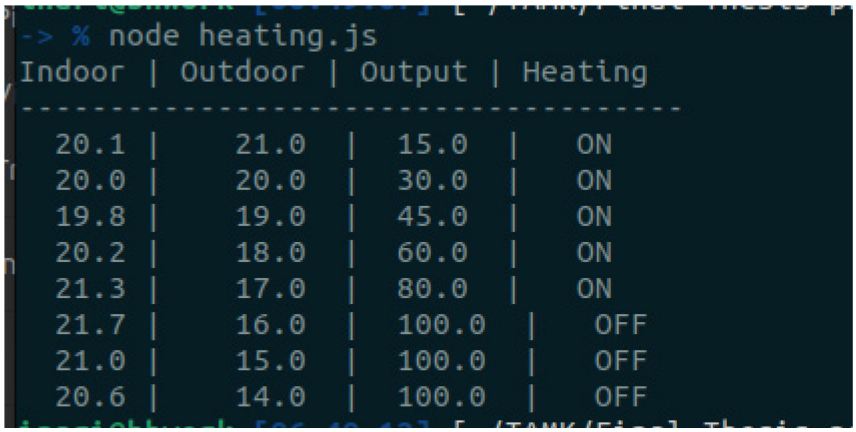
```
-> % python3 heating.py
Indoor | Outdoor | Output | Heating
-------------------------------------
  20.1 |    21.0  |  15.0  |   ON
  20.0 |    20.0  |  30.0  |   ON
  19.8 |    15.0  |  100.0  |   ON
  20.2 |    18.0  |  60.0  |   ON
  21.3 |    17.0  |  80.0  |   ON
  21.7 |    16.0  |  100.0  |   OFF
  21.0 |    15.0  |  100.0  |   OFF
  20.6 |    14.0  |  100.0  |   OFF
```

heating.js

```
// Configuration parameters
const setpoint = 21.0;
const hysteresis = 0.5;
// Heating curve: maps outdoor temp (X) to output level (Y)
const heatingCurve = [
    { pointX: 22, pointY: 0.0 },
    { pointX: 20, pointY: 0.3 },
    { pointX: 18, pointY: 0.6 },
    { pointX: 16, pointY: 1.0 }
];
const indoorTemps = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6];
const outdoorTemps = [21, 20, 15, 18, 17, 16, 15, 14];
let state = 0;

function getHeatingOutput(curve, outdoorTemp) {
    const sortedCurve = curve.sort((a, b) => a.pointX - b.pointX);
    if (outdoorTemp <= sortedCurve[0].pointX) {
        return sortedCurve[0].pointY;
    } else if (outdoorTemp >= sortedCurve[sortedCurve.length - 1].pointX)
{
        return sortedCurve[sortedCurve.length - 1].pointY;
    }
    for (let i = 0; i < sortedCurve.length - 1; i++) {
        const p1 = sortedCurve[i];
        const p2 = sortedCurve[i + 1];
        if (p1.pointX <= outdoorTemp && outdoorTemp <= p2.pointX) {
            let slope = (p2.pointY - p1.pointY) / (p2.pointX -
p1.pointX);
            return (outdoorTemp - p1.pointX) * slope + p1.pointY;
        }
    }
    return 0.0;
}

function thermostatControl(indoorTemp, setpoint, hysteresis, prevState) {
    if (indoorTemp < setpoint - hysteresis) {
        return 1;  // ON
    } else if (indoorTemp > setpoint + hysteresis) {
        return 0;  // OFF
    } else {
        return prevState;
    }
}

console.log("Indoor | Outdoor | Output | Heating");
console.log("-------------------------------------");
for (let i = 0; i < indoorTemps.length; i++) {
    const indoor = indoorTemps[i];
    const outdoor = outdoorTemps[i];
```

```
    const output = getHeatingOutput(heatingCurve, outdoor);
    const heating = thermostatControl(indoor, setpoint, hysteresis,
state);
    state = heating;
    console.log(` ${indoor.toFixed(1).padStart(5)} |   ${out-
door.toFixed(1).padStart(5)}  |  ${(output * 100).toFixed(1).pad-
Start(4)}  |   ${heating === 1 ? 'ON' : 'OFF'}`);
}
```

Output:



Appendix 3. Full measurement scripts and additional test logs

test.lua

```
local startTime = os.clock()

-- Memory usage (RSS from /proc/self/status -- Linux system)
local function getMemoryUsageKB()
    local f = io.open("/proc/self/status", "r")
    if not f then return 0 end
    for line in f:lines() do
        local mem = line:match("^VmRSS:%s+(%d+)")
        if mem then
            f:close()
            return tonumber(mem)
        end
    end
    f:close()
    return 0
end

-- Config parameters
local setpoint = 21.0
local hysteresis = 0.5
local heatingCurve = {
    points = {
        {pointX = 22, pointY = 0.0},
        {pointX = 20, pointY = 0.3},
        {pointX = 18, pointY = 0.6},
        {pointX = 16, pointY = 1.0}
    }
}

-- Curve function: Output scaling
function getHeatingOutput(curve, outdoorTemp)
    table.sort(curve.points, function(a, b) return a.pointX < b.pointX
end)
    if outdoorTemp <= curve.points[1].pointX then
        return curve.points[1].pointY
    elseif outdoorTemp >= curve.points[#curve.points].pointX then
        return curve.points[#curve.points].pointY
    end
    for i = 1, #curve.points - 1 do
        local p1, p2 = curve.points[i], curve.points[i + 1]
        if outdoorTemp >= p1.pointX and outdoorTemp <= p2.pointX then
            local slope = (p2.pointY - p1.pointY) / (p2.pointX -
p1.pointX)
            return (outdoorTemp - p1.pointX) * slope + p1.pointY
        end
    end
    return 0.0
```

```
end

-- Limit control function: Threshold logic
function thermostatControl(indoorTemp, setpoint, hysteresis, prevState)
    if indoorTemp < (setpoint - hysteresis) then
        return 1
    elseif indoorTemp > (setpoint + hysteresis) then
        return 0
    else
        return prevState
    end
end

-- Simulation inputs
local indoorTemps = {20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6}
local outdoorTemps = {21, 20, 19, 18, 17, 16, 15, 14}

-- Rum 100 iterations
for loop = 1, 100 do
    local state = 0
    for i = 1, #indoorTemps do
        local indoor = indoorTemps[i]
        local outdoor = outdoorTemps[i]
        local output = getHeatingOutput(heatingCurve, outdoor)
        local heating = thermostatControl(indoor, setpoint, hysteresis,
state)
        state = heating
    end
end

-- Measure execution time and memory
local endTime = os.clock()
local endMem = getMemoryUsageKB()
local internalMem = collectgarbage("count")
local elapsedTime = (endTime - startTime) * 1000

-- Print results
print(string.format("Execution time (100 iterations): %.3f ms",
elapsedTime))
print(string.format("Memory usage (RSS, kB): %d", endMem))
print(string.format("Memory usage (internal, kB): %.1f", internalMem))

os.execute("sleep 1")
```

Output:

```
-> % for i in {1..5}; do lua test.lua; done
Execution time (100 iterations): 2.975 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
Execution time (100 iterations): 1.881 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
Execution time (100 iterations): 1.861 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
Execution time (100 iterations): 1.853 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
Execution time (100 iterations): 1.645 ms
Memory usage (RSS, kB): 2560
Memory usage (internal, kB): 72.5
```

test.py

```python
import time
import os
import psutil
import tracemalloc

# Start timing and memory tracking
tracemalloc.start()
start_time = time.perf_counter()
process = psutil.Process(os.getpid())

# Config
setpoint = 21.0
hysteresis = 0.5
heating_curve = [
    {"pointX": 22, "pointY": 0.0},
    {"pointX": 20, "pointY": 0.3},
    {"pointX": 18, "pointY": 0.6},
    {"pointX": 16, "pointY": 1.0}
]

# Curve function: Output Scaling
def get_heating_output(curve, outdoor_temp):
    curve = sorted(curve, key=lambda p: p["pointX"])
    if outdoor_temp <= curve[0]["pointX"]:
        return curve[0]["pointY"]
    elif outdoor_temp >= curve[-1]["pointX"]:
        return curve[-1]["pointY"]
    for i in range(len(curve) - 1):
        p1, p2 = curve[i], curve[i + 1]
        if p1["pointX"] <= outdoor_temp <= p2["pointX"]:
            slope = (p2["pointY"] - p1["pointX"]) / (p2["pointX"] -
p1["pointX"])
```

```
            return (outdoor_temp - p1["pointX"]) * slope + p1["pointY"]
    return 0.0

# Limit control function: Threshold logic
def thermostat_control(indoor_temp, setpoint, hysteresis, prev_state):
    if indoor_temp < (setpoint - hysteresis):
        return 1
    elif indoor_temp > (setpoint + hysteresis):
        return 0
    else:
        return prev_state

# Simulation inputs
indoor_temps = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6]
outdoor_temps = [21, 20, 19, 18, 17, 16, 15, 14]

# Run 100 iterations
for _ in range(100):
    state = 0
    for indoor, outdoor in zip(indoor_temps, outdoor_temps):
        output = get_heating_output(heating_curve, outdoor)
        heating = thermostat_control(indoor, setpoint, hysteresis, state)
        state = heating

# Measure execution time and memory
end_time = time.perf_counter()
elapsed_time_ms = (end_time - start_time) * 1000

rss_memory_kb = process.memory_info().rss / 1024  # Full process RSS
current, peak = tracemalloc.get_traced_memory()
heap_memory_kb = peak / 1024                      # Python internal heap

tracemalloc.stop()

# Print results
print(f"Execution time (100 iterations): {elapsed_time_ms:.3f} ms")
print(f"Memory usage (RSS, kB): {rss_memory_kb:.2f}")
print(f"Memory usage (tracemalloc heap, kB): {heap_memory_kb:.2f}")

time.sleep(1)  # Keep process alive for ps checking if needed
```

Output:

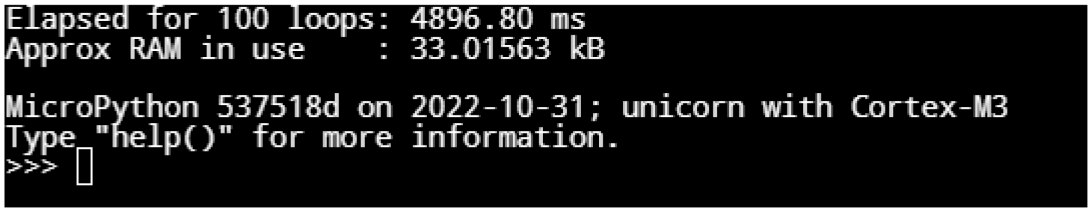
```
-> % for i in {1..5}; do python3 test.py; done
Execution time (100 iterations): 1.519 ms
Memory usage (RSS, kB): 12544.00
Memory usage (tracemalloc heap, kB): 44.41
Execution time (100 iterations): 1.527 ms
Memory usage (RSS, kB): 12544.00
Memory usage (tracemalloc heap, kB): 44.41
Execution time (100 iterations): 1.565 ms
Memory usage (RSS, kB): 12544.00
Memory usage (tracemalloc heap, kB): 44.41
Execution time (100 iterations): 1.494 ms
Memory usage (RSS, kB): 12672.00
Memory usage (tracemalloc heap, kB): 44.41
Execution time (100 iterations): 1.485 ms
Memory usage (RSS, kB): 12672.00
Memory usage (tracemalloc heap, kB): 44.41
```

test_mpy.py

```python
# MicroPython demo – heating control loop
# test_mpy.py  –  MicroPython version, 100 iterations

import time, gc

# ---------------- Configuration ----------------
setpoint   = 21.0
hysteresis = 0.5
heating_curve = [
    {"x": 22, "y": 0.0},
    {"x": 20, "y": 0.3},
    {"x": 18, "y": 0.6},
    {"x": 16, "y": 1.0},
]

# -------------- Helper functions ---------------
def get_heating_output(curve, outdoor):
    curve = sorted(curve, key=lambda p: p["x"])
    if outdoor <= curve[0]["x"]:
        return curve[0]["y"]
    if outdoor >= curve[-1]["x"]:
        return curve[-1]["y"]
    for p1, p2 in zip(curve, curve[1:]):
        if p1["x"] <= outdoor <= p2["x"]:
            slope = (p2["y"] - p1["y"]) / (p2["x"] - p1["x"])
            return (outdoor - p1["x"]) * slope + p1["y"]
    return 0.0

def thermostat(indoor, setp, hyst, prev):
```

```
    if indoor < setp - hyst:
        return 1      # ON
    if indoor > setp + hyst:
        return 0      # OFF
    return prev       # keep state

# -------------- Simulation vectors -------------
indoor  = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6]
outdoor = [21,   20,   19,   18,   17,   16,   15,   14]

# ---------------- Benchmark loop ---------------
state = 0
t0 = time.ticks_us()

for _ in range(100):                    # 100 iterations
    state = 0
    for tin, tout in zip(indoor, outdoor):
        pct   = get_heating_output(heating_curve, tout)
        state = thermostat(tin, setpoint, hysteresis, state)

elapsed_ms = time.ticks_diff(time.ticks_us(), t0) / 1000

print("Elapsed for 100 loops: {:.2f} ms".format(elapsed_ms))
print("Approx RAM in use    : {} kB".format(gc.mem_alloc() / 1024
))
```

Output (tested in web simulator https://micropython.org/unicorn/):

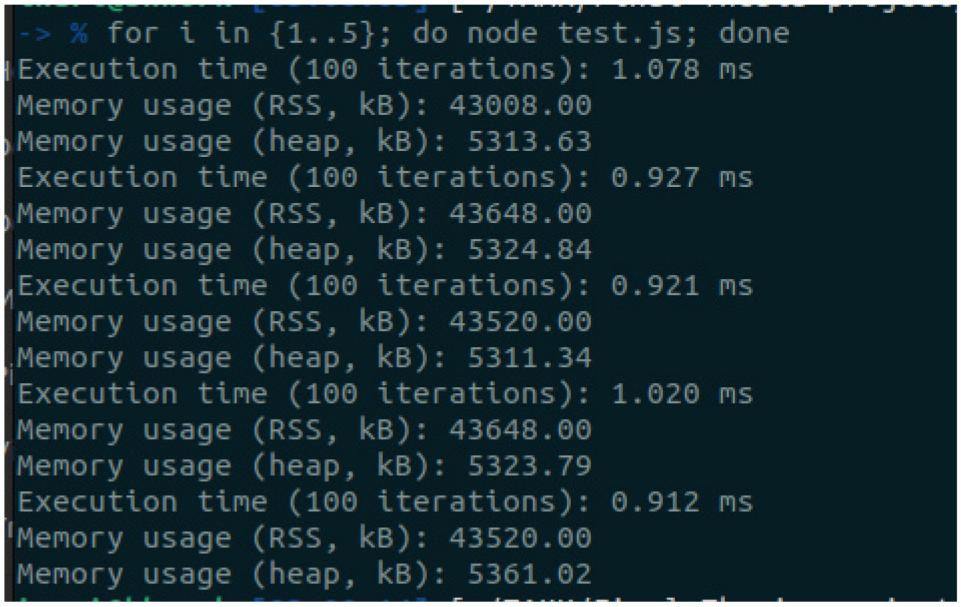


test.js

```
const startTime = process.hrtime.bigint();
const startMem = process.memoryUsage();

const setpoint = 21.0;
const hysteresis = 0.5;

let heatingCurve = [
  { pointX: 22, pointY: 0.0 },
  { pointX: 20, pointY: 0.3 },
  { pointX: 18, pointY: 0.6 },
  { pointX: 16, pointY: 1.0 }
];

// Curve function: Output scaling
```

```
function getHeatingOutput(curve, outdoorTemp) {
  curve.sort((a, b) => a.pointX - b.pointX);
  if (outdoorTemp <= curve[0].pointX) return curve[0].pointY;
  if (outdoorTemp >= curve[curve.length - 1].pointX) return
curve[curve.length - 1].pointY;
  for (let i = 0; i < curve.length - 1; i++) {
    const p1 = curve[i], p2 = curve[i + 1];
    if (outdoorTemp >= p1.pointX && outdoorTemp <= p2.pointX) {
      const slope = (p2.pointY - p1.pointY) / (p2.pointX - p1.pointX);
      return (outdoorTemp - p1.pointX) * slope + p1.pointY;
    }
  }
  return 0.0;
}

// Limit control function: Threshold logic
function thermostatControl(indoorTemp, setpoint, hysteresis, prevState) {
  if (indoorTemp < (setpoint - hysteresis)) return 1;
  if (indoorTemp > (setpoint + hysteresis)) return 0;
  return prevState;
}

// Simulation inputs
const indoorTemps = [20.1, 20.0, 19.8, 20.2, 21.3, 21.7, 21.0, 20.6];
const outdoorTemps = [21, 20, 19, 18, 17, 16, 15, 14];

// Run 100 iterations
for (let loop = 0; loop < 100; loop++) {
  let state = 0;
  for (let i = 0; i < indoorTemps.length; i++) {
    let indoor = indoorTemps[i];
    let outdoor = outdoorTemps[i];
    let output = getHeatingOutput(heatingCurve, outdoor);
    let heating = thermostatControl(indoor, setpoint, hysteresis, state);
    state = heating;
  }
}

// Measure execution time and memory
const endTime = process.hrtime.bigint();
const elapsedMs = Number(endTime - startTime) / 1000000;
const endMem = process.memoryUsage();

// Print results
console.log(`Execution time (100 iterations): ${elapsedMs.toFixed(3)}
ms`);
console.log(`Memory usage (RSS, kB): ${(endMem.rss / 1024).toFixed(2)}`);
console.log(`Memory usage (heap, kB): ${(endMem.heapUsed /
1024).toFixed(2)}`)

setTimeout(()=> {}, 1000);
```

Output:


-> % for i in {1..5}; do node test.js; done
Execution time (100 iterations): 1.078 ms
Memory usage (RSS, kB): 43008.00
Memory usage (heap, kB): 5313.63
Execution time (100 iterations): 0.927 ms
Memory usage (RSS, kB): 43648.00
Memory usage (heap, kB): 5324.84
Execution time (100 iterations): 0.921 ms
Memory usage (RSS, kB): 43520.00
Memory usage (heap, kB): 5311.34
Execution time (100 iterations): 1.020 ms
Memory usage (RSS, kB): 43648.00
Memory usage (heap, kB): 5323.79
Execution time (100 iterations): 0.912 ms
Memory usage (RSS, kB): 43520.00
Memory usage (heap, kB): 5361.02