Cookie Factory Virtual Model Documentation: Python Simulation

Introduction

This document is to assist you in modifying/using the python simulation created for the Cookie Factory Virtual Model.

The following documentation includes an overview of how the simulation module should be used, its different files, the different functions/classes in each file, and a description of each of the constant variables and what they represent.

Note that the simulation code is subject to change and as such this documentation is subject to change as well.

# Simulation Overview

## Description

This set of python programs is made to simulate a resource allocation process reminiscent of a real cookie factory. It uses pyModbusTCP to communicate with a virtual programmable logic controller (OpenPLC) and makes changes to simulated pieces of hardware. It is made up of two python files, one being the main runnable, and the other being a file which contains relevant classes which are used multiple times throughout the simulation.

# Files

## main.py

Main.py represents the main runnable of the simulation, it includes the handling of all incoming register data from the programmable logic controller (PLC). As well as the updating of data to be sent back to the PLC. Its functionality includes reducing the amount of material in either silo, calculating the speed by which the vacuum is running, checking the states of the rotary valves, and writing back updated material weights.

### Constants

**AIR\_DENSITY:** set to 0.075 assuming air density at sea level

**MATERIAL\_AIR\_RATIO:** set to 2.0 which is an appropriate ratio for conveying sugar/flour in a dilute phase pneumatic conveying system

**HOPPER\_MAX\_WEIGHT:** set to 400.0, represents the hopper’s max capacity.

**GRAVITY\_CFM\_ESTIMATE:** set to 814, this is the estimate CFM according to the force of gravity of material falling 1 foot from a hopper outlet of 244 square inches.

**SECONDS\_PER\_MIN:** set to 60, 60 seconds in a minute.

**SILO\_WEIGHT:** set to 10000.0, represents both silo’s max capacity.

**SIGNALS:** contains all relevant signal objects being read/sent to OpenPLC.

### Functions

**encode\_ieee:** Uses the **struct** module to convert a 32-bit float into two 16-bit integers to be sent across separate registers. This is done so that we can safely send 32-bit numbers using Modbus TCP.

**RPM:** Uses a frequency read from OpenPLC to calculate our estimated RPM according to the motor we are emulating. This function emulates the job a variable frequency drive would be performing.

**CFM:** Uses frequency read from OpenPLC to calculate our estimated cubic feet per minute (CFM) of material movement. We calculate the CFM using a best fit curve function from our blower datasheet.

**runnable:** This function acts as the primary runnable for the entire simulation. It begins by reading all the relevant signals from OpenPLC. Then it checks if the vacuum is on, if it is it begins siphoning sugar or flour based on if their respective rotary valves are active and which material the diverter valve is set to siphon. After siphoning sugar or flour using our **CFM** function and constants, we then write the updated weights to OpenPLC as well as the current vacuum RPM.

### Classes

**FloatModbusClient:** This class is an extension of pyModbusTCP’s ModbusClient class with the added functionality of being able to write and read 32-bit float numbers using the **encode\_ieee** function. This client is used at the beginning and end of the **runnable** function, and is defined near the beginning of the main.py file.

## components.py

Components.py contains both the signal and container classes which are used multiple times throughout the main.py file.

### Classes

**Container:** This class represents the container objects in the main runnable, such as silos, hoppers, and other future objects that may require a capacity component. The container class contains two functions:

* **set\_weight:** sets the container’s weight to the number passed
* **get\_weight**: returns the container’s weight

**Signal:** This class represents the signal objects in the main runnable, such as the load cells, hopper, vacuum, and rotary valves. The signal class contains three functions:

* **set\_value:** sets the signal’s value to the value passed
* **get\_value:** returns the signal’s value
* **get\_address:** returns the signal’s modbus address