## GitHub Repository

The file and directory structure of the repository is:

```
├── README.md
├── Demo.ipynb
├── roi.geojson
├── build
│   ├── bashrc
│   ├── buildimage.sh
│   ├── createcontainer.sh
│   ├── createnetwork.sh
│   ├── data_shell_region3db.sh
│   ├── Dockerfile
│   ├── environment.yml
│   └── root_shell_region3db.sh
├── Docs
│   ├── Outline.pdf
│   ├── Overview.pdf
│   ├── Installation.pdf
│   ├── GitHubRepository.pdf
│   ├── Processing.pdf
│   ├── Results.pdf
│   ├── References.pdf
│   └── Demo.pdf
├── checkIngestedFort64.py
├── checkIngestedSwan63.py
├── createClusters.py
├── createGeoClusterTable.py
├── createGeoNcCSV.py
├── createStormClusterTable.py
├── ingestGeoCSV.py
├── ingestSwanAllNcCSV.py
├── ingestVelFortNcCSV.py
├── ingestZetaFortNcCSV.py
├── region3-function-bufopt.sql
├── region3-function.sql
└── storm_list.txt
```

### README

The README.md file contains an explanation of how to build the adcircreg3simdb docker image, create the container, and run the ingest scripts.

**Demo**

The Demo.ipynb file is a Jupyter Notebook that demonstrates how to access the database using a vector tile server (pg_tileserv), extract tiles, using a region of interest (roi.geojson) file, stitch them together and display them as an image. Further discussion on how to run this demo is in the Results section.

**Build directory**

The build directory contains the docker file and shell scripts used to build the docker images, and create the network and container. These files were discussed in the Ingestion section of the documentation. The build directory also contains the bashrc file for the container, the environment.yml file for the creation of the conda environment, and two shell scripts to access the container.

The bashrc file has the settings required to run conda environments. When creating the docker image, with the Dockerfile, the bashrc file is moved to the root directory of the image as a .bashrc. This enables users who are accessing the container to run the conda command:

```
conda activate adcirc
```

which activates the “adcirc” virtual environment necessary for running the scripts. This environment was created when the docker image was created, using the Dockerfile. In the Dockerfile, the environment.yml file is copied from the repository to the image, and the conda environment is created using the following command:

```
RUN conda env create -f environment.yml
```

The environment.yml yaml file contains meta-data for the creation of a conda environment named “adcirc” and the installation of software necessary for the processing of the data to ingest into the database. The contents of the environment.yml file are:

```
name: adcirc
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.7
  - netcdf4
  - xarray
  - pip:
      - psycopg2-binary
      - wget
```

The “adcirc” conda environment uses python 3.7. It installs netcdf4, and xarray for processing the netcdf data files downloaded from the RENCI thredds server, where the ADCIRC Region III Simulation data is archived. The netcdf files are downloaded using wget. The python scripts that ingest the data, into the database, use psycopg2 to access the Postgres database.

The build directory contains to shell scripts for accessing the container:

root_shell_region3db.sh, for accessing the containers root account, and
data_shell_region3db.sh, for accessing the containers data account.

The use of these scripts is discssed in the Installation section of the document.

**Python scripts**

The root directory of the adcircreg3simdb repository contains a README.md file, that explains how to install and use the software, ten python scripts, two sql scripts, and a list of all storm tracks. There are four groups of python scripts, based on their use.

**Python scripts for ingesting ADCIRC variables**
ingestZetaFortNcCSV.py, used for ingesting Fort63 zeta variable.
ingestSwanAllNcCSV.py, used for ingesting SWAN wave variables.
ingestVelFortNcCSV.py used for ingesting Fort64 Velocity variables.

**Python scripts for ingesting geometry and bathymetry**
createGeoFortZipCSV.py used for extracting geometry (lat, lon), and bathymetry from a storm track netcdf file.
ingestGeoFortCSV.py used for ingesting geometry (lat, lon), and bathymetry.

**Python scripts for checking ingested data**
checkIngestedFort64.py used to produce statistics on fort64 velocity variables from both the netcdf file, and table in reg3sim database.
checkIngestedSwan63.py used to produce statistics on swan63 velocity variables from both the netcdf file, and table in reg3sim database.

**Python script for clustering data, using DBSCAN**
createClusters.py used to create clusters using DBSCAN
createGeoClusterTable.py used to cluster geometry and bathymetry, using clusters created by createClusters.py.
createStormClusterTable.py used to cluster ADCIRC variables, using clusters created by createClusters.py.

**A closer look at the Python ingest scripts**

Each of the ingest scripts has three functions, getRegion3NetCDF4, createtable, and ingestData.

The getRegion3NetCDF4 function uses wget to download netcdf file(s) from the RENCI THREDDS server. In the case of Fort63 Zeta variable, and Fort64 velocity variables one netcdf file is downloaded, fort.63_mod.nc, and fort.64.nc, respectively. In the case of Fort63 Swan variables three netcdf files are downloaded, one for each variable, swan_HS.63_mod.nc, swan_TPS.63_mod.nc, and swan_DIR.63_mod.nc.

The createtable function creates a table in the reg3sim database. It takes as input the storm track name, being ingested, and a time interval for creating the hypertable. The time interval determines the chunk sizes in the hypertable, and currently 2 hours is being used. The Region III Simulation data being ingested has a timestep of 30 minutes, so a time interval of 2 hours means each chunk contains 4 timesteps. The createtable function also creates two composite indexes for the table, using btree indexing. The variables used in the first composite index are node and timestamps, while the second index reverses the same variables. These indexes were created for use in the plgsql functions that a vector tile server, such as pg_tileserv, could access. The optimize querying the node and timestamp. Other indexes optimized for querying variables (zeta, hs, tps, dir, u_vel, v_vel) could be created. Form more information on indexing hypertables see TimescaleDB blog "How to use composite indexes to speed up time-series queries", and TimescaleDB Table Management documentation.

The ingestData function reads the netcdf file(s), downloaded by the getRegion3NetCDF4 function, extracts the data to a csv file, and inputs it into the table created by the createtable function. After reading the netcdf file, the function loops through each timestamp, extracting the variable(s) and outputting them to a temporary csv file. The csv file contains the node values, timestamps values, and variable(s) values, for a specific timestamp from the netcdf file. The csv file is copied to the table in the reg3sim database using timescaledb-parallel-copy, which is a GO program that enables parallel input of data. The number of CPU's used in copying the data can be set. The function currently uses 4 CPU's. After the data is copied to the database, the csv file is deleted, and the next csv file is created. The function also writes the time it takes to ingest each timestamp, to a csv file that is stored in the /home/data/ingesProcessing/ingest directory. This file can be used to check the progress of the ingest.

The y (latitude), x (longitude), and depth (bathymetry) data is input into a separate table. This is done to save space, because y, x, and depth do not vary between timestamps or storm tracks netcdf files, therefore only one table is needed to store them. Two scripts, createGeoNcCSV.py, and ingestGeoCSV.py are used to ingest the geometry and bathymetry data. The script createGeoNcCSV.py has one function, createCSVFile, which extracts the y (latitude), x (longitude), and depth (bathymetry) variables, from one of the storm tracks netcdf files, and outputs them to a csv file, along with the node values. The script ingestGeoCSV.py has one function, ingestGeo, which creates the r3sim_fort_geom table, in the reg3sim database, reads the csv file, created by ingestGeoCSV.p, and copies the data to the r3sim_for_geom table. The function also creates a PostGIS geometry (geom) column from the latitude and longitude

values, and indexes the geom column using SPGIST. This type of indexing repeatedly divides the search space into partitions that need not be of equal size. This type of partitioning works well with the nodes from the ADCIRC unstructured grid.

**Check ingest scripts**

Currently there are two scripts, checkIngestedSwan63.py and checkIngestedFort64.py which produce statistics, per timestamp, from both the netcdf file and reg3sim storm track tables, for the swan63 variables, and fort64 variable, respectively. The purpose of these statistics is to have a record of the data in the netcdf file, and reg3sim storm track table. This record can be useful in tracking problems in the ingest of the data.

**Create clusters scripts**

There are three experimental scripts createClusters.py, createGeoClusterTable.py, and createStormClusterTable.py for clustering the original grid nodes, using PostGIS ST_ClusterDBSCAN function. The purpose of these scripts was to create tables with fewer nodes for accessing larger areas, at lower zoom levels, using a vector tile server, such as pg_tileserv.

The DBSCAN algorithm spatially clusters points based on distance density. The ST_ClusterDBSCAN function takes a distance in meters as input, which is used to cluster nodes that have a distance from each other equal to or less than the input distance. The nodes in the ADCIRC unstructured grid are unevenly distributed, with areas of interest, such as the Chesapeake Bay having a much higher density of nodes than other areas, such as the Gulf of Mexico which is outside of FEMA's region III. This high density of nodes can become a problem when querying tiles that cover large areas at low zoom levels. The number of higher density nodes can be reduced by clustering them using the DBSCAN algorithm.

The first script createClusters.py uses the ST_ClusterDBSCAN function to cluster the nodes in the r3sim_fort_geom table geom column, based on the distance between nodes. The results are input into a new column containing numbers which represent the clusters. The next two scripts use those numbers to produce to calculate the centroid of each cluster, in that case of createGeoClusterTable.py, or to calculate the mean of the variable values of each cluster, in the case of createStormClusterTable.py. The results are input into new tables.

**PLPGSQL functions**

Currently there are two PLPGSQL functions region3-function.sql and region3-function-bufopt.sql. These functions are very similar, however, the later has buffer size as an input parameter, and the former does not. The buffer size is used to set the extra data buffer for a tile, the default is 256 units, which is used by region3-function.sql. The extra data buffer is useful in aligning features between tiles when displaying the tiles graphically. The

option for setting buffer size in region3-function-bufopt.sql was made available so that users could specify a 0 buffer size, in cases where they were only interested in getting the data.

The core of each function is the EXECUTE format statement where the SQL statements are located. Below is an example of one of the SQL statements.

```
EXECUTE format(
       'WITH
        bounds AS (
          SELECT ST_TileEnvelope(%s, %s, %s) AS geomclip, ST_Expand(ST_TileEnvelope(%s, %s,
%s), 256) AS geombuf
        ),
        node_ids AS
          (SELECT G.node, G.geom, G.bathymetry
           FROM r3sim_fort_geom AS G, bounds
           WHERE ST_Intersects(G.geom, ST_Transform(bounds.geombuf, 4326))),
        mvtgeom AS (
          SELECT T.geom AS geom, S.node AS node, S.zeta AS zeta, T.bathymetry AS bathymetry
          FROM
            (SELECT node, zeta, timestamp
             FROM %s
             WHERE timestamp = %s
             AND node IN (SELECT node FROM node_ids)) S
            LEFT JOIN
            (SELECT ST_AsMVTGeom(ST_Transform(G.geom, 3857), bounds.geomclip, 4096, 256, true)
AS geom, G.node AS node, G.bathymetry AS bathymetry
             FROM node_ids AS G, bounds
             WHERE ST_Intersects(G.geom, ST_Transform(bounds.geombuf, 4326))) T
            ON S.node = T.node
        )
        SELECT ST_AsMVT(mvtgeom, %s)
        FROM mvtgeom;', z, x, y, z, x, y, quote_ident(stormtable), quote_literal(timestep),
quote_literal('public.region3_sim_storms')
```

In this statement there are four queries. The first query (bounds) selects the boundary of the tile, using the z, x, and y input tile coordinates. The second query (node_ids) selects the node, geometry, and bathymetry variables from the r3sim_fort_geom table, based on the bounds query. The third query (mvtgeom) selects the variables from the storm track table, based on the nodes selected in node_ids, and joins them with the variables selected in node_ids. It also converts the geometry from node_ids to Mapbox Vector Tile coordinate space, using the PostGIS function ST_AsMVTGeom. Finally, the fourth query converts the output from mvtgeom to Mapbox Vector Tile binary format using the PostGIS function ST_AsMVT. The binary format is Protocol Buffers.

The key to the speed of these queries is that there is not a direct join between the r3sim_fort_geom geometry table and the storm track table. Instead they are queried separately with the results of the node_ids query only being joined with mvtgeom query of the storm track table.