Cloud Tracking V2
Code Quick Reference

Sponsor

Orlando Utilities Commission (OUC)

Justin Kramer

Rubin York

Project Supervisor

Dr. Mark Steiner

Design Team

William Askew

Clay DiGiorgio

Lars Gustaf Jiborn

Mohab Kellini

Nicholas Moser-Mancewicz

Timothy O’Brien

Gregory Ratz

Justin Zabel

Code Quick Reference

Our project is split into Jetson code and Website code. The Website is written in JavaScript using a MERN stack and mostly just displays data in the database generated by the Jetson. The Jetson code is written in python and does all the cool stuff like predicting power, pulling weather data from OUC's sensors, etc. It also pushes all the data it reads and generates to the MongoDB database.

Jetson

1. Livestream
    a. Be sure to install all requirements including ffmpeg as described in the user setup guide.
    b. To stream from the camera to the website log into the camera, and send the livestream to an active substation on the website.
    c. Bash this file and replace the 1 at the end of line 11 with sub-{stationID}
    EX: .../sub-27
    d. ~/startup_scripts/run_stream_ffmpeg.sh
        i. Opens an ffmpeg stream, streaming from the camera input to the website's backend
2. ~/config
    a. ~/config/substation_info.py
        i. Constants related to a specific substation. Each Jetson will have its own substation_info.py.
    b. ~/cloud_tracking_config.py
        i. Settings for the cloud tracking code. Includes URL to which the socket io is being set up with.
3. ~/datalogger
    a. ~/datalogger/cloud_height_data.py, ~/datalogger/weather_data.py
        i. Classes used to store relevant data. No logic in them besides some mathematical calculations related to calculating dew point and cloud base height.
    b. ~/datalogger/datalogger.py
        i. Datalogger interaction logic. The constructor instantiates a connection using minimalmodbus and the defined parameters. The parameters need to match those of the datalogger. If using a Campbell Scientific CR300 datalogger, they should not need to be modified.
        ii. Poll queries the datalogger for the most up to date data and returns it in a WeatherData object. Minimalmodbus documentation on the read_float method can be found here.
4. Cloud Tracking
    a. ~/app.py

        i. Establishes connection to camera via ffmpeg
        ii. Extracts frames from stream and processes them before sending them via socket.io to website
        iii. Calculates cloud coverage percentage based on image processing output and sends percentage to database
    b. ~/imageProcessing
        i. ~/imageProcessing/coverage.py
            1. Takes as input an image, filters out all non-cloud pixels and returns a PNG consisting solely of cloud pixels.
        ii. ~/imageProcessing/fisheye_mask.py
            1. To account for the non-lens pixels surrounding the circular camera feed create_fisheye_mask takes as input an image, and colors all non-lens pixels black
            2. To account for fisheye lens distortion image_crop crops out the distorted pixels at the edge of the camera lens.
    c. ~/opticalFlow
        i. ~/opticalFlow/opticalDense.py
            1. Applies the gunnar-farneback optical flow algorithm to create a vector field showing the predicted motion of the cloud. And draws said vector arrows onto the cloud PNG image from coverage.py

5. Power Verification
    a. ~/datalogger_runner.py
        i. Queries the datalogger every 60 seconds for updated weather data, calculates cloud base height, and sends them to the database, that's it.
        ii. The init method attempts to establish a connection with the datalogger over the given path. For example, over /dev/ttyUSB0. The calculation in time.sleep() ensures that the thread sleeps 60 seconds from the time the thread was started, not when it finishes. This ensures there is no stray since there is some amount of processing time needed for the thread to complete.
        iii. In init, self.client creates a pymongo client. self.client.cloudTrackingData specifies the database name. Here, cloudTrackingData is the database name.
        iv. self.db.WeatherData_Only is how you specify which collection to insert into in the database. WeatherData_Only is the name of the collection. If it doesn't exist yet, it will automatically be created. The same concept is used when sending cloud base height to the database.
    b. ~/power_verification.py
        i. Runs power prediction and verifies the predicted power amount to the observed power from the weather sensor.
        ii. format_current_weather_data takes the relevant data from the datalogger and stores it in a list for use by the power prediction methods.
        iii. add_current_data adds the current minute's weather data to the front of the list and removes the latest minute's weather data from the end of the

list if the amount of previous data needed has been meant. For example, if the ML models have been trained on 15 minutes of data and there is 10 minutes worth of data in the weather_data_list, no data will be removed from the end. If there is 15 minutes worth of data in the weather_data_list, then weather_data_list[14] will be removed.

    iv. VerifiedPowerData is a class to store information related to power verification, no logic.
    v. The init method creates a mongoDB client connection to the cloudTrackingData database and establishes a connection with the datalogger over the given port ('/dev/ttyUSB0').
    vi. The run method is the main loop that contains the logic of the script. It is run every 60 seconds.
    vii. The methods included in the Train module are toTimeSeries and makePrediction. toTimeSeries formats the weather data into the format used by the prediction method makePrediction.
    viii. The various send_xxx_data_to_db simply creates an object and sends it to the specified collection of the database. The collections are CloudHeightData, PowerPredictionData, WeatherData, and PowerVerificationData.
    ix. calculate_cloud_height calculates cloud height based on provided weather data.
    x. run_verification runs power verification. There are more verbose comments in the code explaining how this method works.
    xi. The Get_Data_On_Startup thread runs for the same number of minutes as is being predicted. For example, if power is being predicted 15 minutes into the future, then the thread will run for 15 minutes to gather data. It only needs to be run when the Jetson is restarted or if there has been an extended period of time since the last weather data collection. It collects weather data, adds it to the weather_data_list, and sends it to the database.

6. ~/power_prediction
    a. ~/power_prediction/Train.py
        i. Trains new model, saves model and weights in models/weights folders named after time training started
        ii. Hyperparameters like steps and epochs near top of file, useful to change runtitle that is displayed on training vs validation loss graph output after training
        iii. The first two lines after the import statements are for GPU accelerated training, comment them out if the script doesn't work. GPU training requires installation of Nvidia drivers, cuda, and libcudnn. If using a linux machine to train, look into using conda for easy installation.
        iv. File loads the data from csv, organizes it and feeds into NN.
        v. After training, the model name is printed out, save that name to be loaded in the predict script and put it to use

        vi.
    b. ~/power_prediction/Predict.py
        i. Loads the trained model and uses it to predict power from weather data
        ii. Data must first be formatted with toTimeSeries(data, timesteps, batches, start) with a number of timesteps that match the trained model, currently that number is 15
            1. This also means that at least 15 minutes of data are needed to start making predictions.
        iii. Prediction automatically scales the data to between 0 and 1, so network can use it, and upscales after

Website

Note: in the website code, we use the terms “substation”, “station”, and “sub” interchangeably. We mean these terms to refer to a single location with a single jetson.

Note: “~/” refers to the root directory of the website code. For example, if you put the website code at the path “/home/users/adminUser/” on your server, then “~/Front_End/src/App.js” would refer to the full path “/home/users/adminUser/oucseniordesignv2/Front_End/src/App.js”

Table of Contents

Overview

1. **Home Page**
    a. ~/Front_End/src/components/CombinedMap.js
        i. Hosts Full Page google Map
        ii. Sets up Markers for each substation on the system
        iii. On click, navigates user to the proper substationHomePage corresponding to the clicked marker
    b. ~/Front_End/src/App.js
        i. Hosts CombinedMap component
        ii. Sets up the URLs Navigation.js will help the user navigate to
2. **Weather Data Display / Archive**
    a. Made of two parts
    b. API
        i. ~/server.js
            1. require(‘cors’); // allows us to make requests from frontend

        2. init_routes();   // sets up the bas URL for weatherDataRouter.js to expand on
    ii. ~/api/models/weatherDataModel.js
        1. Lays out the format of a single entry to the Weather Data collection (mongoDB version of a table)
    iii. ~/api/controllers/weatherDataController.js
        1. Defines functions for pulling from the database
        2. ie GetAll() pulls all data, GetTargeted() pulls data constrained by certain parameters, etc.
    iv. ~/api/routes/weatherDataRouter.js
        1. Sets up urls to visit that will send you the results of the functions specified above
        2. eg visiting "https://localhost:3000/weatherData/getAll" will cause the backend to call GetAll() and then send you the results via HTTP
  c. Frontend
    i. ~/Front_End/src/components/apiCallers/RetrieveTargetedWeatherData.js
        a. Decides what data to pull from the backend (aka the API)
    ii. There are then two options for rendering the data pulled by RetrieveTargetedWeatherData
        1. ~/Front_End/src/components/miniComponents/DisplayWeatherData.js
            a. Displays all weather data pulled by RetrieveTargetedWeatherData in a table
        2. ~/Front_End/src/components/miniComponents/DisplayWeatherDataFriendly.js
            a. Displays select data points from the data pulled by RetrieveTargetedWeatherData in a more readable, user-friendly format
            b. Also displays cloud coverage percentage data, which isn't pulled by RetrieveTargetedWeatherData
    iii. These components are hosted in ~/Front_End/src/components/SubstationHomepage.js
    iv. And alternatively in ~/Front_End/src/components/Archive.js

3. **Power Prediction Display**
  a. Made of two parts
  b. API:
    i. ~/server.js
        1. require('cors'); // allows us to make requests from frontend
        2. init_routes();   // sets up the base URL for powerPredictionsRouter.js to expand on
    ii. ~/api/models/officialPowerPredictionsModel.js

            1. Lays out the format of a single entry to the Power Predictions collection (mongoDB version of a table)
        iii. ~/api/models/officialPowerPredictionValidations.js
            1. Lays out the format of a single entry to the Power Prediction Validations collection (mongoDB version of a table)
        iv. ~/api/controllers/officialPowerPredictionsController.js
            1. Defines functions for pulling from the database
        v. ~/api/routes/powerPredictionsRouter.js
            1. Sets up urls to visit that will send you the results of the functions specified above
            2. eg visiting “https://localhost:3000/powerPredictions/getAll” will cause the backend to call GetAll() and then send you the results via HTTP
    c. Frontend
        i. ~/Front_End/src/components/apiCallers/OfficialPredictionsLineGraph.js
            1. Retrieves data by calling the API
            2. Formats and displays the line graph
        ii. ~/Front_End/src/components/PowerPredictionsDashboard.js
            1. Hosts the OfficialPowerPredictionsLineGraph.js component
        iii. ~/Front_End/src/components/SubstationHomepage.js
            1. Hosts the Power Predictions Dashboard

4. **Livestream**
    a. Backend
        i. ~/server.js
            1. Create Channels for the four substations in server.js (initChannels()), and names them according to the station’s ID EX: sub-27
            2. These channels are waiting for ffmpeg connections from the substation jetsons.
        ii. ~/api/routes/livestreamRoutes.js
            1. Parses url from “baseurl”/cloudtrackinglivestream/<sub station id> then pushes that livestream data to front end component looking for the specific <sub station id>
    b. Frontend
        i. ~/Front_End/src/components/SubstationLivestream.js
            1. Connect to appropriate livestream ffmpeg channel, respecting stationID prop
            2. Displays livestream
        ii. ~/Front_End/src/components/SubstationHomepage.js
            1. Hosts Livestream component
5. **Cloud-Shadow Leafly Map**
    a. Backend
        i. ~/api.js
            1. Contains and exports all subscribing functions

        ii. ~/server.js
            1. Subscribes to cloud coverage and shadow data from jetson using socketio based on substationID
            2. EX: subscribeToCoverage27, subscribeToShadow27
            3. Calls api to pull local weather statistics (see **Weather Data Display->API**)
    b. Frontend
        i. ~/Front_End/src/components/Map.js
            1. Imports subscribe function from ~/api.js
            2. Leafly Map component
            3. Using weather statistics and data from the socketio subscriptions, creates two separate overlays, one for clouds and one for their shadows.

6. **Substation Home Pages**
    a. ~/Front_End/src/components/Navigation.js
        i. Each homepage uses the same component, that component pulls information from the url to determine which station it's supposed to render
        ii. url example: "http://localhost:3000/Sub/29"
    b. ~/Front_End/src/components/SubstationHomepageWrapper.js
        i. pulls the number at the end of the url (29 in the above example) and passes it to the real SubstationHomepage component
    c. ~/Front_End/src/components/SubstationHomepage.js
        i. The real substation homepage component.
        ii. Responsible for hosting all the components that display the data we want to see (eg the livestream component, the weather data component, etc) and telling them which substation to display data for.
7. **Styling**
    a. ~/Front_End/src/stylesheets/SubstationHomepage.css
        i. SubstationHomePage Styling is all done here.
8. **Navigation**
    a. ~/Front_End/src/components/Navigation.js
        i. Navigation Bar across top of page
    b. ~/Front_End/src/App.js
        i. Hosts Navigation.js
        ii. Sets up the URLs Navigation.js will help the user navigate to