Department of Mechanical Engineering
ARBWCI
Alexander Davis
476C - 001

To: *Whomever It May Concern*

From: *Alexander Davis*

Date: *3/8/2026*

Re: *ArduPilot lua script instructions*

Introduction

This memo contains the knowledge of Ardupilot as well as importing lua scripts into the Mission Planner software accumulated for this Capstone Project. Mission Planner, the Parameters, then the lua script will be explained. Lua is a coding language not familiar to most people, and as such I learned just enough to make what we needed to work as well as troubleshooting. This memo is assuming you know the basics of ArduPilot products.

ArduPilot/Mission Planner

This Capstone Project entails making an autonomous usv for scanning canals for SRP. The boat's thruster layout consists of two rear thrusters working as skid steering, and one central lateral thruster for cross track error correction. In the Mission Planner program there is a preset thruster model for the skid steering frame, in the Full Parameter List, this assigns SERVO1 to 73 (Throttle Left) and SERVO3 to 74 (Throttle Right). In the Parameters, every time you change an option you must press "write" on the right side, you then must reconnect to the Cube again to apply the changes.

There is a tab named "SCR_ENABLE" if you set the value to 1 (Lua Scripts). This will enable you to import a lua file that has code that will modify the model in some way. Doing this makes a new tab in the config menu named MAVFtp which is essentially the files of the cube. If you have a lua code this is where you import it, after clicking MAVFtp click the plus to open the tree and click scripts, now you can right click the empty area and upload the file.

Lua Script for Lateral Thruster

Everything after this point is subject to specifically this capstone project but might still clarify the process. The code I set up is a string of functions that are connected where at the end the final function is called.

```
local LATERAL_FUNC = 94
local PWM_CENTER   = 1500
local PWM_RANGE    = 400

local UPDATE_MS    = 100
local KP = 0.25
local KD = 0.10
local MAX_NORM = 0.7

local last_xtrack = 0.0
local last_ms = millis():tofloat()
```

In Mission Planner, the servo can be assigned a function, in the program I set SERVO2 to 94 which is the "Script1" function. That is what you see above; it is calling the servo/s with Script1. You can also assign a minimum and maximum PWM pulse, which is the output direction for the servo. 1500 is neutral; the range set above makes the minimum 1100 (backwards) and the maximum 1900 (forwards). It is running the function every 100 ms, the KP is the coefficient of the proportional gain (how fast it speeds up), KD is the coefficient of derivative gain (ensures it doesn't overshoot), and the MAX_NORM caps the thruster at 70% power to not strain the thruster. The two "last" variables store the previous runs when the functions are running.

```
local function mission_item_to_loc(i)
  local m = mission:get_item(i)
  if not m then return nil end
  local loc = Location()
  loc:lat(m:x())
  loc:lng(m:y())
  loc:relative_alt(true)
  loc:alt(math.floor(m:z() * 100))
  return loc
end
```

This function asks if there is a mission, ask for waypoint "i" and its location. This is used in the next function.

```
local function cross_track_m(curr_loc, prev_loc, next_loc)
  local v_path = prev_loc:get_distance_NED(next_loc)
  local v_curr = prev_loc:get_distance_NED(curr_loc)
  local px, py = v_path:x(), v_path:y()
  local cx, cy = v_curr:x(), v_curr:y()
  local path_len = math.sqrt(px*px + py*py)
  if path_len < 0.5 then return 0.0 end
  local cross = (cx * py - cy * px) / path_len
  return cross / path_len
end
```

This function first gets the vector between the two waypoints, then gets the vector from the last waypoint to the current location. Then x and y variables are assigned to each vector; then the length of the path is calculated and set to 0 if it is too short. The cross track distance is then calculated and outputted.

```lua
local function set_lateral_norm(norm)
  norm = clamp(norm, -MAX_NORM, MAX_NORM)
  local pwm = math.floor(PWM_CENTER + PWM_RANGE * norm)
  SRV_Channels:set_output_pwm(LATERAL_FUNC, pwm)
end
```

This function ensures that the norm is within the max and min boundaries, and if not, it outputs either one.

```lua
local function update()
  local idx = mission:get_current_nav_index()
  local curr = ahrs:get_location()

  if not idx or idx < 2 or not curr then
    set_lateral_norm(0.0)
    return update, UPDATE_MS
  end

  local prev_wp = mission_item_to_loc(idx - 1)
  local next_wp = mission_item_to_loc(idx)

  if not prev_wp or not next_wp then
    set_lateral_norm(0.0)
    return update, UPDATE_MS
  end

  local xtrack = cross_track_m(curr, prev_wp, next_wp)
  local now_ms = millis():tofloat()
  local dt = (now_ms - last_ms) * 0.001
  if dt < 0.02 then dt = 0.02 end
  local dx = (xtrack - last_xtrack) / dt
  local cmd = -(KP * xtrack + KD * dx)
  set_lateral_norm(cmd)

  if math.floor(now_ms) % 1000 < UPDATE_MS then
    gcs:send_text(6, string.format("xtrack=%.2f m cmd=%.2f", xtrack, cmd))
  end

  last_xtrack = xtrack
  last_ms = now_ms
  return update, UPDATE_MS
end
```

This entire function pulls the current values, runs previous functions and outputs the required information that the controller needs for the servo. First, it pulls the current waypoint index and location; it then says if there are less than 2 waypoints set norm to 0. Then the previous and next waypoints are pulled from the index, and if there is an issue getting either, set norm to 0. Next, it uses the cross-track function, pulls the current time, and makes sure that the time difference is not too small to skew the calculations. Next, it calculates the change in the cross-track error and uses that to calculate the command to output to the servo and uses the set_lateral_norm function to clamp it. Finally, it sends the telemetry repeatedly as well as updates the last cross track, last time, and runs the function again.

Results

At the time of writing this Memo, there was no hardware acquired, meaning that everything that was changed and tested was through the Mission Planner's built-in simulation. As such, the physical layout of the simulations thrusters was only the two rear thrusters as that was a built-in layout. There is no way to use the built-in simulation to add a custom thruster, but there are other programs that may be able to simulate it until we get the hardware.

Appendix

```
-- canal_centering.lua

local LATERAL_FUNC = 94                                                    --
function 94 is "Script 1"
local PWM_CENTER   = 1500                                                  --
1500 is no output
local PWM_RANGE    = 400                                                -- +/-
thrust range 1900 to 1100

local UPDATE_MS    = 100                                                   --
runs every 100ms
local KP = 0.25                                                            --
proportional gain (speed up speed)
local KD = 0.10                                                            --
derivative gain (don't overshoot)
local MAX_NORM = 0.7                                                       --
thruster % limit

local last_xtrack = 0.0                                                    --
stores previous crosstrack data
local last_ms = millis():tofloat()                                         --
stores previous time data

local function clamp(x, lo, hi)
  if x < lo then return lo end
```

```
  if x > hi then return hi end
  return x
end



local function mission_item_to_loc(i)
  local m = mission:get_item(i)                                          --
asks for waypoint "i"
  if not m then return nil end
  local loc = Location()
  loc:lat(m:x())                                                         --
latitude output
  loc:lng(m:y())                                                         --
longitude output
  loc:relative_alt(true)
  loc:alt(math.floor(m:z() * 100))                                       --
altitude output
  return loc
end



local function cross_track_m(curr_loc, prev_loc, next_loc)
  local v_path = prev_loc:get_distance_NED(next_loc)                     --
gets the vector between waypoints
  local v_curr = prev_loc:get_distance_NED(curr_loc)                     --
gets the vector from previous waypoint to current location
  local px, py = v_path:x(), v_path:y()                                  --
assigns x and y variables
  local cx, cy = v_curr:x(), v_curr:y()                                -- for
both vectors
  local path_len = math.sqrt(px*px + py*py)                              --
calculates the length of the path
  if path_len < 0.5 then return 0.0 end                                 -- if
the path is too short, return 0
  local cross = (cx * py - cy * px) / path_len                           --
calculates the cross track error
  return cross / path_len
end
```

```lua
local function set_lateral_norm(norm)
  norm = clamp(norm, -MAX_NORM, MAX_NORM)                                    --
clamp the norm to the maximum allowed
  local pwm = math.floor(PWM_CENTER + PWM_RANGE * norm)                      --
calculate the PWM signal based on the norm
  SRV_Channels:set_output_pwm(LATERAL_FUNC, pwm)
end



local function update()
  local idx = mission:get_current_nav_index()                                --
gets the current waypoint index
  local curr = ahrs:get_location()                                           --
gets the current location of the vehicle

  if not idx or idx < 2 or not curr then
    set_lateral_norm(0.0)                                                 -- if
we don't have enough waypoints or location data, set lateral norm to 0
    return update, UPDATE_MS
  end

  local prev_wp = mission_item_to_loc(idx - 1)                               --
gets the previous waypoint location
  local next_wp = mission_item_to_loc(idx)                                   --
gets the next waypoint location

  if not prev_wp or not next_wp then
    set_lateral_norm(0.0)                                                 -- if
we can't get the waypoint locations, set lateral norm to 0
    return update, UPDATE_MS
  end

  local xtrack = cross_track_m(curr, prev_wp, next_wp)                       --
calculates the cross track error
  local now_ms = millis():tofloat()                                          --
gets the current time in milliseconds
  local dt = (now_ms - last_ms) * 0.001                                      --
calculates the time difference in seconds
  if dt < 0.02 then dt = 0.02 end                                            --
prevent division by very small dt
```

```
  local dx = (xtrack - last_xtrack) / dt                                    --
calculates the derivative of the cross track error
  local cmd = -(KP * xtrack + KD * dx)                                      --
calculates the control command using PD control
  set_lateral_norm(cmd)                                                     --
sets the lateral norm based on the control command

  if math.floor(now_ms) % 1000 < UPDATE_MS then
    gcs:send_text(6, string.format("xtrack=%.2f m cmd=%.2f", xtrack, cmd))   --
sends telemetry to GCS every second
  end

  last_xtrack = xtrack                                                      --
updates the last cross track error
  last_ms = now_ms                                                          --
updates the last time
  return update, UPDATE_MS                                                  --
schedules the next update
end



return update()
```