Arcment Program Documentation

Last Updated Mar 15, 2025

Contents

1 Overview

This covers the main components of the Arcment program. The architecture is modular, allowing for easy updates and enhancements. The modules are designed to work together seamlessly, with clear interfaces for data flow and processing. The following is a summary of the high level implementation of each module:

- **Collection Module:** Data streaming and sensor communication.
- **Processors Module:** Pre-processing (G-code parsing and layer segmentation) and post-processing (laser path generation).
- **Sender Module:** Communication with the printer via the DuetWebAPI.
- **Main Module:** Orchestration of the entire printing process.

2 Collection Module

The **Collection** module is responsible for data acquisition and visualization. It consists of the following files:

- `__init__.py`
- `getData.py`
- `oxapi.py`

2.1 `__init__.py`

This file initializes the module by importing all symbols from `getData.py`. In effect, it makes the functions and classes defined in `getData.py` available as part of the `collection` namespace.

2.2 `getData.py`

This script streams data from the laser sensor and visualizes it using `matplotlib`. Its main components are:

- **Stream Initialization:**
  - Creates an instance of the sensor API using `ox("192.168.0.250")`.
  - Obtains a streaming object via `CreateStream()` and starts the stream with `Start()`.
- **Function:** `update(frame)`
  - Checks if there is data available using `stream.GetProfileCount()`.
  - Reads the latest profile data and extracts `x_data` and `y_data` (using indices `-3` and `-2` respectively).
  - Updates the plotted line data and adjusts the x-axis limits.
  - Returns the updated line for animation.

  TODO: Make this more functional, animation just for testing.

2.3 `oxapi.py`

This file defines a large number of functions that wrap calls to the underlying sensor API. These functions are used to configure and query sensor parameters. See oxAPI documentation for more details.

3 Processors Module

The **Processors** module handles the pre- and post-processing of G-code. It is organized into:

- **Top-Level Files:**
  - `preprocessor.py`
  - `postprocessor.py`
- **Submodules:**
  - **PreProcessors** (contains `layer_parser.py` and `processor_interface.py`)
  - **PostProcessors** (contains `laser_path_gcode.py` and `postprocessor_interface.py`)

3.1 `preprocessor.py`

This file defines the `PreProcessor` class which reads, parses, and processes G-code files.

- **Constructor** `__init__(self, gcode):`
  - Opens and reads a G-code file line by line.
  - Initializes a list of predefined sections (e.g., TOP_COMMENT, STARTUP_SCRIPT, GCODE_MOVEMENTS, END_SCRIPT, BOTTOM_COMMENT).
  - Initializes data structures for storing G-code sections and layers.
  - Calls `parse_sections()` to segment the G-code.
- **Method** `parse_sections(self):`
  - Iterates over the G-code lines.
  - Groups lines into sections based on specific end markers (e.g., `;top metadata end`, `;startup script end`, etc.).
  - Stores the grouped lines in the dictionary `gcode_sections`.
- **Method** `run_processors(self, processors=None):`
  - Applies a series of processor objects (conforming to a `ProcessorInterface`) to designated sections of the G-code.

  - Processes sections in the order: STARTUP_SCRIPT, GCODE_MOVEMENTS, END_SCRIPT.
  - Returns the list of processed G-code sections.

- **Method** `parse_layers(self)`:
  - Instantiates a `LayerParser` (defined in the PreProcessors submodule).
  - Processes the GCODE_MOVEMENTS section to extract layers.
  - Returns the list of layers.

3.2 `postprocessor.py`

This file defines the `PostProcessor` class for post-processing the parsed G-code.

- **Constructor** `__init__(self)`:
  - UPDATE WHEN IMPLEMENTED

3.3 PreProcessors Submodule

This submodule refines G-code before it is sent to the printer. It contains:

3.3.1 `__init__.py`

Simply imports all contents from within the submodule.

3.3.2 `layer_parser.py`

Defines the `LayerParser` class which implements the `ProcessorInterface`:

- **Constructor** `__init__(self)`: Initializes an empty list to store layers.
- **Method** `process(self, gcode: list\[str\]) → list\[str\]`:
  - Iterates over G-code lines and detects layer boundaries using the marker defined by `Sections.CURA_LAYER`.
  - Groups lines into layers.
  - Returns the list of layers.
- **Method** `type(self)`: Returns the processor type (`Sections.GCODE_MOVEMENTS_SECTION`).

3.3.3 `processor_interface.py`

Defines the interface for pre-processors:

- **Class `Sections`:** Contains constants representing various G-code sections and markers.
- **Abstract Class `ProcessorInterface`:**
  - `process(gcode: str) → str`: Method to process a given G-code segment.
  - `process_type()`: Method to return the type of G-code section the processor handles.

3.4 PostProcessors Submodule

This submodule contains all the processors that are applied to G-code layers DURING the print.
TODO: UPDATE WHEN IMPLEMENTED

4 Sender Module

The file `sender.py` defines the `Sender` class which is responsible for sending processed G-code to a 3D printer. It uses the `DuetWebAPI` for communication. Key functions include:

- **Constructor** `__init__(self):`
  - Sets a hard-coded printer IP (`"169.254.1.2"`).
  - Creates a printer connection using `DuetWebAPI` and connects with the password `'reprap'`.
- **Method** `send_code_line(self, code_line):`
  - Sends a single G-code line to the printer.
  - Prints a confirmation message.
  - Waits (polling every 0.25 seconds) until the printer status becomes idle before proceeding.
- **Method** `send_layer(self, layer):`
  - Accepts a layer either as a string or as a list of G-code lines.
  - Splits the layer if provided as a string.
  - Iterates over each line in the layer and sends it via `send_code_line()`.
- **Method** `get_status(self):` Retrieves the printer's status by querying the DuetWebAPI.
- **Method** `get_current_position(self):` Retrieves the current machine position.

5 Main Module

The **main** module (`main.py`) orchestrates the overall workflow of the program:

- **Class `Main`:**
  - **Constructor `__init__(self, gcode_file)`:**
    - Accepts a G-code file (though internally assigns `"test.gcode"` to `self.gcode_file`).
    - Instantiates a `PreProcessor` using the G-code file.
    - Instantiates a `PostProcessor`.
    - Instantiates a `Sender`.
    - Parses layers from the preprocessor and initializes layer tracking variables (`current_layer` and `total_layers`).
  - **Method `run(self)`:**
    - Processes and sends a single layer.
    - Checks if the current layer exceeds the total number of layers.
    - Uses `Sender.send_layer()` to send the current layer.
    - After sending, increments the layer counter.
    - If there are remaining layers, it invokes `PostProcessor.gen_next_layer()` to prepare the next layer.
    - Returns a Boolean value: `True` if all layers have been sent, `False` otherwise.
  - **Method `run_all(self)`:**
    - Iteratively calls `run()` in a loop until all layers are processed.
    - Uses a sleep interval of 0.5 seconds between iterations.