Flight Data Recorder Simulator

Python in the Enterprise

Artur Kucia

April 8, 2016

Contents

1 Introduction

The main objective of this project was to create an application that would mimic the behavior of a plane's flight data recorder (a black box). Problems we were facing:

- acquiring simulation data
- storing data for later analysis
- providing user interface

The following report describes created program.

2 Application overview

2.1 User Interface

2.1.1 Data Collector

Graphical user interface was created using Tkinter and Matplotlib packages. The program offers two basic functionalities: gathering new data from simulation and displaying archived data from the database.

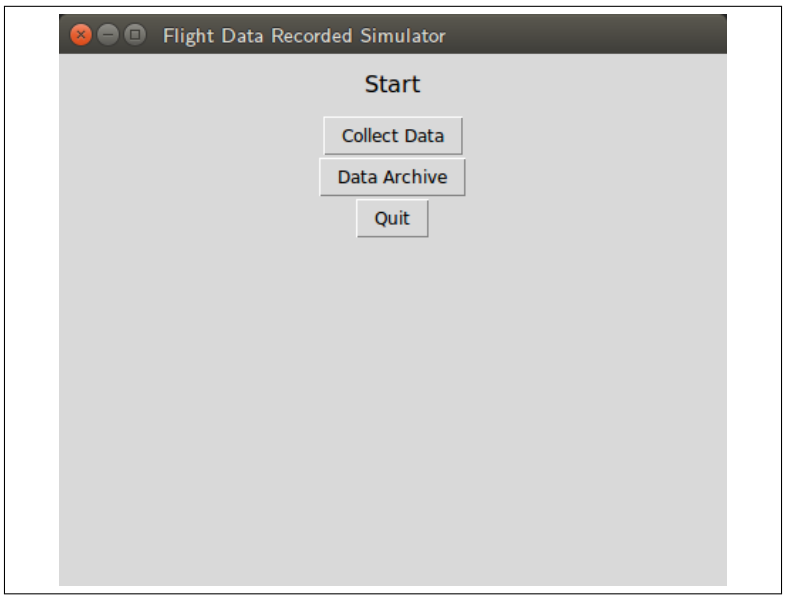

Figure 1: Main Menu of Flight Data Recorder Simulator.

In the Data Collector window user may choose the initial parameters of simulation, such as initial velocities in x, y and z directions as well as type and number of sensors included in flight simulation.
Based on those parameters the program will generate necessary objects and begin the simulation.

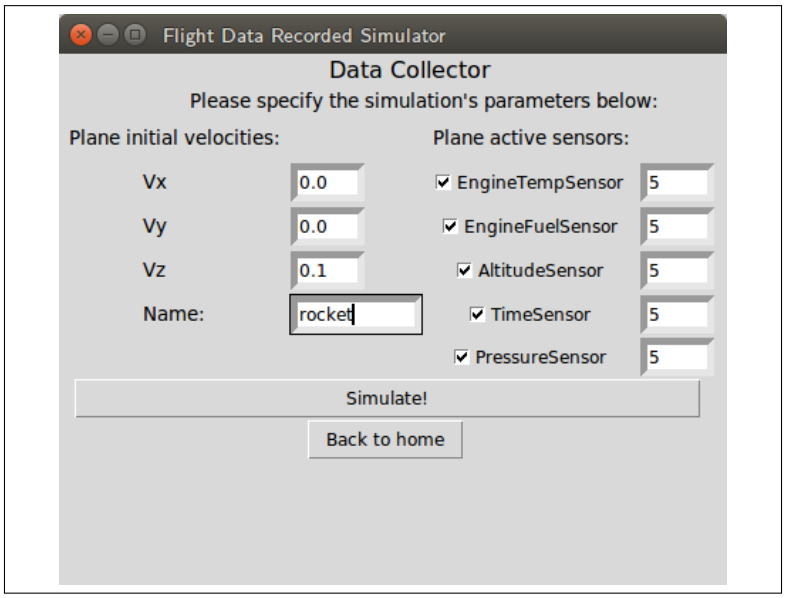

Figure 2: Data Collector window.

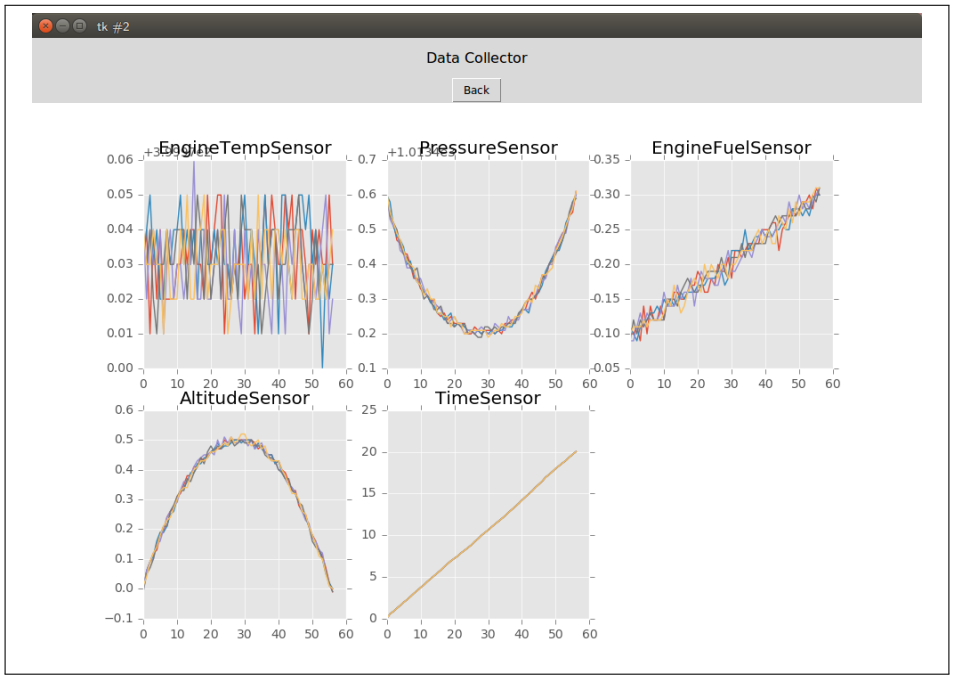

Figure 3: Window with live data read from 5 different types of sensors being plotted. Different colors on a subplot indicate different sensor of the same type.

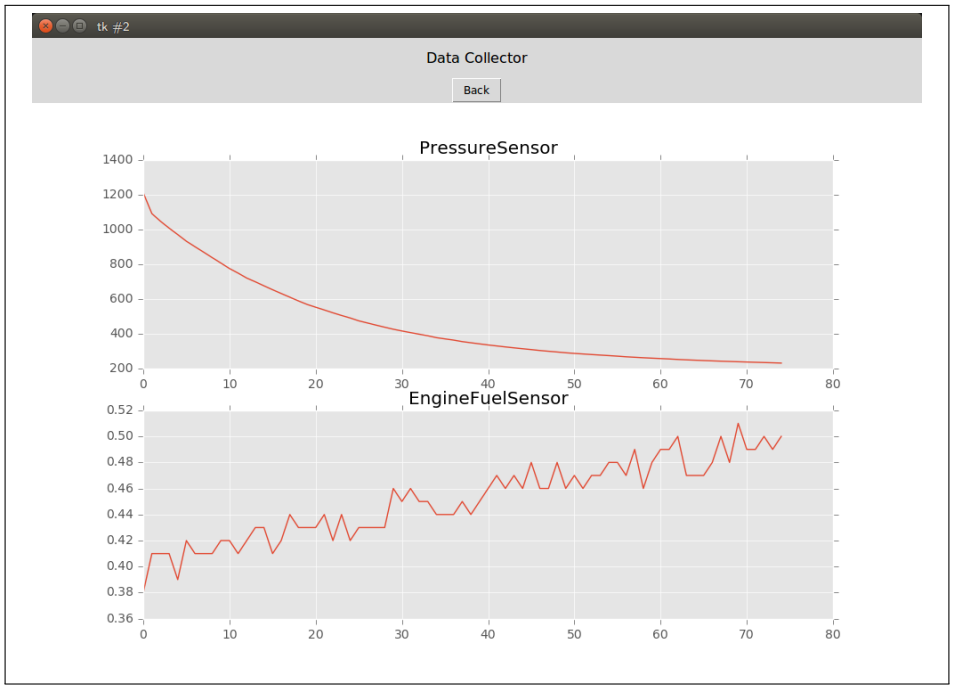

Figure 4: Window with live data read from 2 different types of sensors being plotted.

After the simulation is completed program saves data to a .csv log file.

2.1.2 Data Archive

In the Data Archive page program displays all available historical measurements. Each button corresponds to one previous simulation.

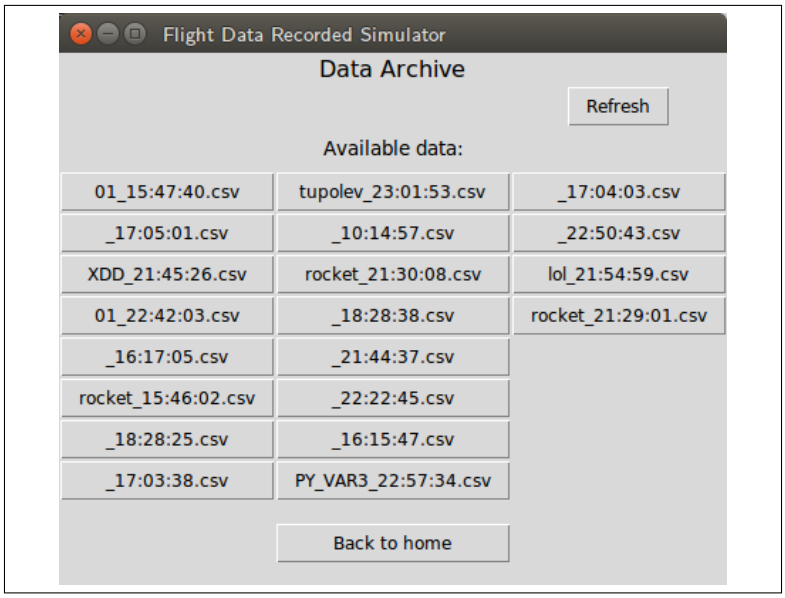

Figure 5: Data Archive window.

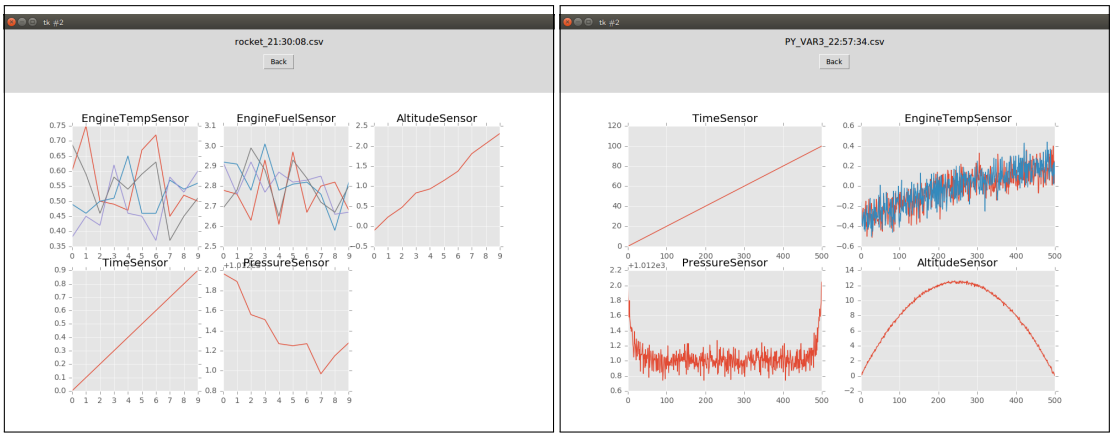

Figure 6: View on two different simulations.

2.2 Simulation of plane

The plane created by the program behaves like a ballistic missile - it is launched in a given direction and falls due to gravitational field. The air resistance was neglected.

Simulation of movement of such "plane" is handled by Plane class. It's most important methods are:

- 'takeoff' - starts the simulation,
- 'update' - calculates current flight parameters and updates internal variables,
- 'isFlying' - used to determine whether the plane is still above the ground or not.

Additionally it has various "get" methods used by the sensors to access flight parameters like position, velocity and so on.

2.3 Simulation of sensors

To simulate behavior of plane's sensors an abstract class Sensors was created, from which specific classes of sensors inherit. Currently available sensors are: 'EngineTempSensor', 'EngineFuelSensor', 'AltitudeSensor', 'TimeSensor' and 'PressureSensor'.
Each sensor type (except TimeSensor) takes data from simulated plane, uses built in mathematical function, adds a random number taken from normal distribution of mean 0 and variance 0.01 and finally returns the sensor's measurement via the 'read' method.
If a program were to take data from physical device, those read methods would have to be modified to access hardware or new sensors classes would have to be written.

2.4 Sensor factory

To make application extendable more easily a simple factory pattern was implemented.

```
class SensorFactory(object):
    """
    class for dynamic creation of Sensors objects,
    factory design pattern
    """
    @staticmethod
    def createSensor(sensorType, numberStr, plane):
        """
        method for dynamic creation of Sensors object, from given type name
        :param sensorType: a string, sensor type name
        :param numberStr: string, number of sensor
        :param plane: a Plane class object,
            used for access to simulation parameters
        :return: Sensor class object
        """
        if sensorType == 'EngineTempSensor':
            return EngineTempSensor(sensorType + numberStr, plane)
        elif sensorType == 'EngineFuelSensor':
            return EngineFuelSensor(sensorType + numberStr, plane)
        elif sensorType == 'AltitudeSensor':
            return AltitudeSensor(sensorType + numberStr, plane)
        elif sensorType == 'TimeSensor':
            return TimeSensor(sensorType + numberStr, plane)
        elif sensorType == 'PressureSensor':
            return PressureSensor(sensorType + numberStr, plane)
        elif sensorType == 'WheelsONOFF':
            return WheelsONOFF(sensorType + numberStr, plane)
```

Figure 7: Code of 'SensorFactory' class.

Besides 'createSensor' method the 'SensorFactory' class also contains a 'createSensorList' method, which will generate the list of Sensor objects given a dictionary with order. It is used for dynamic generation of sensors based on the user's input.

2.5 Storing and retrieving data

The data read from sensors by the 'Buffer' class with a frequency of 10 Hz. They are stored internally as list of lists in two ways:

1. where each sublist (or row) consists of readings from all sensors at a given simulation step

2. where each sublist (or row) consists of all readings of one sensor from entire simulation

The first one is used save the data in '.csv' file (via 'Log' class) and the second is used for plotting in gui.

3 Summary

Simulation of a plane and it sensors turned out to be an easy task, mainly because adopted model was rather basic (a ballistic projectile and data with noise). It didn't require external data input. It was also the first project in which I implemented the design pattern and graphical user interface, which is why they were kept so simple.
Handling data generated by the sensors was a bit harder. Because the the exact number of sensors and their types are not known before the user's decision it was difficult to assign them to correct plots. The developed solution does not have a particularly clear code and is very easy to break. The correct way to handle this matter would be to use a database package, such as SQLlite, which I'm looking forward to use in the next project.