CPE 2073: Lunar Lander Simulation

Yasir Choudhury
orb234

May 1, 2021

Introduction

In 1959, Robert Noyce ushered in the Silicon Age with the invention of the monolithic Integrated Circuit, or IC. Through sequential combinations of these ICs, possibilities for in-flight adjustment and complex thrust control beyond human capabilities became feasible. The Apollo Guidance Computer was the first to realize these capabilities and successfully control the Apollo Command Module and Apollo Lunar Module throughout its mission in 1969. Weighing just 70 pounds when its predecessors were weighed in tons, it became clear that scaling down in size was just as much a possibility as scaling up in power.[1]

Since then, computational capability available to the average person has exponentially increased. Vacuum tubes shrunk into silicon chips, and computers could be rapidly configured and re-configured through the emergence of software. What was once an incredibly complex task for a team of NASA scientists can now be roughly modelled by an undergraduate college student. The following report walks through the design of a rudimentary guidance system written in the C programming language.

Problem Statement

The designed guidance system includes two modes: one with manual pilot input and an autopilot mode to determine the best values for thrust so that the pilot lands safely every time.

The first mode acts as an open loop control system. The user can input any desired thrust value, observe the change in altitude and velocity in accordance with the thrust, and make a judgement on what value to enter in next.

Alternatively, the user can hand over control to a closed-loop automated system. This autopilot mode determines the best thrust value per game tick based on pre-determined output formulas.

Regardless of the control method chosen, the lunar lander must land safely on the surface of the Moon. Both methods must reach a final velocity between -1 and 0 m/s when the altitude reaches zero.

Design

A data flow graph was constructed to understand the overall flow of information throughout the program. This way, the overall operation of the system can be understood without being lost in the details of how everything works.[2] First, the program prints a welcome message and waits for valid input to determine whether the simulated lander will be controlled manually or automatically. Next, the program indefinitely iterates through a while loop that updates values relevant for space flight, prints these values to the screen, and logs them in to a .csv file for later analysis. Once the height value is equal to zero, the program determines if the final velocity is within the bounds for safe landing. Afterward, the program calls a Python script to display the graphed results of the space flight values to the screen.

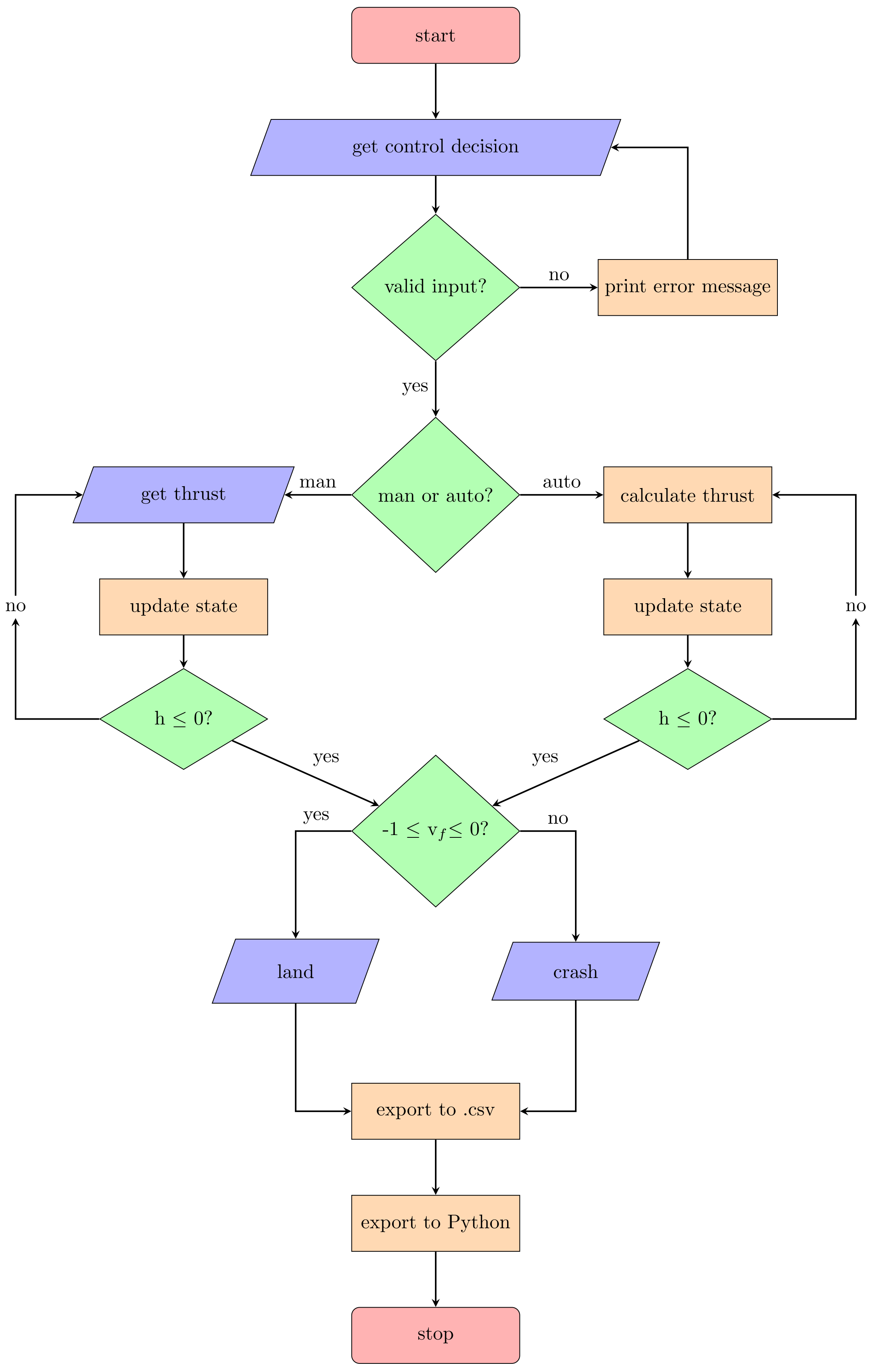

Figure 1: Data flow graph

With the data flow graph constructed and an abstract understanding of the program achieved, further requirements are introduced and refined.

The user or autopilot mode can choose any amount of thrust, up to a maximum of 45 kN. When the user provides a value for thrust, the input is passed to the `maxThrust()` function to verify that it does not exceed 45 kN. `maxThrust()` also verifies that there is enough fuel to service the request for specified thrust. Each kilogram of fuel produces 3000 N of force for one second. For any given thrust value, the fuel burn equation is defined as:

$$Fuel = \frac{T}{3000m/s} kg/s$$

where T is thrust in kN. If there is not enough fuel, the function returns a value that corresponds to how much thrust is remaining based on fuel level. From this point onward, no thrust is returned as there is no fuel to generate it.

The autopilot mode automatically calculates thrust through provided control formulas. Thrust output is defined in phases for higher and lower altitudes.

Values relevant to space flight are created and defined in the `State` struct. These values include altitude, velocity, fuel, and mass. After each iteration of the while loop, these values are updated using the `updateState()` function and the altitude variable is checked to determine how close the lander is to the surface of the moon.

Results

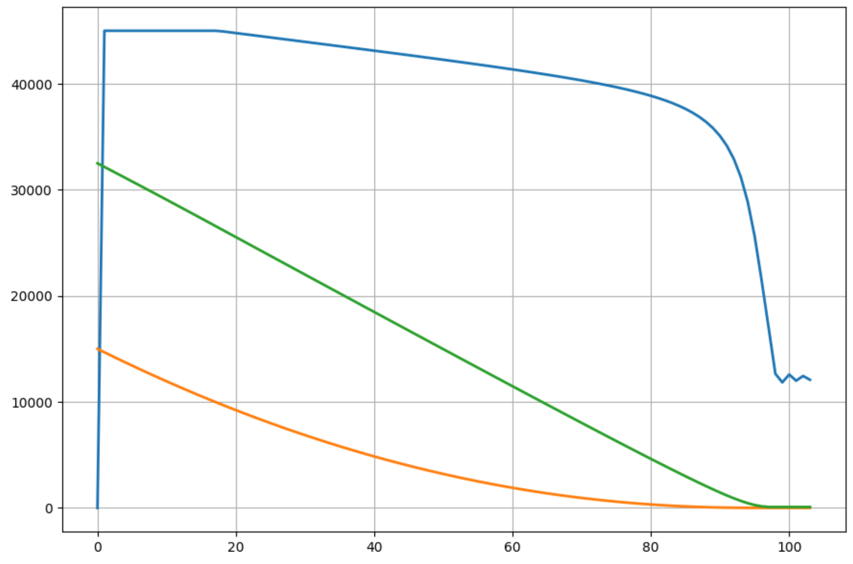

Figure 2: Python output after autopilot landing

Both manual and autopilot modes work as intended for this project. The manual mode enters a while loop, asks for a thrust value, verifies that it is valid when compared to available fuel level, and updates all relevant quantities in the `State` struct. The autopilot mode determines the best thrust value and returns it for every game tick. If the height value is less than or equal to zero, the velocity is read to determine whether the vehicle crashed or landed safely. The output .csv is closed, the Python script is called, and a graph is displayed to the user. Afterward, the program finishes execution.

The graph shown in Figure 2 features lines for thrust, altitude, and velocity. Thrust, marked in blue, starts off at 0 before the autopilot module begins determining an ideal output value. This is reflected in the large jump in thrust at time $t = 0$. The altitude, marked in orange, steadily declines as thrust is applied. As height approaches zero, high levels of refinement are needed if thrust is manually controlled. For purposes of readability, velocity (marked in green) is multiplied by -100. The Y axis approaches 45,000 while velocity starts at -325. Without multiplying by -100, the line for velocity would be nearly indistinguishable from zero.

Challenges

Deconstructing the project into its components initially appeared to be a daunting task. Viewing the entire program as a whole left many questions as to where to begin designing this program. After creating a data flow graph and identifying areas where helper functions could be utilized, systematic decomposition became very straightforward. A new function was written and unit tested each time the project was approached, and it was a short matter of time until the entire project was put together like interlocking building blocks.

At first, the autopilot program was difficult to understand. It was not clear how it was meant to automatically return the ideal level of thrust to slow the descent of the lunar lander. This component of the project benefited the most from systematic decomposition. The function was inspected as a standalone module and analyzed to determine how to design it, line by line. It became clear that the function was meant to return only one thrust value, and the value it would return was calculated with provided equations that factored in the altitude of the lander. Once this function was realized, it was quickly integrated into the program as a whole. Watching the autopilot function work for the first time was deeply satisfying.

Conclusion

Designing this program proved to be quite an exciting task. After overcoming the initial apprehension of understanding the task and configuring its parts, writing each function proved to be very manageable and engaging as well.

This project also helped me understand how to use various pieces of software associated with computer programming. Due to previous experience working in Python and lots of headache with system PATH variables, an existing installation of Visual Studio Code was used instead of the recommended Visual Studio IDE. However, having a goal to accomplish (whether it be a homework assignment, lab, or exam) provided many opportunities to experiment with different editors and systems, like vim and gcc on GNU/Linux. Looking back, I would have liked to learn a little bit more and get direct experience with using version control systems like git or svn.

Overall, this project was an excellent way to take all the concepts learned in this course and use them to make a program that accomplishes a task. While this program is drastically scaled down in complexity when compared to the original Apollo Command Module, it still provides a window into the history of computer programming. Gazing back in to the past is the first step in designing tomorrow's systems.

References

[1] Your Smart Toaster Can't Hold a Candle to the Apollo Computer, *The Atlantic*
Alexis C. Madrigal, 2019
https://www.theatlantic.com/science/archive/2019/07/underappreciated-power-apollo-computer/594121/

[2] Chapter 7: Design and Development, *Introduction to Embedded Systems*
Jonathan Valvano, 2016
http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C7_DesignDevelopment.htm