deepsea

DEEPSEA DATA PROCESSING PROJECT – CHRIS BERRY

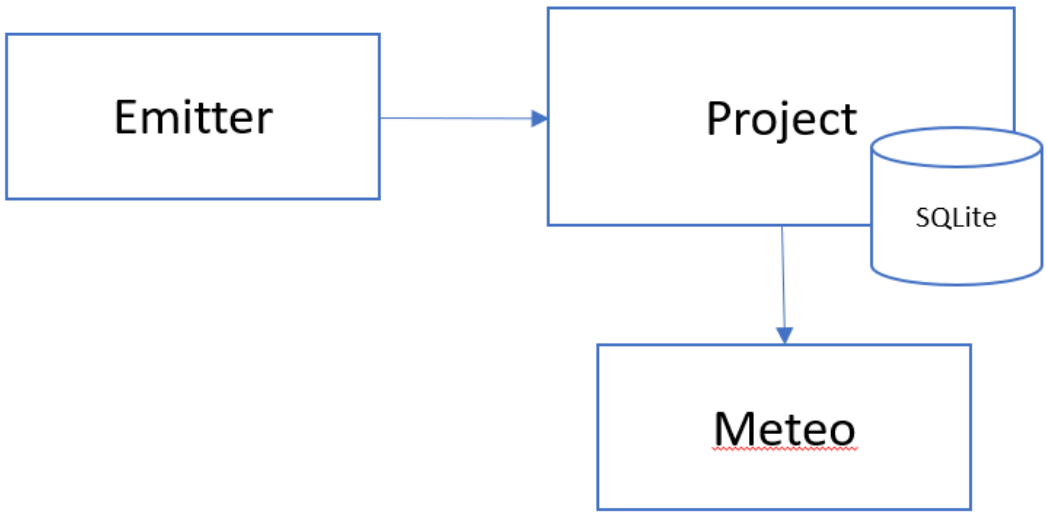
SYNOPSIS

The purpose of the project is to setup an environment to emulate a vessel emitting sensor data a to server, which normalizes and graphs the data.

OVERVIEW

The environment is configured with Docker Compose. There are 3 containers which represent the following elements:

- **Emitter** – This container emulates a vessel sending sensor info to **Project**.
- **Meteo** – This container emulates a meteorological endpoint that can be queried for sea currents data. The endpoint updates every 10 minutes.
- **Project** – This container is the main component which processes the data. It receives sensor data from **Emitter**, queries **Meteo** for sea current. It runs the frontend for graphing and backend, both in **Django**. It hosts the DB as a **SQLite** file.



deepsea

SETUP

Install the project:

```
git clone git@github.com:ChristopherDBerry/Deepsea.git

cd Deepsea

docker compose up –build
```

Visit the site:

http://localhost:8000/

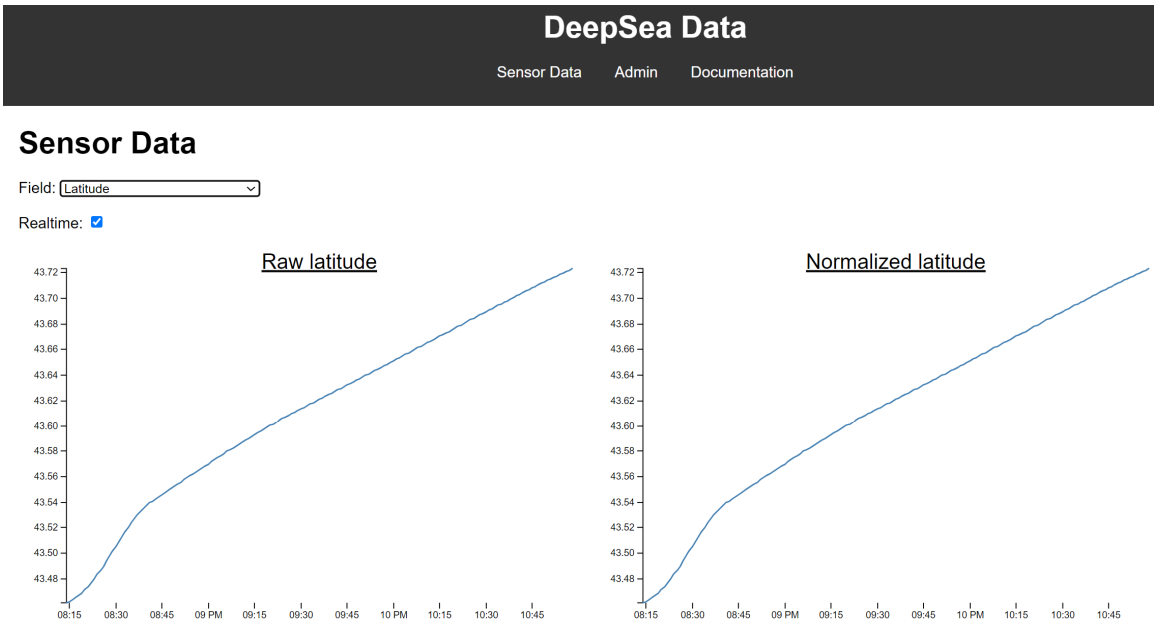
http://localhost:8000/admin/ (root / root)

http://localhost:8000/api/docs/

Start the emitter:

```
docker compose run --rm emitter sh -c "python emitter.py -f DBdataset.csv -d 100"
```

The graphs should now start updating in realtime:



To reset sensor data go to ‘Admin’ -> ‘Reset’

deepsea

DATA PROCESSING

There are 3 forms of processing: fix missing data, fix outlier data, fetch extra data (see currents). All fields all processed identically.

To fix missing data I have implemented a prediction model based on linear regression using the **sklearn** python module. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

To detect outliers I used a modified Bollinger Bands algorithm (https://en.wikipedia.org/wiki/Bollinger_Bands). I modified the bands to be wider following missing data, since we are would be confident an outlier is erroneous. If an outlier is detected it is replaced using the linear regression function.

To lookup sea current the **meteo** container implements the endpoint

```
http://meteo:5000/get_currents_data?latitude={latitude}&longitude={longitude}
```

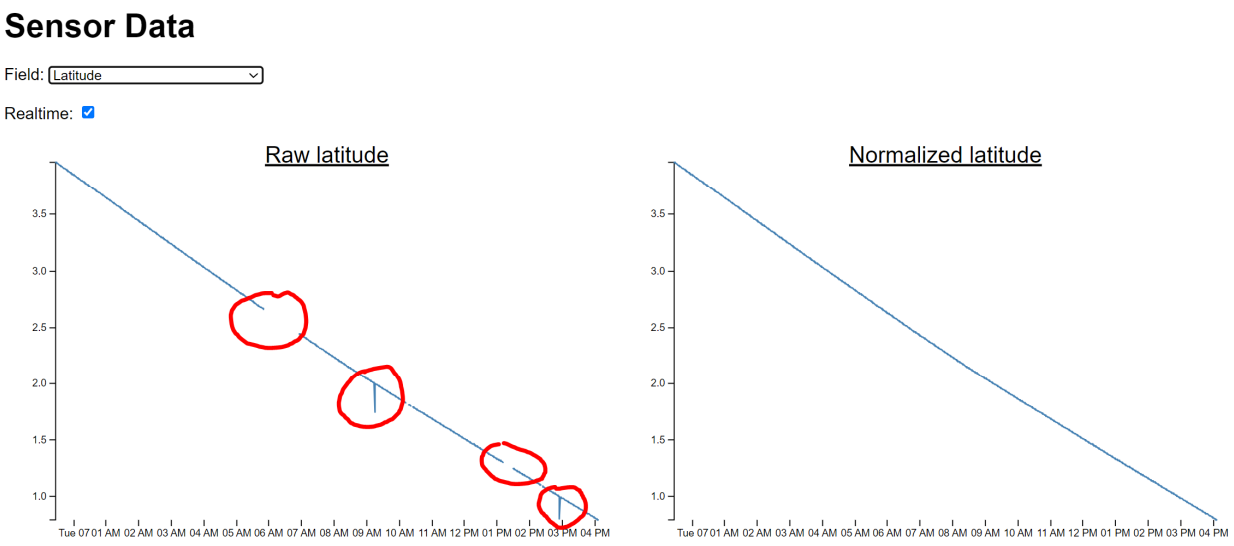
This returns a json file with randomized values in the format:

```
{"sea_currents_angle": 1.01, "sea_currents_speed": 1.09}
```

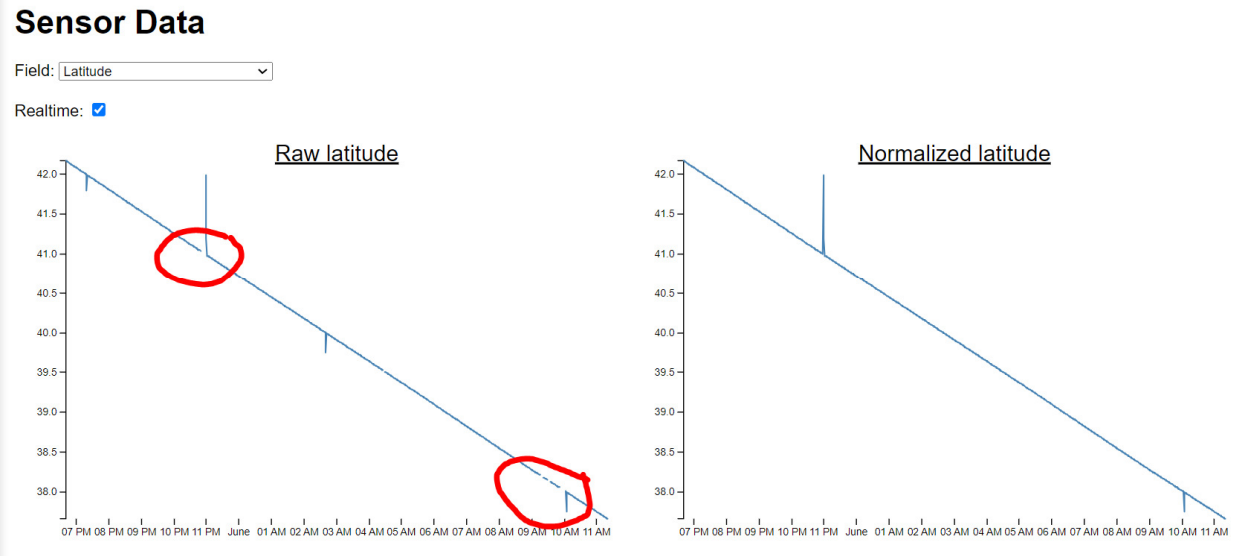
Project checks if the last sea currents update is stale (older than 10 minutes), if so it checks the endpoint. If not it uses the linear regression function to estimate sea currents.

RESULTS

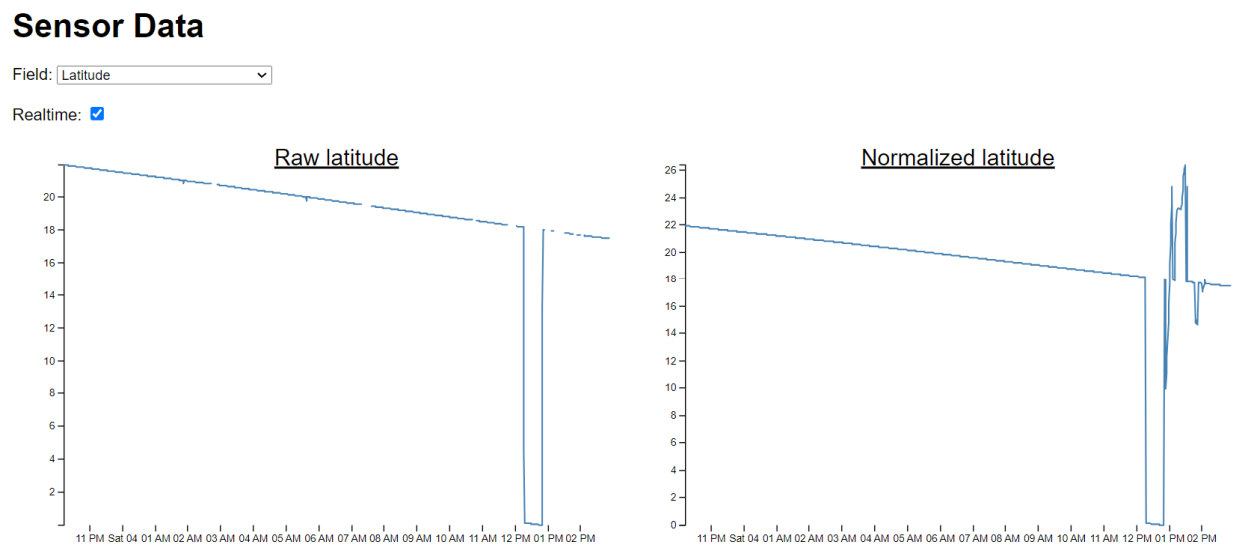
The processing does a reasonable job of detecting and fixing errors:



However, there are some issues. Outliers which immediately follow gaps in data can be overlooked:



Also, it is very difficult to fix gaps in a long series of data, and correction can over-compensate in the opposite direction:



ANALYTICAL DESCRIPTION AND LIMITATIONS

Django Rest Framework (DRF) was chosen to implement the webserver since it has very good tools to quickly setup a website and API endpoints. In this implementation it would be possible for the emitter to detect connection failures and attempt a resend. An alternative approach for real time data is to send the data via UDP to save bandwidth.

This implementation only supports a single vessel. A more realistic implementation would implement client users, and each client could have multiple vessels.

The implementation does not support any kind of authentication. Token based authentication could be implemented.

Timezones are not implemented, all timezone data is assumed to be UTC.

Data processing could be improved. Correction could be done on a per field basis, it could be possible to derive a missing field for other values, for example if we know velocity and location, we can reasonably predict the next location.