Data Fusion and Processing Design Document, and Toolkit for Importing the Nearshore Ocean Data into the Cube

Prepared for

US Army Corps Engineer Research and Development Center (ERDC)

Prepared by

Landry Bernard, Steve Stanic, James Braud, Vishwamithra Sunkara
University of Southern Mississippi

1. Background:

The Gulf coast of Mississippi is marked by a peculiar shallow-water shelf that stretches offshore for approximately 150 kilometers and has depths of less than 30 meters. Given the wide-ranging variabilities in Mississippi's offshore and shallow-coastal waters, the Gulf coast provides a perfect testing ground for unmanned maritime technologies, including sensor systems and sensor data fusion concepts.

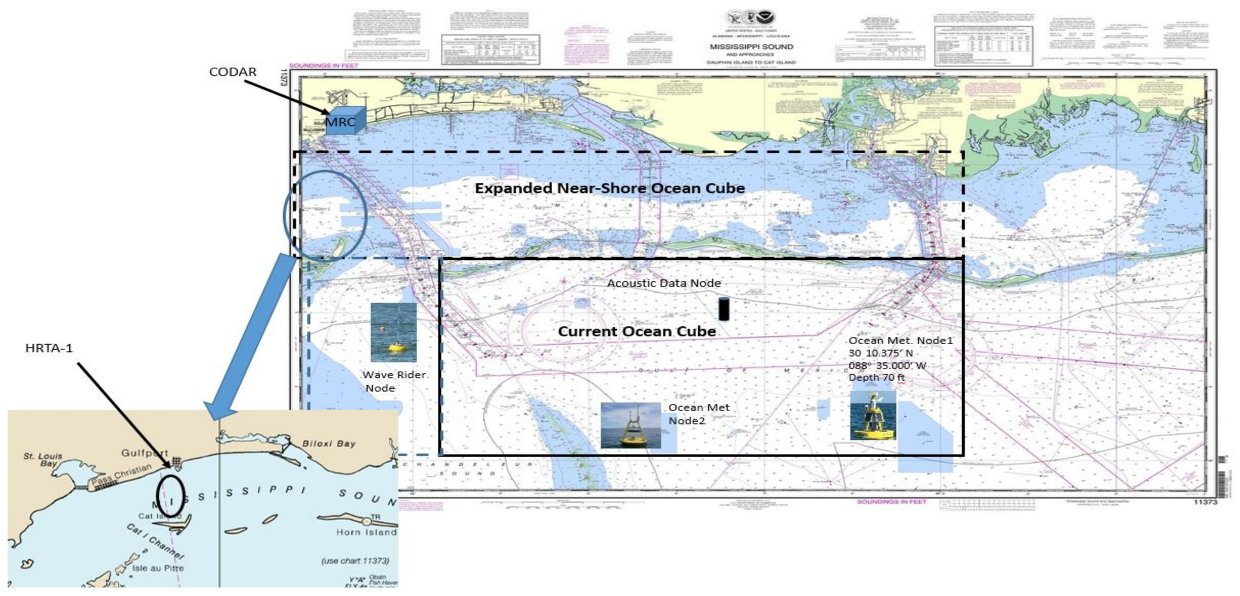


Figure 1.1: Full coverage area of Ocean Cube

To take the advantage of unique characteristics of the Gulf coast of Mississippi, the University of Southern Mississippi (USM) developed a web accessible interactive marine data tool 4-D Cube in collaboration with its research partners. USM’s 4-D Cube provides simulated and observed marine conditions in the Mississippi sound and nearby waters. Nowcasts and forecasts using high-resolution Navy and NOAA operational models have been integrated into visualization and interactive tools to describe the 4-D Cube’s operational environment. The 4-D Cube framework can visualize the ocean conditions that help the testing, performance, and evaluation of new and emerging unmanned maritime systems.

The 4-D Cube spans an area of 775 square kilometers just south of the Mississippi barrier islands with depths varying between 3 and 25 meters. The total area covered in 4-D cube is shown in ***Figure 1.1***. Also shown in the **Figure 1.1** is the high resolution near-shore coastal inset test area (HRTA-1) within the expanded near-shore cube area. Deployed in each HRTA will be several scalable underwater high-resolution sensor networks (HRSN), advanced underwater vehicle sensor technologies, and laser Light Detection and Ranging (LIDAR) systems that will fully characterize the temporal and spatial characteristics of this variable near-shore coastal test area.

The 4-D cube provides parameter fields and vertical profiles from the hydrodynamic models, and satellite and field observations. The cube is currently accommodating three widely used ocean models for visualizing corresponding model data in cube’s web-interface.

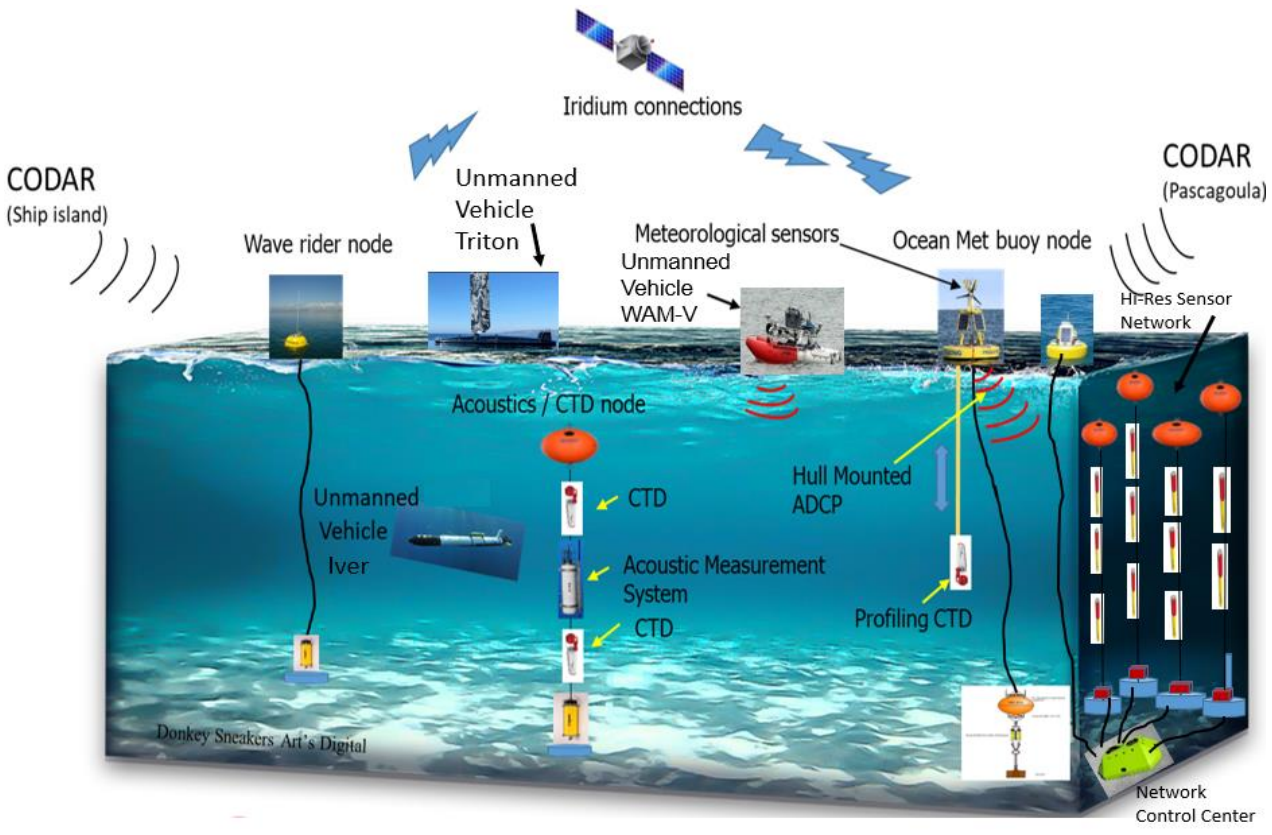


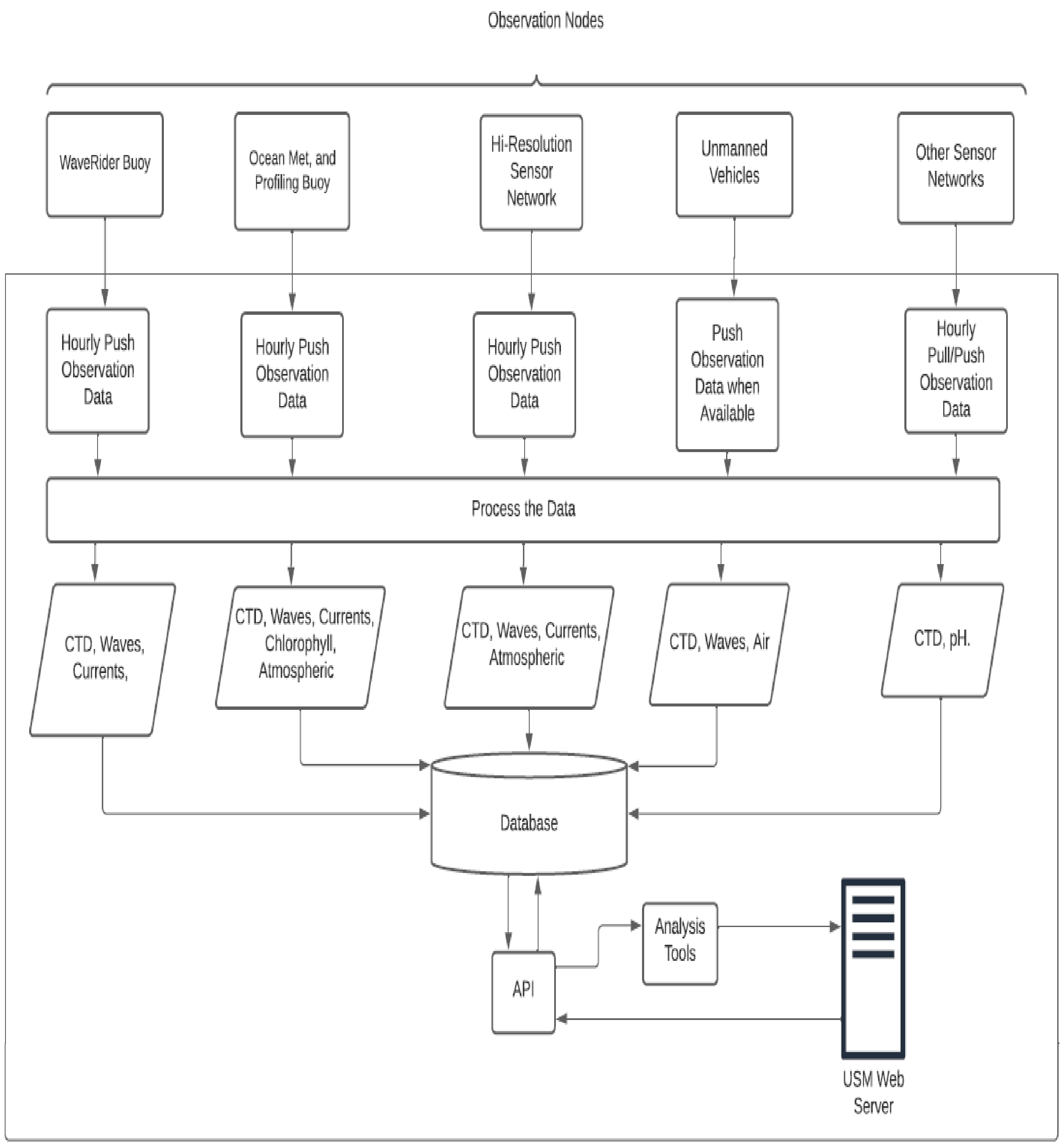
Figure 1.2: 4D Ocean Cube Training, Testing and Evaluation Area

The observations data is also collected from several observation nodes deployed at different locations within the cube area. A schematic of the 4D-Cube area with its observation nodes is shown in ***Figure 1.2***.

2. Observations Data Fusion and Processing Design:

The 4-D cube shelters several observation nodes to collect oceanographic, meteorological, and other environmental data. Each observation node may be fitted with various sensors, allowing for the collection of various types of environmental data at various times. While in-situ observation nodes collect and transmit data at regular intervals, unmanned vehicles both surface and underwater vehicles collect the observations data along their path for specific missions. The data transmission system for each node may also differ. Thus, the collected data must be processed and organized to make useful data visualization in ocean cube. ***Figure 2.1*** shows the data ingest of observation nodes to ocean cube server. During the observation data processing, the observation data is organized in the database as per the Enhanced Entity Relationship (EER) model shown in ***Figure 2.2.*** The observations data stored in the database is accessible by making corresponding queries to the database. When users click on a node in the ocean cube web interface, they will be directed to Grafana, a visualization and analytics web tool to access and download the observations data. As the capabilities of 4D-cube evolve over time, more features will be added to the ocean cube interface to make it more efficient.

The deployed observation nodes in 4-D Cube area consist of a wave rider buoy system (WaveRider), an ocean meteorological buoy (Viking), a Hi-resolution sensor network (HRSN) and a subsurface ambient noise and CTD mooring along with several unmanned vehicles. Below we describe each observation node specifications, their data communication system, data fusion and processing of collected data in the ocean cube server.



Observation Nodes
WaveRider Buoy
Ocean Met, and Profiling Buoy
Hi-Resolution Sensor Network
Unmanned Vehicles
Other Sensor Networks
Hourly Push Observation Data
Hourly Push Observation Data
Hourly Push Observation Data
Push Observation Data when Available
Hourly Pull/Push Observation Data
Process the Data
CTD, Waves, Currents,
CTD, Waves, Currents, Chlorophyll, Atmospheric
CTD, Waves, Currents, Atmospheric
CTD, Waves, Air
CTD, pH.
Database
API
Analysis Tools
USM Web Server

Figure 2.1: Observation Nodes Data Ingest to Ocean Cube Architecture

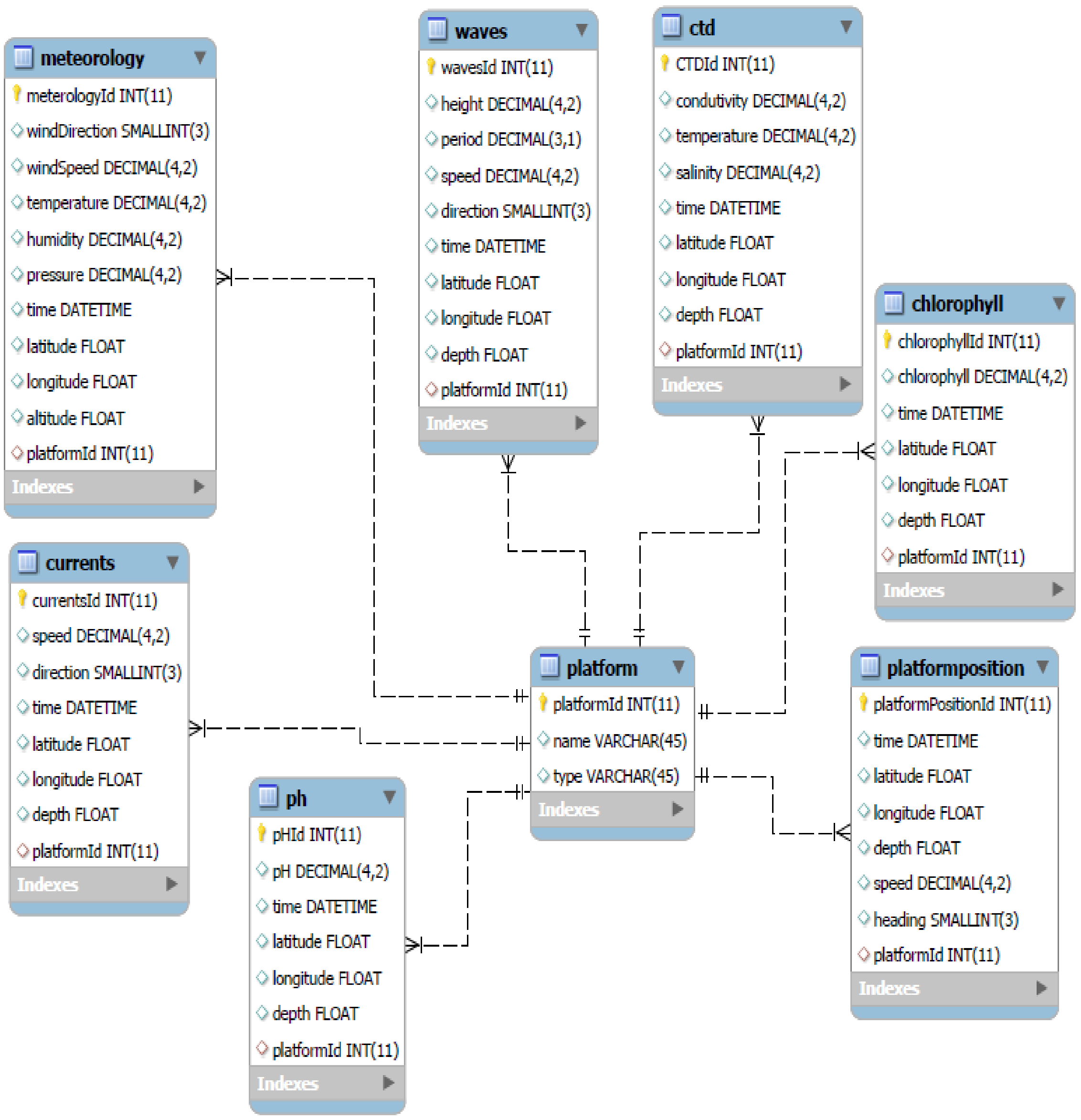
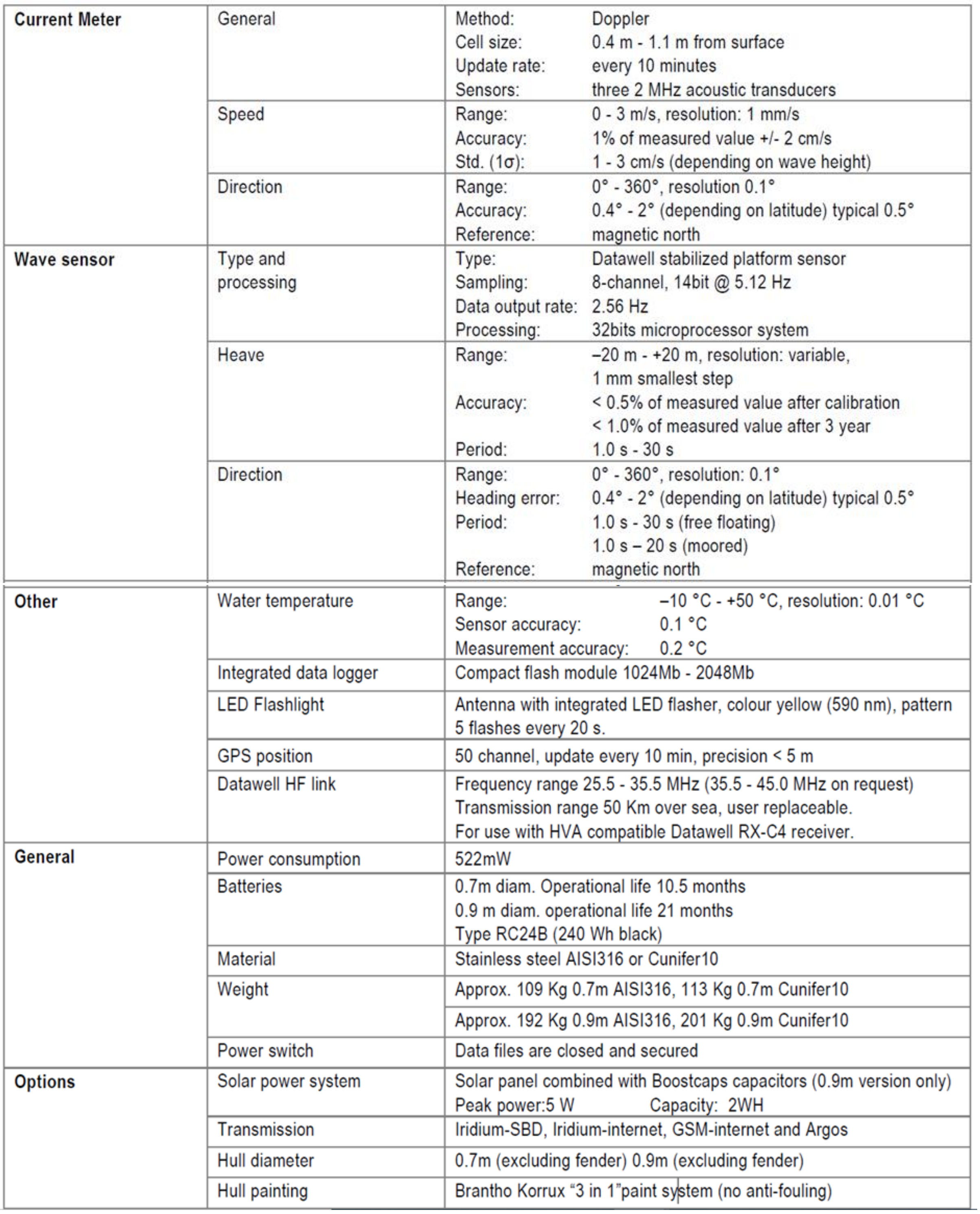


Figure 2.2: Enhanced Entity Relationship (EER) data model

1. Directional WaveRider 4 (DWR4) Buoy:

The DWR4 buoy is equipped with wave sensor and acoustic current meter. It measures waves height, wave direction, surface current and water temperature. The acoustic current meter combines a robust measuring principle, Doppler shift, with a mechanical design that avoids vulnerability. This results in a coherent oceanographic instrument that meets the challenges at sea. The specifications of the DWR4 buoy are shown in ***Figure 2.1.1***.

The raw data collected by the buoy is stored on the logger flash card (BVA files) and output through the radio link (HVA files). However, the buoy also outputs processed data through the radio link or via communication satellite. Processed data are much more compact, nevertheless they still give a good impression of the sea state. Especially for the limited data transmission capacity of satellites, the processing and compressing is essential. While the buoy can transmit the data to the user using several different communication systems, USM's



Current Meter	General	Method: Doppler Cell size: 0.4 m - 1.1 m from surface Update rate: every 10 minutes Sensors: three 2 MHz acoustic transducers
	Speed	Range: 0 - 3 m/s, resolution: 1 mm/s Accuracy: 1% of measured value +/- 2 cm/s Std. (1σ): 1 - 3 cm/s (depending on wave height)
	Direction	Range: 0° - 360°, resolution 0.1° Accuracy: 0.4° - 2° (depending on latitude) typical 0.5° Reference: magnetic north
Wave sensor	Type and processing	Type: Datawell stabilized platform sensor Sampling: 8-channel, 14bit @ 5.12 Hz Data output rate: 2.56 Hz Processing: 32bits microprocessor system
	Heave	Range: –20 m - +20 m, resolution: variable, 1 mm smallest step Accuracy: < 0.5% of measured value after calibration < 1.0% of measured value after 3 year Period: 1.0 s - 30 s
	Direction	Range: 0° - 360°, resolution: 0.1° Heading error: 0.4° - 2° (depending on latitude) typical 0.5° Period: 1.0 s - 30 s (free floating) 1.0 s – 20 s (moored) Reference: magnetic north
Other	Water temperature	Range: –10 °C - +50 °C, resolution: 0.01 °C Sensor accuracy: 0.1 °C Measurement accuracy: 0.2 °C
	Integrated data logger	Compact flash module 1024Mb - 2048Mb
	LED Flashlight	Antenna with integrated LED flasher, colour yellow (590 nm), pattern 5 flashes every 20 s.
	GPS position	50 channel, update every 10 min, precision < 5 m
	Datawell HF link	Frequency range 25.5 - 35.5 MHz (35.5 - 45.0 MHz on request) Transmission range 50 Km over sea, user replaceable. For use with HVA compatible Datawell RX-C4 receiver.
General	Power consumption	522mW
	Batteries	0.7m diam. Operational life 10.5 months 0.9 m diam. operational life 21 months Type RC24B (240 Wh black)
	Material	Stainless steel AISI316 or Cunifer10
	Weight	Approx. 109 Kg 0.7m AISI316, 113 Kg 0.7m Cunifer10
		Approx. 192 Kg 0.9m AISI316, 201 Kg 0.9m Cunifer10
	Power switch	Data files are closed and secured
Options	Solar power system	Solar panel combined with Boostcaps capacitors (0.9m version only) Peak power:5 W Capacity: 2WH
	Transmission	Iridium-SBD, Iridium-internet, GSM-internet and Argos
	Hull diameter	0.7m (excluding fender) 0.9m (excluding fender)
	Hull painting	Brantho Korrux "3 in 1"paint system (no anti-fouling)

Figure 2.1.1: Directional WaveRider 4 Buoy System Specifications

DWR4 buoy uses Iridium-Short Burst Data (SBD) satellite communication system as shown in **Figure 2.1.2**. Iridium SBD communicates data via short messages, much the same as GSM text messages (SMS). Messages originating from the Iridium SBD modem are limited to 340 bytes and messages terminating at the modem are limited to 270 bytes. Iridium SBD messages follow the Datawell Message Format (DMF).

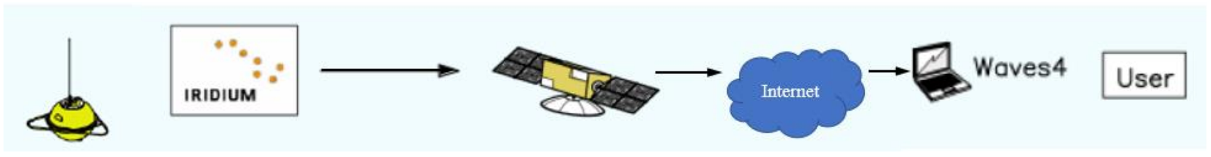


Figure 2.1.2: DWR4 Iridium-SBD communication

An Iridium modem dedicated to SBD is used in DWR4 buoys. This 9603N SBD modem does not require a SIM-card. The Iridium SBD service will only use the IMEI number. Iridium SBD messages follow the Datawell Message Format (DMF), and the message transmission interval can be set independently for each message. ***Table 2.1.1*** lists the messages that are available for Iridium SBD. To configure the messages and intervals for Iridium SBD a configuration message must be sent by email to the buoys SBD modem. The Waves4 software offers a configurator to do so conveniently.

MsgID	Description	Size (bytes)
0xF20	Heave spectrum message	161
0xF21	Primary directional spectrum message	309
0xF23	Spectrum synchronisation message	22
0xF25	Directional spectral parameters message	27
0xF26	Online upcross wave statistics message	25
0xF28	Secondary directional spectrum message	459
0xF80	GPS position message	14
0xF81	Sea surface temperature message	10
0xF82	Acoustic current meter message	29
0xFB0	DWR4/ACM summary message	30
0xFC1	System message for the DWR4	67
0xFC3	Battery life expectancy	9

Table 2.1.1: List of DMF messages available for Iridium SBD

The Waves4 software supports the storage, processing, and presentation of received Iridium SBD emails, as well as composing configuration emails. It can deal with a network of Waverider buoys equipped with Iridium SBD. Furthermore, the PC does not have to be permanently on-line or permanently powered since emails will be buffered by the internet provider. The Waves4 is setup in USM's ocean cube server to store and process the data received from the buoy.

DWR4 also has an internal data storage slot. A data logger is standard on the DWR4. Data is logged on a compact-flash card inserted in a slot in electronics unit. The logger stores the same data as transmitted over the HF link: raw displacements, calculated (directional) spectra, wave parameters, system information, etcetera. Logged files (binary) can be converted into CSV files using the library *libdatawell*. Apart from data, the logger also keeps a log of system events in "SYSLOG.TXT". This is useful for diagnosis. The standard compact flash card of 512 Mb will hold almost 1 year of data. After that the oldest data is overwritten. The logger stores all data in the BVA format, Binary Vector format A. BVA data is the binary equivalent of the HVA data (Hexadecimal Vector format A) as transmitted over the HF link. BVA data are collected in files containing 4 days of data. The name convention of the files is "YYYYMMDD.BVA" where YYYY is year, MM is month, and DD is day.

The date refers of the first day of the data in the file. For example, the file "20120305.BVA" contains data from March 5, 6, 7 and 8 2012. Every new month a new file starts, leaving the last file of the month with a variable number of days. The logger generates about 462 Mbyte of BVA data per year. BVA files can be read by Waves4.

As the SBD messages transmitted by DWR4 buoy are collected and run through waves4 in ocean cube server, the CSV files are generated and stored in the ocean cube server at a dedicated location. The Waves4 software generates CSV (Comma Separated Value) files that contain wave data from DWR4 buoy. ***Table 2.1.2*** shows the list of messages defined for the HVA files. Depending on the type of buoy from the buoy manufacturer Datawell some messages may not be transmitted. Out of all the CSV message files generated and stored by the Waves4 in ocean cube server, just 0xFB0, 0xF80, 0xF82 and 0xF25 files are used to extract the data that is intended to be displayed in 4-D Cube. This extracted data from these files is pushed into the database and organized in corresponding tables as per the EER model shown in ***Figure 2.2***. The remaining files are available in server for any future use. Below the format of files 0xFB0, 0xF80, 0xF82 and 0xF25, and the parameters that are extracted from these files is described. All CSV messages share a common header as described in ***Table 2.1.3***.

MsgID	Description	Status
0xF20	Heave spectrum message	a
0xF21	Primary directional spectrum message	a
0xF22	Secondary directional spectrum message	r
0xF23	Spectrum synchronisation message	a
0xF24	Spectral parameters message	a
0xF25	Directional spectral parameters message	a
0xF26	Online upcross wave statistics message	a
0xF27	Low frequency heave spectrum message	a
0xF28	Secondary directional spectrum message	a
0xF29	Upcross wave height quantiles message	a
0xF2A	Upcross wave period quantiles message	a
0xF80	GPS location message	a
0xF81	Sea surface temperature message	a
0xF82	Acoustic current meter message	a
0xF83	CAT4 air temperature message	a
0xFB0	DWR4 /ACM summary message	a
0xFC0	System message for the GPS-DWR4	a
0xFC1	System message for the DWR4	a
0xFC2	System message for the WR4	r
0xFC3	Battery life expectancy message	a

Table 2.1.2: Overview of the defined messages

Field	Description	Unit
1	Timestamp	-
2	Datastamp	-

Table 2.1.3: Format of the common message header

The Timestamp is the timestamp at which the data acquisition for the message started. This number represents the number of seconds elapsed after 1-1-1970 in UTC time, excluding leap seconds. The Datastamp is an internal number used to identify the buoy and based on the combination of Hatch UID and Hull UID.

From the 0xF25 file, the significant wave height (H_s), mean wave period (T_1), average wave period (T_z) and mean wave direction (θp) values are extracted and pushed into the database. The mean wave direction (θp) is the direction from which the waves arrive. The value 0 corresponds to North, π/2 or 90° with East. All values refer to *magnetic* North. The format for the spectral parameter message is shown in ***Table 2.1.4***.

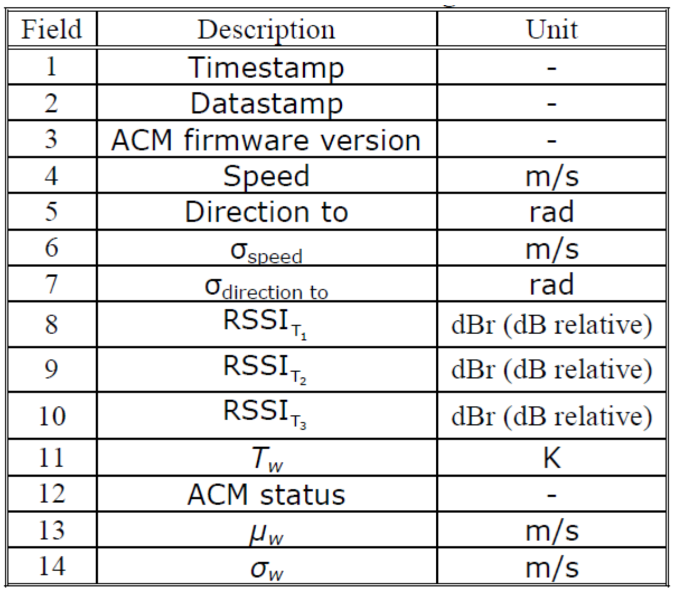
Field	Description	Unit
1	Timestamp	-
2	Datastamp	-
3	Number of segments used	-
4	H_s	m
5	T_I	s
6	T_E	s
7	T_1	s
8	T_z	s
9	T_3	s
10	T_c	s
11	R_p	-
12	T_p	s
13	S_{max}	m^2/Hz
14	θ_p	rad
15	σ_p	rad

Table 2.1.4: Format of the spectral parameters message.

The GPS location message 0xF80 file contains the current location of the buoy. The format for the GPS location message is shown in ***Table 2.1.5***. The latitude and longitude location of the buoy is extracted from this file and is pushed into the corresponding table in the database.

Field	Description	Unit
1	Timestamp	-
2	Datastamp	-
3	Latitude	rad
4	Longitude	rad

Table 2.1.5: Format of the GPS location message.
The Latitude is the latitude the location. A positive value for the Latitude means the location is on the northern hemisphere. A negative value for the Latitude means the location is on the southern hemisphere. The Longitude is the longitude of the location. A positive value for the Longitude means the location lies east of the Prime Meridian. A negative value for the Longitude means location lies west of the Prime Meridian.
The message file 0x82 contains the information of the acoustic current meter. The format of the acoustic current meter message is shown in ***Table 2.1.6.***

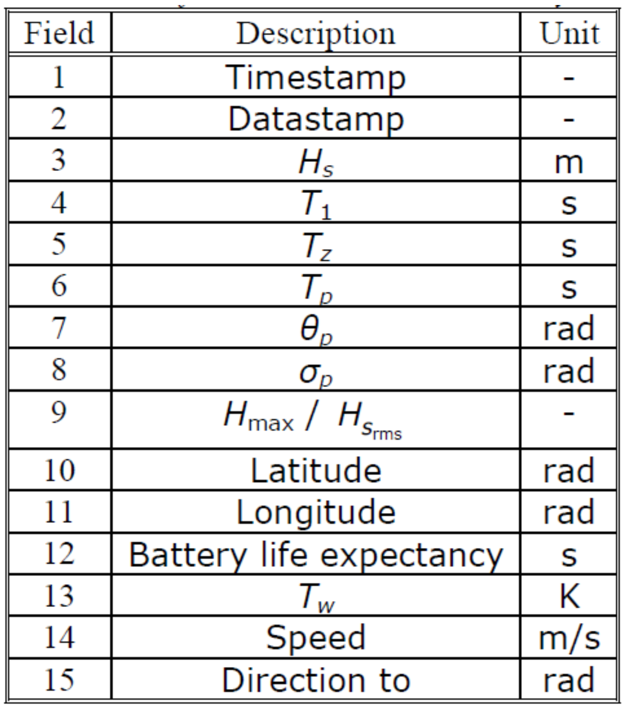


Field	Description	Unit
1	Timestamp	-
2	Datastamp	-
3	ACM firmware version	-
4	Speed	m/s
5	Direction to	rad
6	σ_{speed}	m/s
7	$\sigma_{\text{direction to}}$	rad
8	RSSI_{T_1}	dBr (dB relative)
9	RSSI_{T_2}	dBr (dB relative)
10	RSSI_{T_3}	dBr (dB relative)
11	T_w	K
12	ACM status	-
13	μ_w	m/s
14	σ_w	m/s

Table 2.1.6: Format of the acoustic current meter message.

The Speed is the mean current speed. The Direction to is the mean current direction. Current direction is defined as direction the water particles are moving towards. The T_w is the temperature of the water at the sea surface. The mean current speed and direction, and the sea surface water temperature are the parameters that are extracted from the 0x82 message file.

The message file 0xFB0 contains all the above parameters values as shown in the ***Table 2.1.7.*** This message contains a summary of the information of a DWR4 /ACM buoy.



Field	Description	Unit
1	Timestamp	-
2	Datastamp	-
3	H_s	m
4	T_1	s
5	T_z	s
6	T_p	s
7	θ_p	rad
8	σ_p	rad
9	$H_{\max} / H_{s_{\text{rms}}}$	-
10	Latitude	rad
11	Longitude	rad
12	Battery life expectancy	s
13	T_w	K
14	Speed	m/s
15	Direction to	rad

Table 2.1.7: Format of the DWR4 /ACM summary message.

2. The Oceanographic and Meteorology (Viking) Buoy:

The Oceanographic and Meteorology Buoy shown in ***Figure 2.2.1*** is designed to accommodate the use of many instruments to satisfy the oceanographic researcher's needs. These instruments, offered as options, are grouped into three main categories:

Optical parameters

- Measurement of the light spectrum level in air and water
- Measurement of the concentration of coloured biological matter dissolved in sea water

Meteorological parameters

- Wind direction and speed
- Atmospheric temperature and humidity
- Atmospheric pressure

Oceanographic parameters

- Water temperature
- Water salinity, density and conductivity
- Measurement of the water flow at many programmable depths

A Viking View software is designed to help the researcher interpret the information from the buoy. This information is sent to a land station via cellular modem or satellite or a combination of cellular and Internet. USM's Viking buoy is set at 1 hour intervals to transmit data from the buoy to the ocean cube server via Iridium satellite.

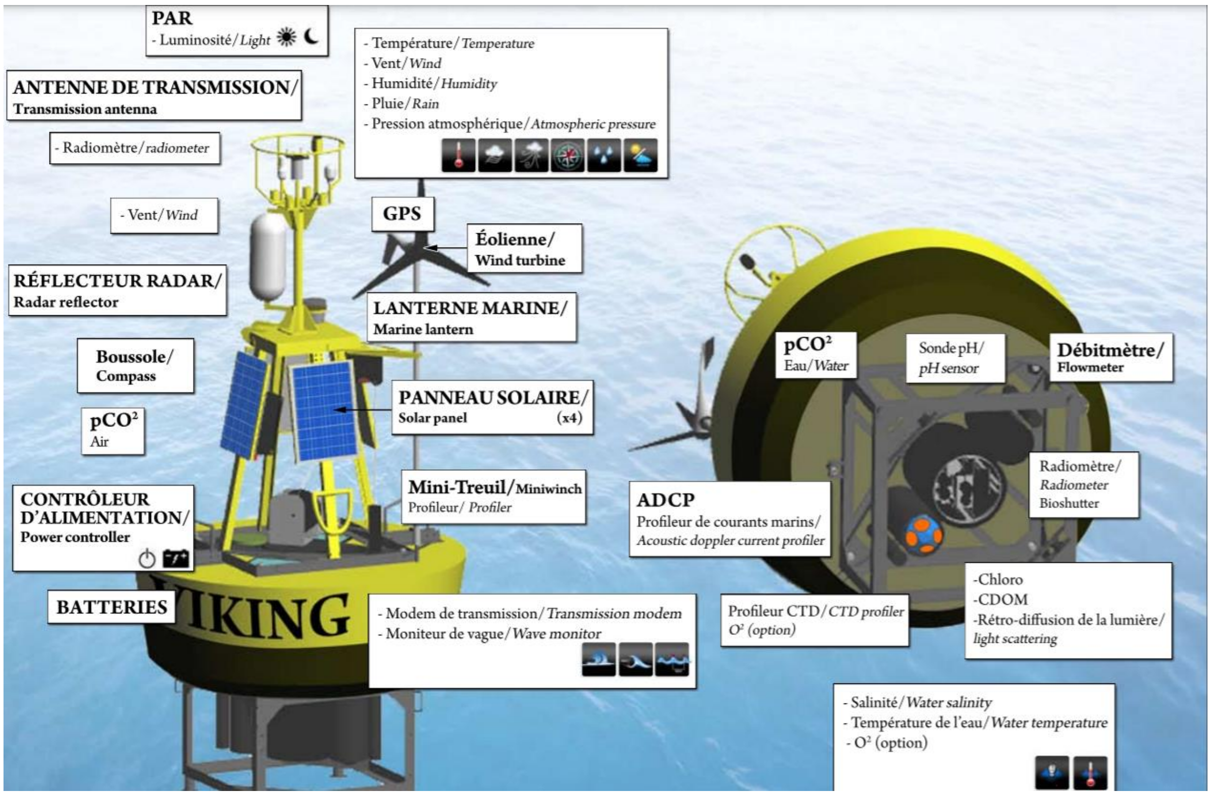


Figure 2.2.1: Viking Buoy

The buoy is equipped with a automate profiler including the controller that can determine automatically if the conditions are good to collect data.

Characteristics:

- Approximated height of 4,57 meters (180 in)
- Approximated diameter of 2 meters (79 in)
- In air weight: around 1100 kg (2425 lbs) (according to installed equipments)
- 2 x batteries
- Solar panels
- Operating temperature
  - Water: -2°C to 35°C (28,4°F to 95°F)
  - Air: -10°C to 50°C (14°F to 122°F)
- Material:
  - Wetter section: SS316 and Titanium
  - Air section: Paint Aluminum
  - Floater: Medium density UV-stabilized virgin polyethylene

Communication:

- Frequency range
  - Cellular: 3G or 4G
  - Satellite: 1616 to 1626.5 MHz
- Operating range
  - Cellular: Where the signal is available
  - Operating fee: Depend on the paid service and the size of the transferred data
  - Satellite: Worldwide
  - Operating fees: Required paid service (IRIDIUM)
- Link throughput:
  - Cellular: 50 mb/s
  - Satellite: 4,8 kbps max

The sensors on board the USM's Viking buoy allow it to collect meteorological and oceanographic data. The Viking buoy can use two methods of communication as shown in ***Figure 2.2.2.***

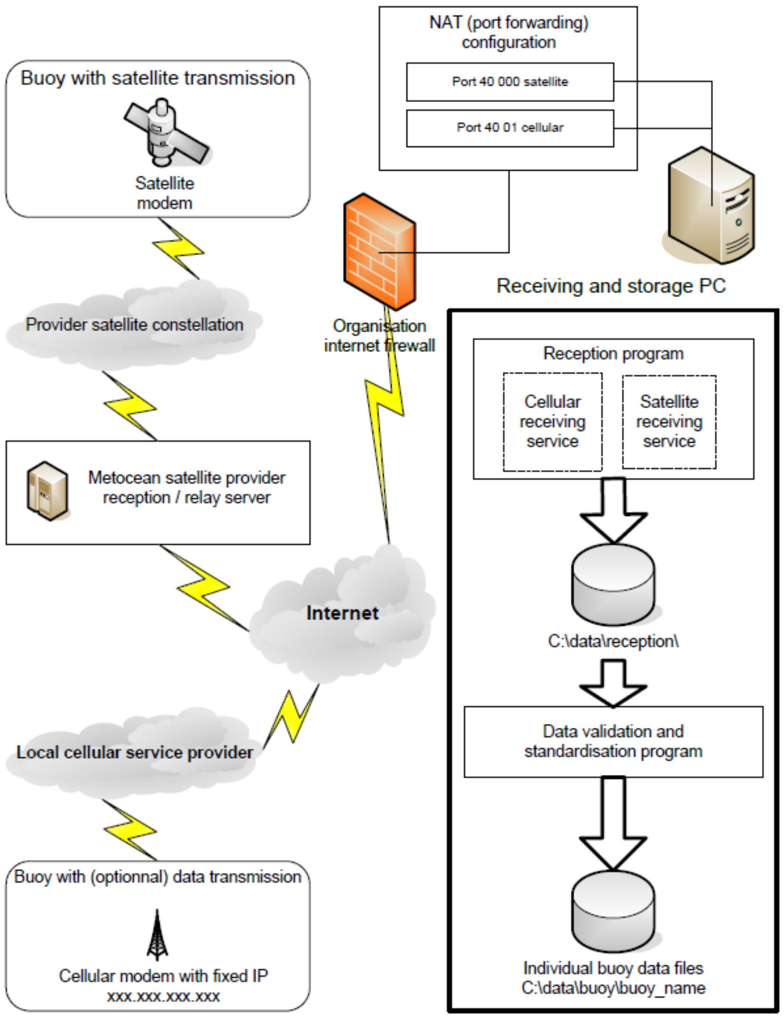

NAT (port forwarding) configuration
Port 40 000 satellite
Port 40 01 cellular
Buoy with satellite transmission
Satellite modem
Provider satellite constellation
Organisation internet firewall
Receiving and storage PC
Reception program
Cellular receiving service
Satellite receiving service
Metocean satellite provider reception / relay server
Internet
C:\data\reception\
Data validation and standardisation program
Local cellular service provider
Buoy with (optionnal) data transmission
Cellular modem with fixed IP
xxx.xxx.xxx.xxx
Individual buoy data files
C:\data\buoy\buoy_name

Figure 2.2.2: Overview of the communication and system

The first one is the SBD (Short Burst Data) that is used by the buoy to transmit the collected data to the main computer. The data packets will be received by the computer that runs the reception software.

The second one is CSD (Circuit Switched Data) that is used to communicate with the buoy and enter in menus for any reasons, such as changing some settings.

In satellite mode, normally we set the buoy to take measurements each 30 minutes and send only the abbreviated data that with a tag [MO]. You can select any information that you want to be transmitted, but it has a non-negligible cost.

The reception program installed in ocean cube server is responsible for the cellular and satellite communication setup. Once, the data service is started, it manages satellite and incoming cellular connection. The data packet is recorded in a dedicated receiving directory in the ocean cube server. The data packet will be picked up by the analysis program, this allows receiving data from multiple buoys. If reception services are not running, data will be waiting at the satellite provider data center until service is running back again and the provider will retry sending the data. As for the cellular communication, buoy will hold data until the destination host with receiving service is running.

The validation and standardization program is responsible for picking up received data packets from the pick-up directory, using the proper method for extracting data and data formats. Then validation is done on received data packet, if it has an error it is moved to the received with error directory, otherwise the extraction is done on the received data packet, individual buoy daily data files are updated from the data packet received. The received data packet is then archived to the received directory for its services.

All files are plain text CSV text files. The validation program extracts all valid data for all equipment and appends information to the resulting daily data files by type. Special file type RAW contains all received data packets exactly as received before any validation and formatting.

The winch data communication to the reception computer is shown in ***Figure 2.2.3.*** Once the result of a winch mission is finished, the data is received at the Buoy controller from the winch controller and the data is broken down into small data packets due to satellite and cellular transmission limits and sent to the reception program.

Reception program intercept winch mission data packets create a temporary file until all data received. Once received, file is copied to the specific buoy winch data directory "WINCH_MISSIONS" subdirectory for that buoy.

There are two types of winch mission data that can be configured in the buoy controller, full data transmission and low-resolution data transmission. Low resolution is suited for satellite transmission reducing dramatically transmission cost while maintaining data validity with slightly less precision. All winch mission full data is always stored on a flash card in the buoy controller and are recoverable with a simple communication using the buoy menu also known as win mission dump file.

All winch mission full data is always stored on a flash card in the buoy controller and are recoverable with a simple communication using the buoy menu also known as win mission dump file.

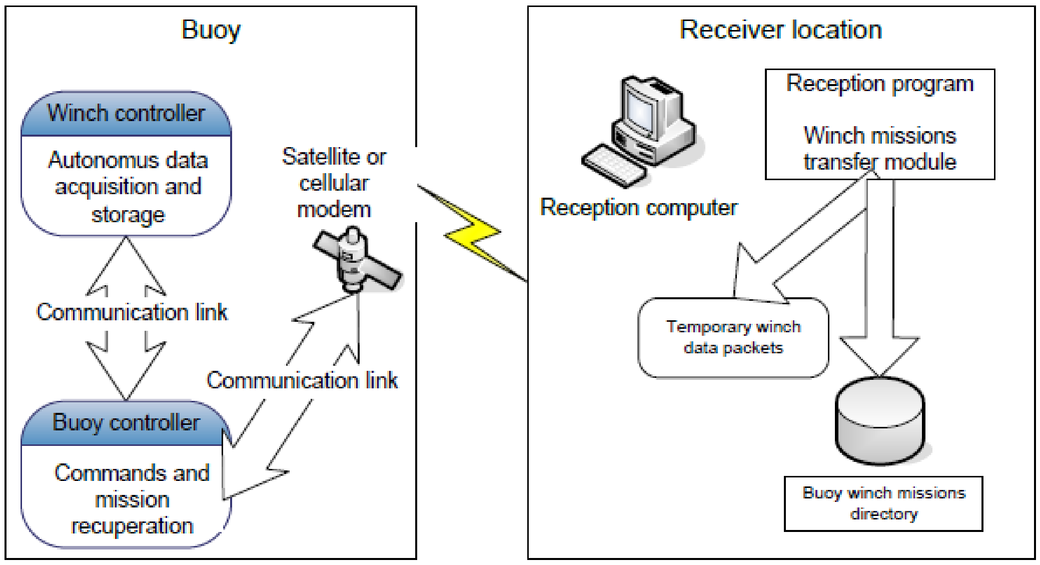

Buoy
Winch controller
Autonomus data acquisition and storage
Communication link
Satellite or cellular modem
Communication link
Buoy controller
Commands and mission recuperation
Receiver location
Reception computer
Reception program
Winch missions transfer module
Temporary winch data packets
Buoy winch missions directory

Figure 2.2.3: Overview of Winch Mission Data Transfer

After the validation of received data packets is done, the reception software generates .dat files tagged with specific data, time, and name. Each of these files contains data relevant to the sensors mounted on the Viking buoy. Additional files containing a summary of user-specified sensor data are also produced by the reception program. The file with tag ” _SD_” is a summary data file with the data format as shown below

<buoy name>_SD_<date>.dat
#1 Name of the buoy
#2 date buoy
#3 hour buoy
#4 latitude buoy
#5 longitude buoy
#6 speed of the wind in knots
#7 maximal speed of the wind in knots
#8 wind direction in degree
#9 air temperature in degree Fahrenheit
#10 air humidity relative in %
#11 air pressure in inHg
#12 Wave, period in second
#13 Wave, average height in meter
#14 Wave, height of the biggest wave in meter
#15 Voltage of the batteries in Volt
#16 Power of the charging solar in ampere
#17 Power of the charging Wind turbine in ampere
#18 Power drained in ampere
#19 Pitch in degree (compass)
#20 Roll in degree (compass)
#21 Heading buoy (compass) in degree
#22 Moving speed (GPS) in m/s
#23 Moving direction (GPS) in degree
#24 Rain accumulation from midnight in mm
#25 Current (ADCP RTI or RDI) bin #1 in m/s
#26 Current direction (ADCP RTI or RDI) in degree
#27 Water presence in the buoy controller box (0= no water, 1= water)
#28 Water presence in the Power controller box (0= no water, 1= water)
#29 Water presence in the Winch controller box (0= no water, 1= water)

Meteorological and oceanographic data can be extracted from this file and the CTD data can be extracted from the files with tag “WDATA_” stored in a separate folder. The data format of files tagged with “WDATA_” is shown below.

WDATA_<buoy_name>_<date and time>.txt

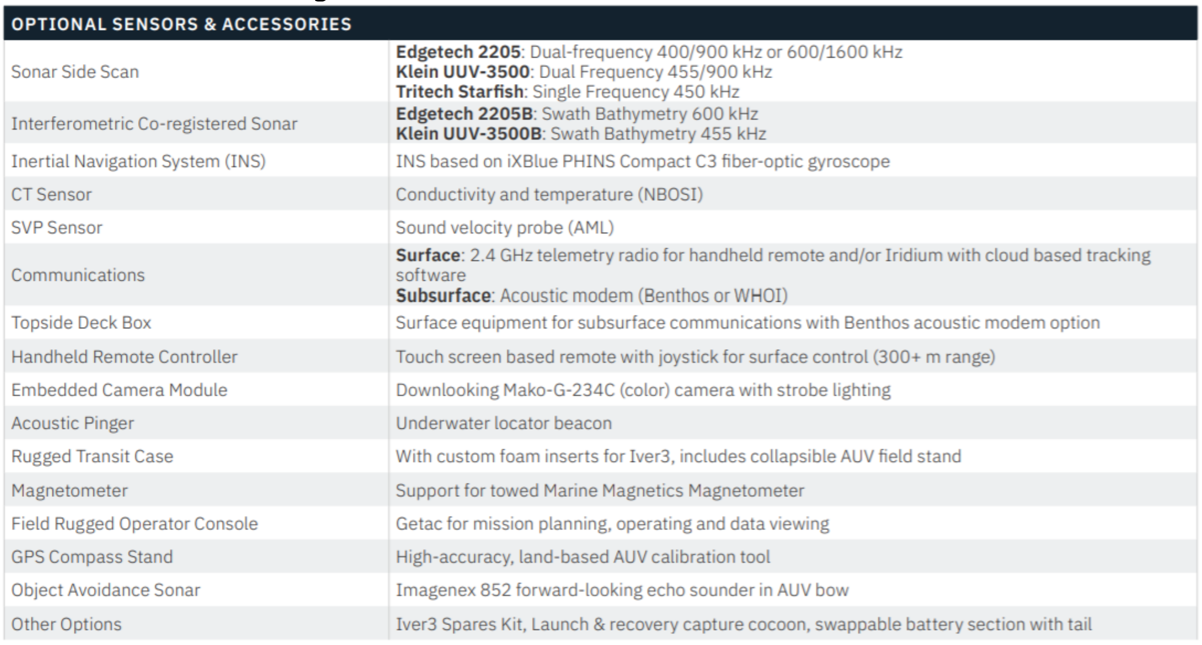
#1 Date hour
#2 conductivity in mS/cm
#3 temperature in C
#4 pressure in dbar
#5 depth in m,
#6 salinity in PSU

If the data is sent by satellite, 1 data by meter with less digits and only the values taken when it is going down are transmitted, to save on transmission fees. In such case, low resolution files are generated by the reception software tagged with "_LOWRES".

WDATA_<buoy_name>_<date and time>_LOWRES.txt
#1 Date hour
#2 temperature in mS/cm
#3 depth in m
#4 salinity (PSU)

3. Iver3-Autonomous Underwater Vehicle

Iver3 is the first commercially developed family of low-cost Autonomous Underwater Vehicles (AUVs). They are ideal for coastal applications such as sensor development, general survey work, sub-surface security, research and environmental monitoring. The Iver3 AUV vehicle can survey in coastal waters in depths ranging from 1 to 100 meters and in water temperature ranging from 0º C to 35º C. Iver3 vehicle can be equipped with various sensors from the list below in ***Figure 2.3.1***



OPTIONAL SENSORS & ACCESSORIES	
Sonar Side Scan	**Edgetech 2205**: Dual-frequency 400/900 kHz or 600/1600 kHz **Klein UUV-3500**: Dual Frequency 455/900 kHz **Tritech Starfish**: Single Frequency 450 kHz
Interferometric Co-registered Sonar	**Edgetech 2205B**: Swath Bathymetry 600 kHz **Klein UUV-3500B**: Swath Bathymetry 455 kHz
Inertial Navigation System (INS)	INS based on iXBlue PHINS Compact C3 fiber-optic gyroscope
CT Sensor	Conductivity and temperature (NBOSI)
SVP Sensor	Sound velocity probe (AML)
Communications	**Surface**: 2.4 GHz telemetry radio for handheld remote and/or Iridium with cloud based tracking software **Subsurface**: Acoustic modem (Benthos or WHOI)
Topside Deck Box	Surface equipment for subsurface communications with Benthos acoustic modem option
Handheld Remote Controller	Touch screen based remote with joystick for surface control (300+ m range)
Embedded Camera Module	Downlooking Mako-G-234C (color) camera with strobe lighting
Acoustic Pinger	Underwater locator beacon
Rugged Transit Case	With custom foam inserts for Iver3, includes collapsible AUV field stand
Magnetometer	Support for towed Marine Magnetics Magnetometer
Field Rugged Operator Console	Getac for mission planning, operating and data viewing
GPS Compass Stand	High-accuracy, land-based AUV calibration tool
Object Avoidance Sonar	Imagenex 852 forward-looking echo sounder in AUV bow
Other Options	Iver3 Spares Kit, Launch & recovery capture cocoon, swappable battery section with tail

Figure 2.3.1: Optional Sensors for Iver3-AUV

USM's Iver3 vehicle as shown in ***Figure 2.3.2*** is equipped with a CT sensor that measures conductivity, temperature, and salinity. The vehicle can measure this data along its surface and sub-surface tracks at varying depths. Iver3 does not have the capability to relay data in real time while it is being recorded, but the data can be recovered from the vehicle after the mission is completed. The headers of the data log file retrieved from the vehicle is shown in ***Figure 2.3.3.*** The data files recovered from the vehicle after the mission are uploaded to a dedicated folder assigned for this vehicle in the ocean cube server. Afterwards, the temperature and salinity information from these data sets is extracted to update the ocean cube database.

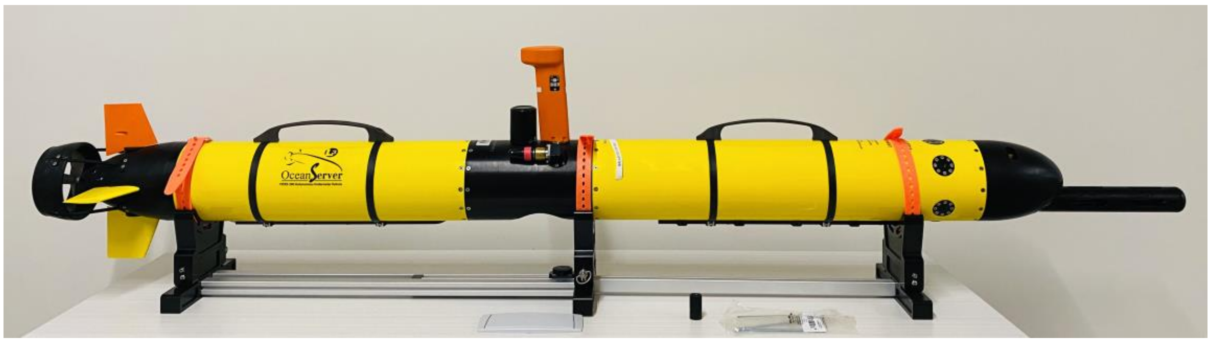
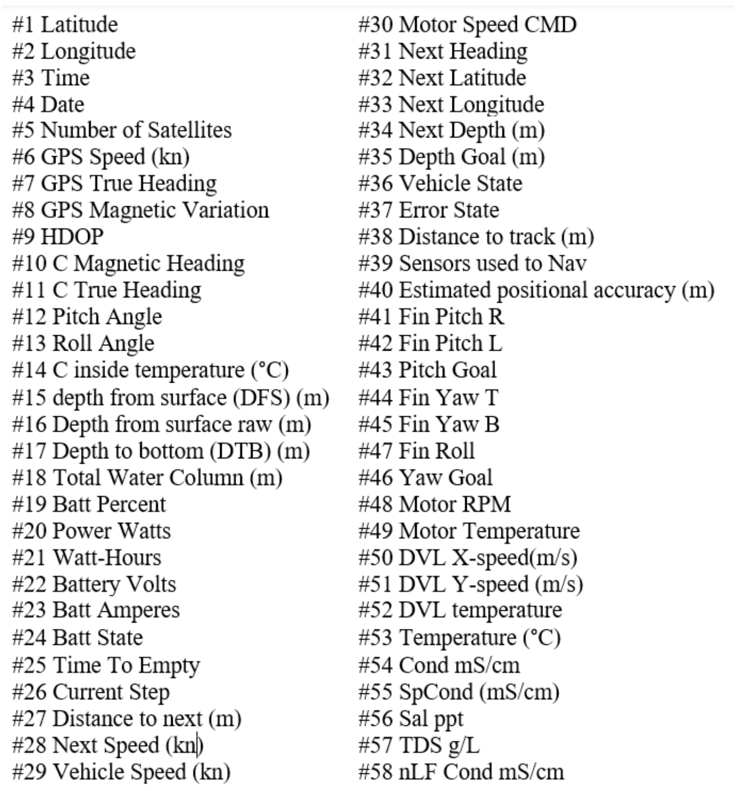


Figure 2.3.2: Autonomous Underwater Vehicle Iver equipped with CTD sensor.



#1 Latitude	#30 Motor Speed CMD
#2 Longitude	#31 Next Heading
#3 Time	#32 Next Latitude
#4 Date	#33 Next Longitude
#5 Number of Satellites	#34 Next Depth (m)
#6 GPS Speed (kn)	#35 Depth Goal (m)
#7 GPS True Heading	#36 Vehicle State
#8 GPS Magnetic Variation	#37 Error State
#9 HDOP	#38 Distance to track (m)
#10 C Magnetic Heading	#39 Sensors used to Nav
#11 C True Heading	#40 Estimated positional accuracy (m)
#12 Pitch Angle	#41 Fin Pitch R
#13 Roll Angle	#42 Fin Pitch L
#14 C inside temperature (°C)	#43 Pitch Goal
#15 depth from surface (DFS) (m)	#44 Fin Yaw T
#16 Depth from surface raw (m)	#45 Fin Yaw B
#17 Depth to bottom (DTB) (m)	#47 Fin Roll
#18 Total Water Column (m)	#46 Yaw Goal
#19 Batt Percent	#48 Motor RPM
#20 Power Watts	#49 Motor Temperature
#21 Watt-Hours	#50 DVL X-speed(m/s)
#22 Battery Volts	#51 DVL Y-speed (m/s)
#23 Batt Amperes	#52 DVL temperature
#24 Batt State	#53 Temperature (°C)
#25 Time To Empty	#54 Cond mS/cm
#26 Current Step	#55 SpCond (mS/cm)
#27 Distance to next (m)	#56 Sal ppt
#28 Next Speed (kn)	#57 TDS g/L
#29 Vehicle Speed (kn)	#58 nLF Cond mS/cm

Figure 2.3.3: Iver3-3089 Data Log Format

4. Hi-Resolution Sensor Network (HRSN)

USM custom designed a high-resolution sensor network as shown in ***Figure 2.4.1*** consisting of several sensors that acquire oceanographic, meteorological and acoustic data at various depths. This HRSN is deployed at near shore location HRTA1 as shown in ***Figure 1.1.*** The HRSN is confined to an area of 35m radius circular region centered from HRSN station as shown in ***Figure 2.4.2.*** A total of 18 temperature sensors, two CTD sensors, a tidal sensor, an ADCP, a hydrophone, a wave sensor, a wind sensor and a weather station are included in the HRSN. Out of the nine temperature nodes, two temperature sensors are attached to each node, and each node is spaced 5m, 10m, and 20m apart from the HRSN station. One CTD sensor is attached to each corresponding CTD node. Hydrophone, tidal sensor and ADCP can be found at HRSN station. The HRSN station is connected to buoy on surface through a data cable. Wind sensor and weather station are on the buoy along with a cellular modem. The main controller on the buoy, known as vak-controller, collects observations data from each sub-surface node as well as wind, wave, and weather station data for data transmission to the ocean cube database through cellular communication.

HRSN has two software components, vak-acquisition and vak-reception softwares. Vak acquisition software runs in the vak-controller that communicates with each sensor in the network and cellular modem for data acquisition and transmission purposes. Vak-reception software is installed in the ocean cube server that collects all the observations data, processes it and then stores it in the ocean cube database. The real-time data transmission interval is user defined and currently set at 15 minutes interval.

Grafana, a web-application is used for interactive data visualization and analytics. As Grafana is an open-source platform, users can create several custom dashboards for data visualizations as per their needs. ***Figure 2.4.3*** shows CTD, wind, and tide data dashboards.

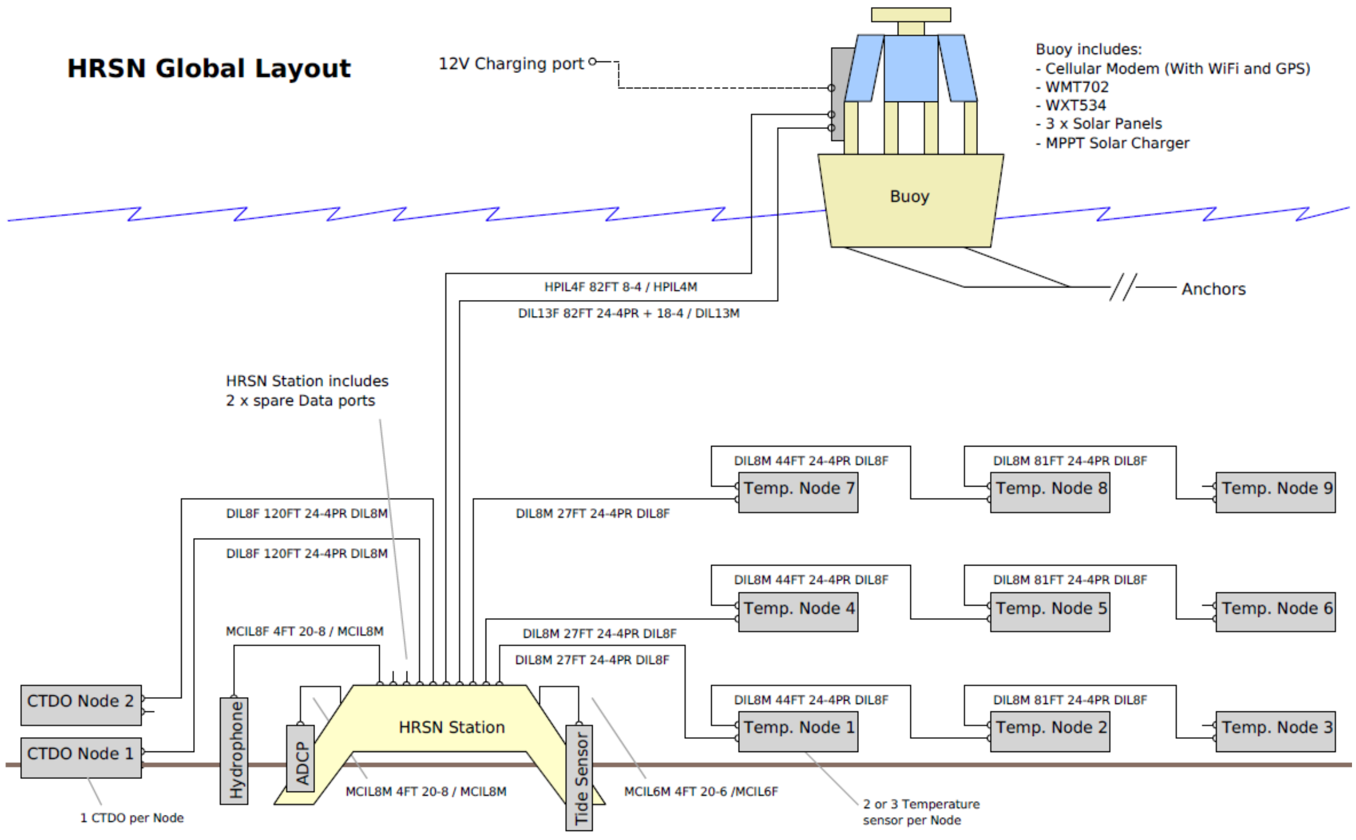


Figure 2.4.1: HRSN architecture

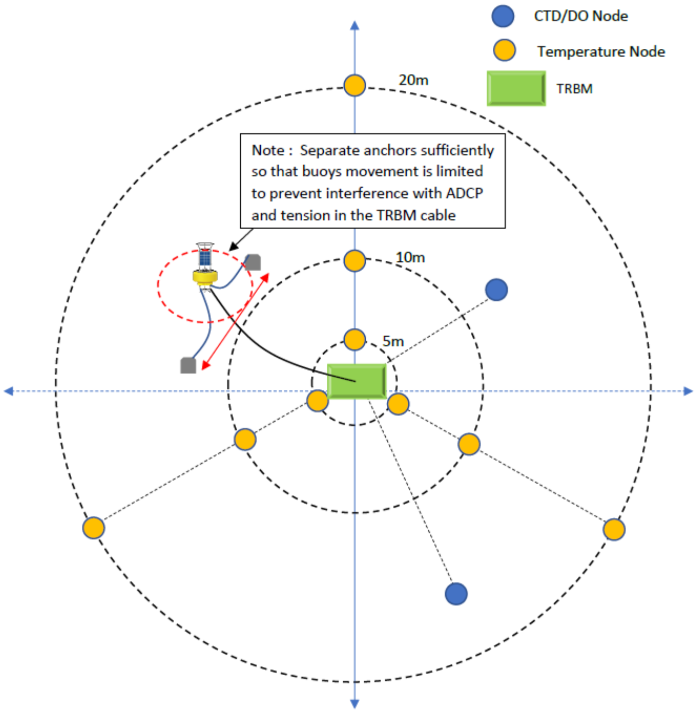

CTD/DO Node
Temperature Node
TRBM
20m
Note : Separate anchors sufficiently so that buoys movement is limited to prevent interference with ADCP and tension in the TRBM cable
10m
5m

Figure 2.4.2: HRSN node layout

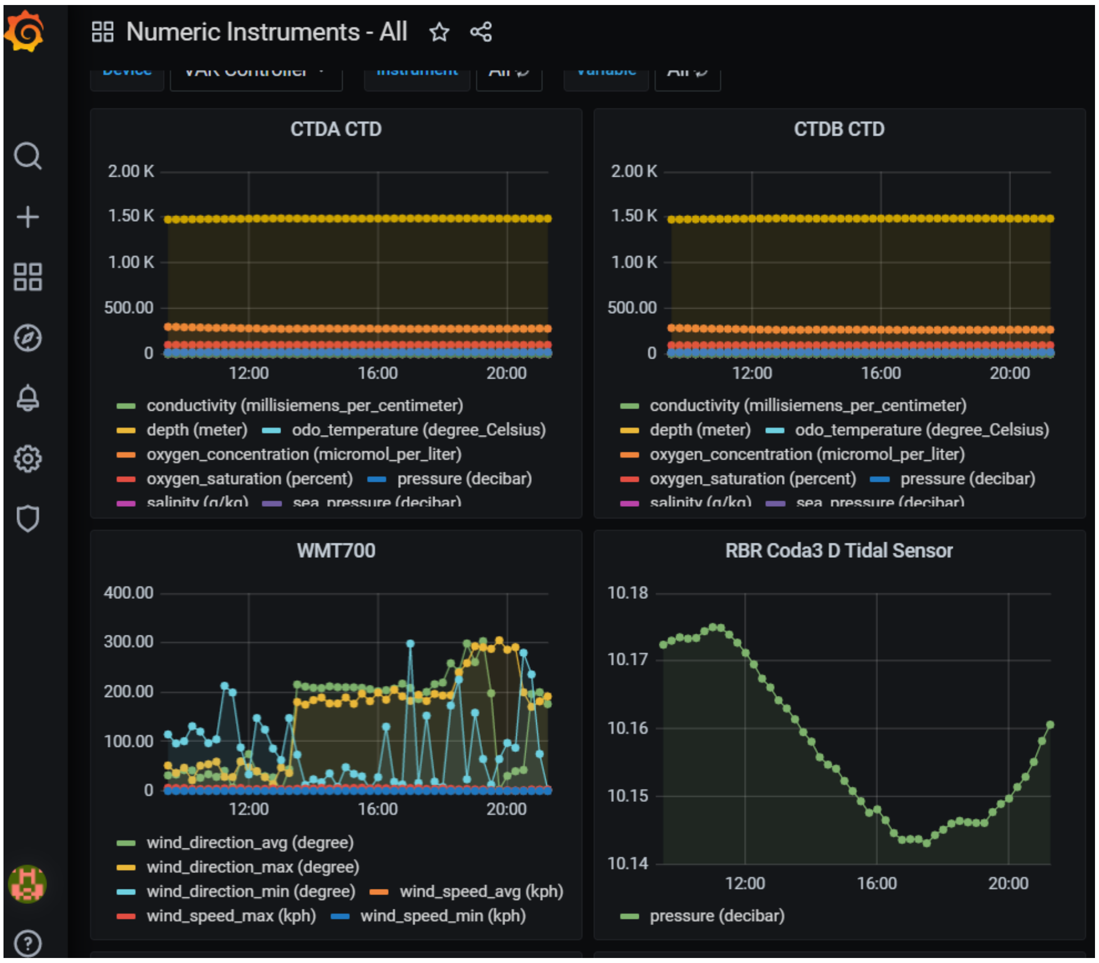


Figure 2.4.3: Grafana web application interface

3. Toolkit for Importing the Nearshore Ocean Cube Data:

Ocean cube observations data is stored in the ocean cube database as the data is ingested by several observation nodes. This stored data can be used for visualization and analysis purposes. Users will be able to import ocean cube observations data from the ocean cube web browser, and users will also be able to visualize and analyze the data using Grafana.

The observations data files collected from several observation nodes is stored in the ocean cube server at their corresponding dedicated folders. These files as per their data format provided by the corresponding manufacturers are processed and then the processed data is stored in the ocean cube database. A loader program is created for each observation node to load the ocean cube database with the corresponding data from each observation node. The data stored in the ocean cube database is available for retrieval depending on the SQL queries made to the database from the ocean cube web interface.

The loader programs are written in widely used programming language called Python. The ocean cube server hosts a Linux distribution and has Python 2.7.17 version installed. Each loader program takes the path of data folder, database username and password as inputs. Each loader program runs through the data files stored in the specified folder, extracts the needed data and updates the data in database. Depending on the data transmission frequency of the observation node, each loader program will be executed accordingly. In each run of the loader program, the tables and columns of corresponding observation node in the database are updated. The loader programs do not process the data files that are already existing in the database as all the data is time stamped. The loader programs codes for WaveRider, Viking, and Iver are presented in the appendices A-C respectively.

Appendix A:

WaveRider_Loader.py

```
'''

Created on March 17, 2021

This is the loader program for WaveRider Data Buoy. This program takes path to the folder where
WaveRider data files are collected, database username and password as inputs.

@author: vishwa sunkara.

Email: Vishwamithra.sunkara@usm.edu

'''

import json

import os

import sys

import mysql.connector

import math

import time

import datetime

from datetime import import date



def getplatforms(cur):

    #

    #Read in the platform table and create a dictionary of platformIds with the platform name as the
key

    #

    sql="SELECT * FROM platform;"

    cur.execute(sql)

    recs=cur.fetchall()

    platforms={}

    for rec in recs:

        platforms[rec[1]]=rec[0]

    return platforms
```

```
def readFB0(filename,platforms):
    #
    #Read the "*{0xFB0}*.csv" file and create an array of dictionaries (one for each line or record in
the file)
    #containing only the data needed in the database tables
    #

    f=open(filename,'r')
    waves_records=[]
    currents_records=[]
    platform_records=[]
    ctd_records=[]
    #check if the file is empty
    if((os.stat(filename).st_size)==0):
        print("empty {0xFB0} file")
        return records
    for line in f:
        line = line.rstrip()
        waves_rec={}
        currents_rec={}
        platform_rec={}
        ctd_rec={}
        fields = line.split("\t")
        #
        #file is empty with new lines when no data is recorded
        #
        if fields[0] == '\n':
            print("no data except newlines in {0xFB0} file")
            return records
        waves_rec["platformId"]=int(platforms[filename[:filename.find('{')]])
        currents_rec["platformId"]=int(platforms[filename[:filename.find('{')]])
        platform_rec["platformId"]=int(platforms[filename[:filename.find('{')]])
        ctd_rec["platformId"]=int(platforms[filename[:filename.find('{')]])

        timestamp=time.gmtime(float(fields[0]))
        waves_rec["time"] = str(timestamp.tm_year) + '-' + str(timestamp.tm_mon)+ '-' +
str(timestamp.tm_mday) + ' ' + str(timestamp.tm_hour) + ':' + str(timestamp.tm_min) + ':' +
str(timestamp.tm_sec)
        lat = float(fields[9])*180/3.14159#lat rad to degrees
        waves_rec["latitude"] = lat
        lon = float(fields[10].replace('\n',''))*180/3.14159#long rad to degrees
        waves_rec["longitude"] = lon
        currents_rec["time"] = waves_rec["time"]
        platform_rec["time"] = waves_rec["time"]
        ctd_rec["time"] = waves_rec["time"]

        currents_rec["latitude"] = waves_rec["latitude"]
        platform_rec["latitude"] = waves_rec["latitude"]
        ctd_rec["latitude"] = waves_rec["latitude"]

        currents_rec["longitude"] = waves_rec["longitude"]
        platform_rec["longitude"] = waves_rec["longitude"]
        ctd_rec["longitude"] = waves_rec["longitude"]



        waves_rec["height"] = float(fields[2])#significant wave height
        waves_rec["period"] = float(fields[3])#mean wave period
        waves_rec["direction"] = float(fields[6])*180/3.14159#wave direction rad to deg conversion
        waves_rec["speed"] = 0.0#float("NaN")
        waves_rec["depth"] = 0.0

        ctd_rec["temperature"] = float(fields[12]) - 273.15# kelvin to deg celcius
        ctd_rec["conductivity"] = None#float("NaN")
        ctd_rec["salinity"] = None#float("NaN")
        ctd_rec["depth"] = None#float("NaN")



        currents_rec["speed"] = float(fields[13])#current speed m/s
        currents_rec["direction"] = float(fields[14])*180/3.1459#current direction rad to deg conversion
        currents_rec["depth"] = 0.0

        platform_rec["depth"] = 0.0
        platform_rec["speed"] = 0.0
        platform_rec["heading"] = 0.0



        waves_records.append(waves_rec)
        currents_records.append(currents_rec)
        platform_records.append(platform_rec)
        ctd_records.append(ctd_rec)



    f.close()
    return waves_records,currents_records,platform_records,ctd_records


def readF82(filename,platforms):
    #
    #Read the "*{0xF82}*.csv" file and create an array of dictionaries (one for each line or record in
the file)
    #containing only the data needed in the database tables
    #


    f=open(filename,'r')
    records=[]

    #check if the file is empty

    if((os.stat(filename).st_size)==0):

        print("empty {0xF82} file")

        return records

    for line in f:

        line=line.rstrip()

        rec={}

        fields = line.split("\t")

        #

        #file is empty with new lines when no data is recorded

        #

        if fields[0] == '\n':

            print("no data except newlines in {0xF82} file")

            return records

        rec["platformId"]=int(platforms[filename[:filename.find('{')]])

        timestamp=time.gmtime(float(fields[0]))

        rec["time"] = str(timestamp.tm_year) + '-' + str(timestamp.tm_mon)+ '-' +
str(timestamp.tm_mday) + ' ' + str(timestamp.tm_hour) + ':' + str(timestamp.tm_min) + ':' +
str(timestamp.tm_sec)

        rec["speed"] = float(fields[3])#currents speed m/s

        rec["direction"] = float(fields[4])*180/3.1417#currents direction rad to deg conversion

        rec["depth"] = 0.0

        rec["temperature"] = float(fields[10])-273.15#sea surface water temperature Kelvin to deg
celcius conversion

        rec["conductivity"] = None#float("NaN")

        rec["salinity"] = None#float("NaN")

        records.append(rec)

    f.close()

    return records
```

```
def readF80(filename,platforms):
    #
    #Read the "*{0xF80}*.csv" file and create an array of dictionaries (one for each line or record in
the file)
    #containing only the data needed in the database tables
    #

    f=open(filename,'r')
    records=[]
    #check if the file is empty
    if((os.stat(filename).st_size)==0):
        print("empty {0xF80} file")
        return records
    for line in f:
        rec={}
        fields = line.split("\t")
        #
        #file is empty with new lines when no data is recorded
        #
        if fields[0] == '\n':
            print("no data except newlines in {0xF80} file")
            return records
        rec["platformId"]=int(platforms[filename[:filename.find('{')]])
        timestamp=time.gmtime(float(fields[0]))
        rec["time"] = str(timestamp.tm_year) + '-' + str(timestamp.tm_mon)+ '-' +
str(timestamp.tm_mday) + ' ' + str(timestamp.tm_hour) + ':' + str(timestamp.tm_min) + ':' +
str(timestamp.tm_sec)
        lat = float(fields[2])*180/3.14159#lat rad to degrees
        rec["latitude"] = lat
        lon = float(fields[3].replace('\n',''))*180/3.14159#long rad to degrees
        rec["longitude"] = lon
        rec["depth"] = 0.0
        rec["heading"] = 0.0#float(NaN)
        rec["speed"] = 0.0#float(NaN)
        records.append(rec)
    f.close()
    return records


def readF25(filename,platforms):
    #
    #Read the "*{0xF25}*.csv" file and create an array of dictionaries (one for each line or record in
the file)
    #containing only the data needed in the database tables
    #

    f=open(filename,'r')
    records=[]
    #check if the file is empty
    if((os.stat(filename).st_size)==0):
        print("empty {0xF25} file")
        return records
    for line in f:
        rec={}
        fields = line.split("\t")
        #
        #file is empty with new lines when no data is recorded
        #
        if fields[0] == '\n':
            print("no data except newlines in {0xF25} file")
            return records
        rec["platformId"]=int(platforms[filename[:filename.find('{')]])
        timestamp=time.gmtime(float(fields[0]))
        rec["time"] = str(timestamp.tm_year) + '-' + str(timestamp.tm_mon)+ '-' +
str(timestamp.tm_mday) + ' ' + str(timestamp.tm_hour) + ':' + str(timestamp.tm_min) + ':' +
str(timestamp.tm_sec)

        rec["height"] = float(fields[3])#wave height in m

        rec["period"] = float(fields[6])#wave period in s

        rec["direction"] = float(fields[13])*180/3.14159#wave direction in deg


        records.append(rec)

    f.close()

    return records


#

#This is the main program.  It requires 3 arguments on the command line.

#1st argument is a directory name where the Viking (USMR1) data files reside. This program will
open all the

#  source files into a dictionary array (a dictionary for each row) maintaining only the values needed
for the load.

#2nd argument is the database username needed to establish the connection.

#3rd argument is the password.

#

#The following line makes the database connection

#



con =
mysql.connector.connect(host="localhost",user=sys.argv[2],passwd=sys.argv[3],database="oceancu
be")

cur=con.cursor()

#

#Read in the platform table and create a dictionary of platformIds with the platform name as the key

#

platforms=getplatforms(cur)

#

#These are the list of fields in the 3 tables that are populated by the SD records.
```

```
#
metfields=["time","latitude","longitude","windSpeed","windDirection","temperature","humidity","p
ressure","altitude","platformId"]
currentsfields=["time","latitude","longitude","speed","direction","depth","platformId"]
wavesfields=["time","latitude","longitude","period","height","platformId"]
#
#These are the list of fields in the 3 tables that are populated by the METEOCE and WDATA records.
#
pHfields=["time","latitude","longitude","depth","pH","platformId"]
platformfields=["time","latitude","longitude","depth","speed","heading","platformId"]
CTDfields=["time","latitude","longitude","conductivity","temperature","salinity","depth","platformI
d"]
#
#These are the list of fields in the 1 table that are populated by the TRIPLET records.
#
chlorofields=["time","latitude","longitude","depth","chlorophyll","platformId"]
#
#The argument is the path to the checkins directory (where all the .dat files are stored)
#
checkinsdir=sys.argv[1]
os.chdir(checkinsdir)
yyyy_folders=os.listdir(checkinsdir)
for yyyy_folder in yyyy_folders:
   mm_folders = os.listdir(checkinsdir + '/' + yyyy_folder)
   for mm_folder in mm_folders:
      files = os.listdir(checkinsdir + '/' + yyyy_folder + '/' + mm_folder)
      os.chdir(checkinsdir + '/' + yyyy_folder + '/' + mm_folder)
      files = sorted(files,reverse=True)
      #
      #flag variables to indicate if any of these files exist or not
      #
      FB0_flag=0
```

```
F82_flag=0

F80_flag=0

F25_flag=0

#

for filename in files:

   #

   #Only process the .csv files

   #

   if filename[-4:] != ".csv":continue

   #

   #print(filename)

   #

   if filename.find("{0xFB0}") != -1:


      #

      #raise FB0 flag to skip reading other files except F83file

      #

      FB0_flag = 1

      #


      #

      #see if the platform name exists in the platform table

      #

      p=filename[:filename.find("{")]

      try:

         platformId=platforms[p]

      except:

         sql="INSERT INTO platform (name) VALUES ('"+p+"');"

         cur.execute(sql)

         con.commit()

         #
         #Read in the platform table and create a dictionary of platformIds with the platform
name as the key
         #
         platforms=getplatforms(cur)
         #
      #
      #Read the "{0xF80}" file and return a list of dictionaries (one for each row)
      #
      waves_recs,currents_recs,platform_recs,CTD_recs=readFB0(filename,platforms)



      #
      if (platform_recs==[]):continue
      #
      #check for duplicates. Check if a platformPosition record with the same time and platform
of the first row exists
      #
      sql="SELECT * FROM platformPosition WHERE
platformId="+str(platform_recs[0]["platformId"])+" AND time='"+platform_recs[0]["time"]+"';"
      cur.execute(sql)

      duplicate=cur.fetchall()
      if len(duplicate)>0:continue
      for rec in platform_recs:
         #
         #create the platformPosition insert sql command for each row
         #
         fields="("
         values="("
         for field in platformfields:
            value=rec[field]
            if value is None:continue
```

```
                if isinstance(value,basestring):
                    values+="'"+value+"',"
                else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            values=values[:-1]+")"
            sql="insert into platformPosition "+fields+" VALUES "+values+";"
            #
            #insert the platformPosition record
            #
            cur.execute(sql)
        if (waves_recs==[]):continue
        #
        #check for duplicates. Check if a waves record with the same time and platform of the first row exists
        #
        sql="SELECT * FROM waves WHERE platformId="+str(waves_recs[0]["platformId"])+" AND time='"+waves_recs[0]["time"]+"';"
        cur.execute(sql)


        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in waves_recs:
            #
            #create the waves insert sql command for each row
            #
            fields="("
            values="("
            for field in wavesfields:
                value=rec[field]
                if value is None:continue
                if isinstance(value,basestring):
                    values+="'"+value+"',"
                else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            values=values[:-1]+")"
            sql="insert into waves "+fields+" VALUES "+values+";"
            #
            #insert the waves record
            #
            cur.execute(sql)

        if (currents_recs==[]):continue
        #
        #check for duplicates. Check if a currents record with the same time and platform of the
first row exists
        #
        sql="SELECT * FROM currents WHERE platformId="+str(currents_recs[0]["platformId"])+"
AND time='"+currents_recs[0]["time"]+"';"
        cur.execute(sql)

        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in currents_recs:
            #
            #create the currents insert sql command for each row
            #
            fields="("
            values="("
            for field in currentsfields:
                value=rec[field]
```

```
                    if value is None:continue
                    if isinstance(value,basestring):
                        values+="'"+value+"',"
                    else:
                        values+=str(value)+","
                    fields+=field+","
                fields=fields[:-1]+")"
                values=values[:-1]+")"
                sql="insert into currents "+fields+" VALUES "+values+";"
                #
                #insert the currents record
                #
                cur.execute(sql)

            if (CTD_recs==[]):continue
            #
            #check for duplicates. Check if a CTD record with the same time and platform of the first
row exists
            #
            sql="SELECT * FROM CTD WHERE platformId="+str(CTD_recs[0]["platformId"])+" AND
time='"+CTD_recs[0]["time"]+"';"
            cur.execute(sql)

            duplicate=cur.fetchall()
            if len(duplicate)>0:continue
            for rec in CTD_recs:
                #
                #create the currents insert sql command for each row
                #
                fields="("
                values="("
                for field in CTDfields:
```

```
          value=rec[field]
          if value is None:continue
          if isinstance(value,basestring):
            values+="'"+value+"',"
          else:
            values+=str(value)+","
          fields+=field+","
        fields=fields[:-1]+")"
        values=values[:-1]+")"
        sql="insert into CTD "+fields+" VALUES "+values+";"
        #
        #insert the CTD record
        #
        cur.execute(sql)
    elif FB0_flag == 0 and filename.find("{0xF83}") != -1:


      #
      #see if the platform name exists in the platform table
      #
      p=filename[:filename.find("{")]
      try:
        platformId=platforms[p]
      except:
        sql="INSERT INTO platform (name) VALUES ('"+p+"');"
        cur.execute(sql)
        con.commit()
        #
        #Read in the platform table and create a dictionary of platformIds with the platform
name as the key
        #
        platforms=getplatforms(cur)
        #
      #
      #Read the "{0xF83}" file and return a list of dictionaries (one for each row)
      #
      F83recs=readF83(filename,platforms)
      if F83recs:
        F83_flag=1
    elif FB0_flag == 0 and filename.find("{0xF82}") != -1:
      #
      #see if the platform name exists in the platform table
      #
      p=filename[:filename.find("{")]
      try:
        platformId=platforms[p]
      except:
        sql="INSERT INTO platform (name) VALUES ('"+p+"');"
        cur.execute(sql)
        con.commit()
        #
        #Read in the platform table and create a dictionary of platformIds with the platform
name as the key
        #
        platforms=getplatforms(cur)
        #
      #
      #Read the "{0xF82}" file and return a list of dictionaries (one for each row)
      #
      F82recs=readF82(filename,platforms)
      if F82recs:
        F82_flag=1
    elif FB0_flag == 0 and filename.find("{0xF80}") != -1:
      #
      #see if the platform name exists in the platform table
      #
      p=filename[:filename.find("{")]
      try:
        platformId=platforms[p]
      except:
        sql="INSERT INTO platform (name) VALUES ('"+p+"');"
        cur.execute(sql)
        con.commit()
        #
        #Read in the platform table and create a dictionary of platformIds with the platform
name as the key
        #
        platforms=getplatforms(cur)
        #
      #
      #Read the "{0xF80}" file and return a list of dictionaries (one for each row)
      #
      F80recs=readF80(filename,platforms)
      if F80recs:
        F80_flag=1
      #print(F80recs)
    elif FB0_flag == 0 and filename.find("{0xF25}") != -1:
      #
      #see if the platform name exists in the platform table
      #
      p=filename[:filename.find("{")]
      try:
        platformId=platforms[p]
      except:
              sql="INSERT INTO platform (name) VALUES ('"+p+"');"
              cur.execute(sql)
              con.commit()
              #
              #Read in the platform table and create a dictionary of platformIds with the platform
name as the key
              #
              platforms=getplatforms(cur)
              #
          #
          #Read the "{0xF25}" file and return a list of dictionaries (one for each row)
          #
          F25recs=readF25(filename,platforms)
          if F25recs:
              F25_flag=1
          #print(F25recs)
      if FB0_flag == 0:


        if F80_flag == 1 and F82_flag ==1:
          #
          #check for duplicates. Check if a currents record with the same time and platform of the
first row exists
          #this condition also applies for all other records
          #
          sql="SELECT * FROM currents WHERE platformId="+str(F82recs[0]["platformId"])+" AND
time='"+F82recs[0]["time"]+"';"
          cur.execute(sql)
          duplicate=cur.fetchall()
          if len(duplicate) == 0:
              for row,rec in enumerate(F82recs):
                  if row>=len(F80recs):
                      F82recs[row]["latitude"] = F80recs[len(F80recs)-1]["latitude"]
```

```
   F82recs[row]["longitude"] = F80recs[len(F80recs)-1]["longitude"]
else:
   F82recs[row]["latitude"] = F80recs[row]["latitude"]
   F82recs[row]["longitude"] = F80recs[row]["longitude"]
fields="("
values="("
#
#create the currents insert sql command for each row
#
for field in currentsfields:
   value=rec[field]
   if value is None:continue
   if isinstance(value,basestring):
      values+="'"+value+"',"
   else:
      values+=str(value)+","
   fields+=field+","
fields=fields[:-1]+")"
values=values[:-1]+")"
sql="insert into currents "+fields+" VALUES "+values+";"

fields="("
values="("
#
#create the CTD insert sql command for each row
#
for field in CTDfields:
   value=rec[field]
   if value is None:continue
   if isinstance(value,basestring):
      values+="'"+value+"',"
```

```
                    else:
                        values+=str(value)+","
                    fields+=field+","
                fields=fields[:-1]+")"
                values=values[:-1]+")"
                sql="insert into CTD "+fields+" VALUES "+values+";"
                cur.execute(sql)
    if F80_flag ==1 and F25_flag ==1:
        #check for duplicates. Check if a waves record with the same time and platform of the first
row exists
        #this condition also applies for all other records
        #
        sql="SELECT * FROM waves WHERE platformId="+str(F25recs[0]["platformId"])+" AND
time='"+F25recs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate) ==0:
            for row,rec in enumerate(F25recs):
                if row>=len(F80recs):
                    F25recs[row]["latitude"] = F80recs[len(F80recs)-1]["latitude"]
                    F25recs[row]["longitude"] = F80recs[len(F80recs)-1]["longitude"]
                else:
                    F25recs[row]["latitude"] = F80recs[row]["latitude"]
                    F25recs[row]["longitude"] = F80recs[row]["longitude"]
                fields="("
                values="("
                #
                #create the waves insert sql command for each row
                #
                for field in wavesfields:
                    value=rec[field]
                    if value is None:continue
```

```
              if isinstance(value,basestring):
                values+="'"+value+"',"
              else:
                values+=str(value)+","
              fields+=field+","
            fields=fields[:-1]+")"
            values=values[:-1]+")"
            sql="insert into waves "+fields+" VALUES "+values+";"
            #print sql
      if F80_flag ==1:
        #check for duplicates. Check if a platformPosition record with the same time and platform
of the first row exists
        #this condition also applies for all other records
        #
        sql="SELECT * FROM waves WHERE platformId="+str(F80recs[0]["platformId"])+" AND
time='"+F80recs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate) ==0 and F80_flag == 1:
          for row,rec in enumerate(F80recs):
            fields="("
            values="("
            #
            #create the platformPosition insert sql command for each row
            #
            for field in platformfields:
              value=rec[field]
              if value is None:continue
              if isinstance(value,basestring):
                values+="'"+value+"',"
              else:
                values+=str(value)+","
```

```
                        fields+=field+","
                    fields=fields[:-1]+")"
                    values=values[:-1]+")"
                    sql="insert into platformPosition "+fields+" VALUES "+values+";"
#
#It doesn't really happen until we do a commit
#

con.commit()
print "Finished"
```

Appendix B:

Viking_Loader.py

```
'''
Created on Feb 23, 2021

@author: vishwa
'''
import json
import os
import sys
import mysql.connector
import math
def getplatforms(cur):
    #
    #Read in the platform table and create a dictionary of platformIds with the platform name as the key
    #
    sql="SELECT * FROM platform;"
    cur.execute(sql)
    recs=cur.fetchall()
    platforms={}
    for rec in recs:
        platforms[rec[1]]=rec[0]
    return platforms


def readSD(filename,platforms):
    #
    #Read the "*_SD*.dat" file and create an array of dictionaries (one for each line or record in the
file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    #print("filename SD?")
    #print(filename)
    #print("PLATFORMS INSIDE readSD = ")
    #print(platforms)
    records=[]
    for line in f:
        line=line.rstrip()
        rec={}
        fields=line.split(",")
        #print(fields)
        rec["platformId"]=int(platforms[fields[0].replace('\xef\xbb\xbf','')])
        if fields[1][0] == '#':
            rec["time"] = float("NaN")
        else:
            rec["time"]=fields[1].replace ('/','-')+" "+fields[2]
        if fields[3][0]=='#':
            rec["latitude"] = float("NaN")
        else:
            degmin=fields[3].split()
            lat=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            if degmin[1][-1]=="S":lat=-lat
            rec["latitude"]=lat
        if fields[4][0] == '#':
            rec["longitude"] = float("NaN")
        else:
            degmin=fields[4].split()
            lon=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            if degmin[1][-1]=="W":lon=-lon
            rec["longitude"]=lon
        if fields[5][0] == '#':
            rec["windSpeed"] = float("NaN")
        else:
            rec["windSpeed"]=float(fields[5])*0.51444444444444 # convert knots to meters/second
        if fields[6][0] == '#':
            rec["windMaxSpeed"] = float("NaN")
        else:
            rec["windMaxSpeed"]=float(fields[6])*0.51444444444444 # convert knots to meters/second
        if fields[7][0] == '#':
            rec["windDirection"]=999
        else:
            rec["windDirection"]=int(fields[7])
        if fields[8][0] == '#':
            rec["temperature"]=float("NaN")
        else:
            rec["temperature"]=(float(fields[8])-32.)*5./9. #Fahrenheit to Celsius
        if fields[9][0] == '#':
```

```
        rec["humidity"]=float("NaN")
    else:
        rec["humidity"]=float(fields[9])
    if fields[10][0] == '#':
        rec["pressure"]=float("NaN")
    else:
        rec["pressure"]=float(fields[10])
    #
    #A '#' in the field indicates the data does not exists.  Replace the value with a NaN for floats and 
999 for integers
    #
    if fields[11][0]=='#':
        rec["period"]=float("NaN")
    else:
        rec["period"]=float(fields[11])
    if fields[12][0]=='#':
        rec["height"]=float("NaN")
    else:
        rec["height"]=float(fields[12])
    if fields[13][0]=='#':
        rec["peak"]=float("NaN")
    else:
        rec["peakHeight"]=float(fields[13])
    if fields[24][0]=='#':
        rec["speed"]=float("NaN")
    else:
        rec["speed"]=float(fields[24])
    if fields[25][0]=='#':
        rec["direction"]=999
    else:
        rec["direction"]=int(fields[25])
        rec["depth"]=float(0.)
        rec["altitude"]=float(0.)
        records.append(rec)
    f.close()
    return records

def readADCP(filename,platforms,lat,lon):
    #
    #Read the "*_ADCP*.dat" file and create an array of dictionaries (one for each line or record in the file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    #print("filename ADCP?")
    #print(filename)
    #print("PLATFORMS INSIDE readADCP = ")
    #print(platforms)
    records=[]
    for position,line in enumerate(f):
        line=line.rstrip()
        #rec={}
        fields=line.split(",")
        #print(fields)
        #print(position)

        for position1,field in enumerate(fields):
            rec={}
            rec["platformId"]=int(platforms[filename[:filename.find("_")]])
            if position1 == 0:
                # if fields[0][0] == '#':
                #     rec["time"] = float("NaN")
                # else:
                #     rec["time"]=(fields[0].replace('\xef\xbb\xbf','')).replace ('/','-')+" "+fields[1]
                individual_bin_height = float(fields[3])/100#cm to m conversion
                depth = float(fields[4])/100#cm to m conversion
            if position1%6 == 0.0:
                if fields[0][0] == '#':
                    rec["time"] = float("NaN")
                else:
                    rec["time"]=(fields[0].replace('\xef\xbb\xbf','')).replace ('/','-')+" "+fields[1]
                rec["depth"] = depth#float(fields[4])/100 + (position1/6)*individual_bin_height#cm to m
conversion
                depth = depth + individual_bin_height
                rec["latitude"] = lat
                rec["longitude"] = lon
                if fields[position1+4][0] == '#':
                    rec["speed"] = float("NaN")
                else:
                    rec["speed"] = float(fields[position1+4])
                if fields[position1+5][0] == "#":
                    rec["direction"] = 999
                else:
                    rec["direction"] = int(fields[position1+5])
                records.append(rec)
            #print(rec)
            #print(records)

    f.close()
    return records



def readMETEOCE(filename,platforms):
    #
    #Read the "*_METEOCE_*.dat" file and create an array of dictionaries (one for each line or record
in the file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    #print("filename METEOCE?")
    #print(filename)
    records=[]
    for line in f:
       line=line.rstrip()
       rec={}
       fields=line.split(",")
       #print(fields)
       rec["platformId"]=int(platforms[filename[:filename.find("_")]])
       rec["time"]=fields[0].replace ('/','-')+" "+fields[1]
       degmin=fields[2].split()
       #print(degmin)
       lat=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
       if degmin[1][-1]=="S":lat=-lat
       rec["latitude"]=lat
       degmin=fields[3].split()
       lon=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
       if degmin[1][-1]=="W":lon=-lon
       rec["longitude"]=lon
       #
       #A '#' in the field indicates the data does not exists.  Replace the value with a NaN for floats and
999 for integers
       #
       if fields[27][0]=='#':
            rec["pH"]=float("nan")
       else:
            rec["pH"]=float(fields[27])
        rec["depth"]=float(0.)
        rec["altitude"]=float(0.)
        rec["speed"]=float(0.)
        rec["heading"]=float(0.)
        records.append(rec)
    f.close()
    return records


def readTRIPLET(filename,platforms):
    #
    #Read the "*_TRIPLET_*.dat" file and create an array of dictionaries (one for each line or record in the file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    records=[]
    for line in f:
        line=line.rstrip()
        rec={}
        fields=line.split(",")
        rec["platformId"]=int(platforms[filename[:filename.find("_")]])
        rec["time"]=fields[0].replace ('/','-')+" "+fields[1]
        #rec["chlorophyl"]=float(fields[10])#ug/L
        #
        #A '#' in the field indicates the data does not exists.  Replace the value with a NaN for floats and 999 for integers
        #
        if fields[10][0]=='#':
            rec["chlorophyll"]=float("NaN")
        else:
            rec["chlorophyll"]=float(fields[10])
        rec["depth"]=float(0.)
        rec["altitude"]=float(0.)
        records.append(rec)
    f.close()
    return records


def readWDATA(filename,platforms):
    #
    #Read the "*WDATA*.txt" file and create an array of dictionaries (one for each line or record in the file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    #f = f.read().decode("utf-8-sig").encode("utf-8")
    #print("filename WDATA?")
    #print(filename)
    #print("PLATFORMS INSIDE readWDATA = ")
    #print(platforms)
    records=[]
    for position,line in enumerate(f):
        line = line.rstrip()
        rec={}
        if position == 0:
            fields=line.split(" ")
            #print("time fields=")
            #print(fields)
            time=(fields[1].replace ('/','-')).replace('\r\n','')+" "+fields[0].replace('\xef\xbb\xbf','')
            #print("time?")
            #print(time)
            #print(fields[1])
        elif position == 1:
            platform = line.replace('\r\n','')#"USM-R1"
            #print("platform?")
            #print(platform)
            #print(len(platform))

        elif position == 2:
            fields=line.split(",")
            degmin=fields[0].split()
            lat=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            #rec["latitude"]=lat
            if degmin[1][-1]=="S":lat=-lat
            degmin=fields[1].split()
            lon=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            if degmin[1][-1]=="W":lon=-lon
        elif position>3:
            fields=line.split(",")
            if fields==['']:continue
            #print(fields)
            rec["platformId"]=int(platforms[platform])
            #print(rec["platformId"])
            #print("time in line >3")
            #print(time)
            rec["time"]=time
            rec["latitude"]=lat
            rec["longitude"]=lon
            #print("fields=")
            #print(fields)
            #print("fields[0][0]=")
            #print(fields[0][0])
            if fields[3][0] == '#':
                rec["depth"]=float("NaN")#meters
            else:
              rec["depth"]=float(fields[3])#meters
            if fields[1][0] == '#':
              rec["temperature"]=float("NaN")#deg Celius
            else:
              rec["temperature"]=float(fields[1])#meters
            if fields[4][0] == '#':
              rec["salinity"]=float("NaN")#PSU
            else:
              rec["salinity"]=float(fields[4])#PSU
            if fields[0][0] == '#':
              rec["conductivity"]=float("NaN")#
            else:
              rec["conductivity"]=float(fields[0])#
            if fields[3][0] == '#':
              rec["pressure"]=float("NaN")#
            else:
              rec["pressure"]=float(fields[3])#
            #print(rec)
            records.append(rec)
   # if position == 3:#if line count is 4 then there is no ctd data acquired. So, inserting NaNs to
indicate this.
   #     rec["platformId"]=int(platforms[platform])
   #     rec["time"]=time
   #     rec["latitude"]=lat
   #     rec["longitude"]=lon
   #     rec["depth"]=float("NaN")#meters
   #     rec["temperature"]=float("NaN")#deg Celius
   #     rec["salinity"]=float("NaN")
   #     rec["conductivity"]=float("NaN")
   #     rec["density"]=float("NaN")
```

```
    #    records.append(rec)

    f.close()
    return records

def readWDATA_LOWRES(filename,platforms):
    #
    #Read the "*WDATA _LOWRES*.txt" file and create an array of dictionaries (one for each line or record in the file)
    #containing only the data needed in the database tables
    #
    f=open(filename,'r')
    #f = f.read().decode("utf-8-sig").encode("utf-8")
    #print("filename WDATA_LOWRES?")
    #print(filename)
    #print("PLATFORMS INSIDE readWDATA = ")
    #print(platforms)
    records=[]
    for position,line in enumerate(f):
        line = line.rstrip()
        rec={}
        if position == 0:
            fields=line.split(" ")
            #print("time fields=")
            #print(fields)
            time=(fields[1].replace ('/','-')).replace('\r\n','')+" "+fields[0].replace('\xef\xbb\xbf','')
            #print("time?")
            #print(time)
            #print(fields[1])
        elif position == 1:
            platform = line.replace('\r\n','')#"USM-R1"
            # print("platform?")
            # print(platform)
            #print(len(platform))

        elif position == 2:
            fields=line.split(",")
            degmin=fields[0].split()
            lat=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            #rec["latitude"]=lat
            if degmin[1][-1]=="S":lat=-lat
            degmin=fields[1].split()
            lon=float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            if degmin[1][-1]=="W":lon=-lon
        elif position>3:
            fields=line.split(",")
            if fields==['']:continue
            #print(fields)
            rec["platformId"]=int(platforms[platform])
            #print(rec["platformId"])
            #print("time in line >3")
            #print(time)
            rec["time"]=time
            rec["latitude"]=lat
            rec["longitude"]=lon
            # print("fields=")
            # print(fields)
            #print("fields[0][0]=")
            #print(fields[0][0])
            if fields[0][0] == '#':
                rec["temperature"]=float("NaN")#deg Celius
            else:
                rec["temperature"]=float(fields[0])#
            if fields[1][0] == '#':
                rec["depth"]=float("NaN")
            else:
                rec["depth"]=float(fields[1])#meters
            if fields[2][0] == '#':
                rec["salinity"]=float("NaN")#PSU
            else:
                rec["salinity"]=float(fields[2])#PSU
            rec["conductivity"]=None#float("NaN")
            #print(rec)
            records.append(rec)
            #print(records)
        # if position == 3:#if line count is 4 then there is no ctd data acquired. So, inserting NaNs to
    indicate this.
        #    rec["platformId"]=int(platforms[platform])
        #    rec["time"]=time
        #    rec["latitude"]=lat
        #    rec["longitude"]=lon
        #    rec["depth"]=float("NaN")#meters
        #    rec["temperature"]=float("NaN")#deg Celius
        #    rec["salinity"]=float("NaN")
        #    rec["conductivity"]=float("NaN")
        #    rec["density"]=float("NaN")
        #    records.append(rec)

        f.close()
        return records



    #
```

```
#This is the main program.  It requires 3 arguments on the command line.

#1st argument is a directory name where the Viking (USMR1) data files reside. This program will
open all the

#  source files into a dictionary array (a dictionary for each row) maintaining only the values needed
for the load.

#2nd argument is the database username needed to establish the connection.

#3rd argument is the password.

#

#The following line makes the database connection

#


con =
mysql.connector.connect(host="localhost",user=sys.argv[2],passwd=sys.argv[3],database="oceancu
be")

cur=con.cursor()

#

#Read in the platform table and create a dictionary of platformIds with the platform name as the key

#

platforms=getplatforms(cur)

#

#These are the list of fields in the 3 tables that are populated by the SD records.

#

metfields=["time","latitude","longitude","windSpeed","windDirection","temperature","humidity","p
ressure","altitude","platformId"]

currentsfields=["time","latitude","longitude","speed","direction","depth","platformId"]

wavefields=["time","latitude","longitude","period","height","platformId"]

#

#These are the list of fields in the 3 tables that are populated by the METEOCE and WDATA records.

#

pHfields=["time","latitude","longitude","depth","pH","platformId"]

platformfields=["time","latitude","longitude","depth","speed","heading","platformId"]

CTDfields=["time","latitude","longitude","conductivity","temperature","salinity","depth","platformI
d"]
```

```
#
#These are the list of fields in the 1 table that are populated by the TRIPLET records.
#
chlorofields=["time","latitude","longitude","depth","chlorophyll","platformId"]
#
#The argument is the path to the checkins directory (where all the .dat files are stored)
#
checkinsdir=sys.argv[1]
#print checkinsdir
os.chdir(checkinsdir)
files=os.listdir(checkinsdir)
#print("files = ")
#print(files)
files = sorted(files,reverse=True)
print(files)
for filename in files:
   #print(filename)
   #
   #Only process the .dat files
   #
   if filename[-4:] != ".dat":continue
   #
   #only process the "_SD_" files
   #
   if filename.find("_SD_") != -1:
      #
      #see if the platform name exists in the platform table
      #
      p=filename[:filename.find("_")]
      try:
         platformId=platforms[p]
```

```
        except:
            sql="INSERT INTO platform (name) VALUES ('"+p+"');"
            #print sql
            cur.execute(sql)
            con.commit()
            #
            #Read in the platform table and create a dictionary of platformIds with the platform name as
the key
            #
            platforms=getplatforms(cur)
        #
        #Read the "_SD_" file and return a list of dictionaries (one for each row)
        #
        SDrecs=readSD(filename,platforms)
        #print("SDrecs=")
        #print(SDrecs)
        #
        if (SDrecs==[]):continue
        #
        #check for duplicates. Check if a meteorology record with the same time and platform of the
first row exists
        #
        #print("SDrec[0]=")
        #print(SDrecs[0])
        lat = SDrecs[0]["latitude"]
        lon = SDrecs[0]["longitude"]


        sql="SELECT * FROM meteorology WHERE platformId="+str(SDrecs[0]["platformId"])+" AND
time='"+SDrecs[0]["time"]+"';"
        cur.execute(sql)


        duplicate=cur.fetchall()
```

```
    if len(duplicate)>0:continue
    #print("printed?")
    for rec in SDrecs:
      #
      #create the meteorology insert sql command for each row
      #
      fields="("
      values="("
      if not math.isnan(rec["windSpeed"]): #If there was a '#' in the field it was converted to a NaN.
Don't load these records
        for field in metfields:
          value=rec[field]
          if value is None:continue
          if isinstance(value,basestring):
            values+="'"+value+"',"
          else:
            values+=str(value)+","
          fields+=field+","
        fields=fields[:-1]+")"
        #print fields
        values=values[:-1]+")"
        #print values
        sql="insert into meteorology "+fields+" VALUES "+values+";"
        #print("meteorology sql")
        #print sql
        #
        #insert the meteorology record
        #
        cur.execute(sql)
      if not math.isnan(rec["speed"]): #If there was a '#' in the field it was converted to a NaN.
Don't load these records
        #
```

```
                #create the currents insert sql command for each row
                #
                fields="("
                values="("
                for field in currentsfields:
                    value=rec[field]
                    if value is None:continue
                    if isinstance(value,basestring):
                        values+="'"+value+"',"
                    else:
                        values+=str(value)+","
                    fields+=field+","
                fields=fields[:-1]+")"
                #print fields
                values=values[:-1]+")"
                #print values
                sql="insert into currents "+fields+" VALUES "+values+";"
                #print("currents sql")
                #print sql
                #
                #insert the currents record
                #
                cur.execute(sql)
            if not math.isnan(rec["period"]): #If there was a '#' in the field it was converted to a NaN.
Don't load these records
                #
                #create the waves insert sql command for each row
                #
                fields="("
                values="("
                for field in wavefields:
```

```
                value=rec[field]
                if value is None:continue
                if isinstance(value,basestring):
                    values+="'"+value+"',"
                else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            #print fields
            values=values[:-1]+")"
            #print values
            sql="insert into waves "+fields+" VALUES "+values+";"
            #print("waves sql")
            #print sql
            #
            #insert the currents record
            #
            cur.execute(sql)
    elif filename.find("_METEOCE_") != -1:
            #Read the "_METEOCE_" file and return a list of dictionaries (one for each row)
        #
        METEOCErecs=readMETEOCE(filename,platforms)
        #print("METEOCErecs=")
        #print(METEOCErecs)
        #
        if(METEOCErecs==[]): continue
        #check for duplicates. Check if a pH record with the same time and platform of the first row
exists
        #
        sql="SELECT * FROM pH WHERE platformId="+str(METEOCErecs[0]["platformId"])+" AND
time='"+ METEOCErecs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in METEOCErecs:
            fields="("
            values="("
            if not math.isnan(rec["pH"]): #If there was a '#/NaN' in the field it was converted to a NaN.
Don't load these records
            for field in pHfields:
                value=rec[field]
                if value is None:continue
                if isinstance(value,basestring):
                    values+="'"+value+"',"
                else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            #print fields
            values=values[:-1]+")"
            #print values
            sql="insert into pH "+fields+" VALUES "+values+";"
            #print("pH sql")
            #print sql
            #
            #insert the ctd record
            #
            cur.execute(sql)

        #check for duplicates. Check if a platformPosition record with the same time and platform of the
first row exists
        #
        sql="SELECT * FROM platformPosition WHERE
platformId="+str(METEOCErecs[0]["platformId"])+" AND time='"+ METEOCErecs[0]["time"]+"';"
```

```
        cur.execute(sql)

        duplicate=cur.fetchall()

        if len(duplicate)>0:continue

        for rec in METEOCErecs:

          fields="("

          values="("

          if not math.isnan(rec["speed"]): #If there was a '#/NaN' in the field it was converted to a NaN.
Don't load these records

            for field in platformfields:

              value=rec[field]

              if value is None:continue

              if isinstance(value,basestring):

                values+="'"+value+"',"

              else:

                values+=str(value)+","

              fields+=field+","

            fields=fields[:-1]+")"

            #print fields

            values=values[:-1]+")"

            #print values

            sql="insert into platformPosition "+fields+" VALUES "+values+";"

            #print("platform sql")

            #print sql

            #

            #insert the platformPosition record

            #

            cur.execute(sql)

    elif filename.find("_ADCP_") != -1:

            #Read the "_ADCP_" file and return a list of dictionaries (one for each row)

        #

        ADCPrecs=readADCP(filename,platforms,lat,lon)
        #print("ADCPrecs=")
        #print(ADCPrecs)
        #
        if(ADCPrecs==[]): continue
        #check for duplicates. Check if a pH record with the same time and platform of the first row exists
        #
        sql="SELECT * FROM currents WHERE platformId="+str(ADCPrecs[0]["platformId"])+" AND time='"+ ADCPrecs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in ADCPrecs:
            fields="("
            values="("
            if not math.isnan(rec["speed"]): #If there was a '#/NaN' in the field it was converted to a NaN. Don't load these records
                for field in currentsfields:
                    value=rec[field]
                    if value is None:continue
                    if isinstance(value,basestring):
                        values+="'"+value+"',"
                    else:
                        values+=str(value)+","
                    fields+=field+","
                fields=fields[:-1]+")"
                #print fields
                values=values[:-1]+")"
                #print values
                sql="insert into currents "+fields+" VALUES "+values+";"
                #print("currents sql")
                #print sql
            #
            #insert the currents record
            #
            cur.execute(sql)
    elif filename.find("_TRIPLET_") != -1:
           #Read the "_TRIPLET_" file and return a list of dictionaries (one for each row)
        #
        TRIPLETrecs=readTRIPLET(filename,platforms)
        #
        #print("TRIPLETrecs=")
        #print(TRIPLETrecs)
        #
        if(TRIPLETrecs==[]): continue
        #check for duplicates. Check if a chloro record with the same time and platform of the first row
exists
        #
        sql="SELECT * FROM chlorophyll WHERE platformId="+str(TRIPLETrecs[0]["platformId"])+" AND
time='"+ TRIPLETrecs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in TRIPLETrecs:
          fields="("
          values="("
          if not math.isnan(rec["chlorophyll"]): #If there was a '#/NaN' in the field it was converted to a
NaN. Don't load these records
            for field in chlorofields:
              value=rec[field]
              if value is None:continue
              if isinstance(value,basestring):
                values+="'"+value+"',"
              else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            #print fields
            values=values[:-1]+")"
            #print values
            sql="insert into chlorophyll "+fields+" VALUES "+values+";"
            #print("chlorophyll sql")
            #print sql
            #
            #insert the chlorophyll record
            #
            cur.execute(sql)



    #empty file? Skip
        #





#
#The argument is the path to the checkins directory (where all the .dat files are stored we use this
path to get into winch data files folder)
#
checkinsdir=sys.argv[1] + "/WINCH_MISSIONS"
#print("WINCH FILES PATH?")
#print checkinsdir
os.chdir(checkinsdir)
files=os.listdir(checkinsdir)
#print("files = ")
#print(files)
```

```
for filename in files:
    #print(filename[-4:])
    #print(filename.find("WDATA_"))
    #
    #Only process the .txt files
    #
    if filename[-4:] != ".txt":continue
    #
    #only process the "_WDATA_" files
    #
    if filename.find("WDATA_") != -1 and filename.find("LOWRES") !=-1:
        #
        #see if the platform name exists in the platform table
        #
        first_idx = filename.find("_")+1
        second_idx = first_idx + (filename[filename.find("_")+1:]).find("_")
        p=filename[first_idx:second_idx]
        #print("platform name?")
        #print(p)
        try:
            platformId=platforms[p]
        except:
            sql="INSERT INTO platform (name) VALUES ('"+p+"');"
            #print sql
            cur.execute(sql)
            con.commit()
            #
            #Read in the platform table and create a dictionary of platformIds with the platform name as
the key
            #
```

```
          platforms=getplatforms(cur)
        #
        #Read the "WDATA_" file and return a list of dictionaries (one for each row)
        #
        WDATA_LOWRESrecs=readWDATA_LOWRES(filename,platforms)
        #print("WDATA_LOWRESrecs=")
        #print(WDATA_LOWRESrecs)



        if (WDATA_LOWRESrecs==[]):continue
        #
        #check for duplicates. Check if a CTD record with the same time and platform of the first row
exists
        #
        sql="SELECT * FROM CTD WHERE platformId="+str(WDATA_LOWRESrecs[0]["platformId"])+"
AND time='"+WDATA_LOWRESrecs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in WDATA_LOWRESrecs:
          #
          #create the CTD insert sql command for each row
          #
          fields="("
          values="("
          if not math.isnan(rec["temperature"]): #If there was a '#/NaN' in the field it was converted to
a NaN. Don't load these records
            #
            #create the CTD insert sql command for each row
            #
            fields="("
            values="("
            for field in CTDfields:
              value=rec[field]
              if value is None:continue
              if isinstance(value,basestring):
                values+="'"+value+"',"
              else:
                values+=str(value)+","
              fields+=field+","
            fields=fields[:-1]+")"
            #print fields
            values=values[:-1]+")"
            #print values
            sql="insert into CTD "+fields+" VALUES "+values+";"
            #print("CTD sql")
            #print sql
            cur.execute(sql)

      else:
        #
        #see if the platform name exists in the platform table
        #
        first_idx = filename.find("_")+1
        second_idx = first_idx + (filename[filename.find("_")+1:]).find("_")
        p=filename[first_idx:second_idx]
        #print("platform name?")
        #print(p)
        try:
          platformId=platforms[p]
        except:
          sql="INSERT INTO platform (name) VALUES ('"+p+"');"
          #print sql
```

```
            cur.execute(sql)

            con.commit()

            #

            #Read in the platform table and create a dictionary of platformIds with the platform name as
the key

            #

            platforms=getplatforms(cur)

        #

        #Read the "WDATA_" file and return a list of dictionaries (one for each row)

        #

        WDATArecs=readWDATA(filename,platforms)

        #print("WDATArecs=")

        #print(WDATArecs)



        if (WDATArecs==[]):continue

        #

        #check for duplicates. Check if a CTD record with the same time and platform of the first row
exists

        #

        sql="SELECT * FROM CTD WHERE platformId="+str(WDATArecs[0]["platformId"])+" AND
time='"+WDATArecs[0]["time"]+"';"

        cur.execute(sql)

        duplicate=cur.fetchall()

        if len(duplicate)>0:continue

        for rec in WDATArecs:

            #

            #create the CTD insert sql command for each row

            #

            fields="("

            values="("
        if not math.isnan(rec["temperature"]): #If there was a '#/NaN' in the field it was converted to a NaN. Don't load these records
            #
            #create the CTD insert sql command for each row
            #
            fields="("
            values="("
            for field in CTDfields:
                value=rec[field]
                if value is None:continue
                if isinstance(value,basestring):
                    values+="'"+value+"',"
                else:
                    values+=str(value)+","
                fields+=field+","
            fields=fields[:-1]+")"
            #print fields
            values=values[:-1]+")"
            #print values
            sql="insert into CTD "+fields+" VALUES "+values+";"
            #print("CTD sql")
            #print sql
            cur.execute(sql)

#
#
#It doesn't really happen until we do a commit
#

con.commit()
print "Finished"
```

Appendix C:

Iver_Loader.py

```
'''

Created on March 11, 2021

This is the loader program for viking Data Buoy. This program takes path to the folder where
WaveRider data files are collected, database username and password as inputs.

@author: vishwa sunkara.

Email: Vishwamithra.sunkara@usm.edu

'''

import json

import os

import sys

import mysql.connector

import math

def getplatforms(cur):
   #
   #Read in the platform table and create a dictionary of platformIds with the platform name as the key
   #
   sql="SELECT * FROM platform;"
   cur.execute(sql)
   recs=cur.fetchall()
   platforms={}
   for rec in recs:
      platforms[rec[1]]=rec[0]
   return platforms

def readIVER(filename,platforms):
   #
   #Read the "*IVER*.log" file and create an array of dictionaries (one for each line or record in the file)
   #containing only the data needed in the database tables
   #
   f=open(filename,'r')
   records=[]
   for position,line in enumerate(f):
      rec={}
      if position>0:
         fields=line.split(";")
         if not fields:continue
         rec["platformId"]=int(platforms[filename[filename.rindex("I"):len(filename)-4]])
         #print("IVER3-3072?")
         #print(filename[filename.rindex("I"):len(filename)-4])
         rec["time"]=fields[3].replace ('/','-')+" "+fields[2]
         #degmin=fields[0].split()
         lat=fields[0]#float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
         rec["latitude"]=lat
```

```
            #if degmin[1][-1]=="S":lat=-lat
            #degmin=fields[1].split()
            lon=fields[1]#float(degmin[0])+float(degmin[1][:-1])/60. #degrees minutes to degrees
            #if degmin[1][-1]=="W":lon=-lon
            rec["longitude"]=lon
            rec["speed"]=float(fields[26])*0.51444444444444 # convert knots to meters/second
            rec["temperature"]=float(fields[52]) #Celsius
            rec["heading"]=float(fields[10])# compass true heading in deg
            rec["depth"]=float(fields[14])#DFS-depth from surface
            rec["altitude"]=float(fields[16])#HFB-Height from bottom
            rec["salinity"]=float(fields[55])#ppt
            rec["conductivity"]=float(fields[53])#mS/cm
            records.append(rec)
    f.close()
    return records


#
#This is the main program.  It requires 3 arguments on the command line.
#1st argument is a directory name where the source (*IVER*.log) files reside.  This program will open all the
#  source files into a dictionary array (a dictionary for each row) maintaining onny the values needed for the load.
#2nd argument is the database username needed to establish the connection.
#3rd argument is the password.
#
#The following line makes the database connection
#


con = mysql.connector.connect(host="localhost",user=sys.argv[2],passwd=sys.argv[3],database="oceancube")
```

```
cur=con.cursor()

#

#Read in the platform table and create a dictionary of platformIds with the platform name as the key

#

platforms=getplatforms(cur)

#

#These are the list of fields in tables that cab be populated by IVER records.

#

#metfields=["time","latitude","longitude","windSpeed","windDirection","temperature","humidity","
pressure","altitude","platformId"]

#currentfields=["time","latitude","longitude","speed","direction","depth","platformId"]

#wavefields=["time","latitude","longitude","period","height","platformId"]

platformfields=["time","latitude","longitude","depth","speed","heading","platformId"]

CTDfields=["time","latitude","longitude","conductivity","temperature","salinity","depth","platformI
d"]

#

#The argument is the path to the checkins directory (where all the .log files are stored)

#

checkinsdir=sys.argv[1]

#print checkinsdir

os.chdir(checkinsdir)

files=os.listdir(checkinsdir)

for filename in files:

   #

   #Only process the .log files

   #

   if filename[-4:] != ".log":continue

   #

   #only process the "IVER log" files

   #

   if filename.find("IVER") != -1:

      #
```

```
    #see if the platform name exists in the platform table

    #

    p=filename[filename.rindex("I"):len(filename)-4]

    #print("IVER3-3072?")

    #print(filename[filename.rindex("I"):len(filename)-4])

    try:

      platformId=platforms[p]

    except:

      sql="INSERT INTO platform (name) VALUES ('"+p+"');"

      print sql

      cur.execute(sql)

      con.commit()

      #

      #Read in the platform table and create a dictionary of platformIds with the platform name as
the key

      #

      platforms=getplatforms(cur)

    #

    #Read the file and return a list of dictionaries (one for each row)

    #

    IVERrecs=readIVER(filename,platforms)

    #print("IVERrecs?")

    #print(IVERrecs)

    #

    #empty file? Skip

    #

    if IVERrecs==[]:continue

    #

    #check for duplicates. Check if a CTD record with the same time and platform of the first row
exists

    #
```

```
        sql="SELECT * FROM CTD WHERE platformId="+str(IVERrecs[0]["platformId"])+" AND
time='"+IVERrecs[0]["time"]+"';"

        cur.execute(sql)

        duplicate=cur.fetchall()

        if len(duplicate)>0:continue

        for rec in IVERrecs:

           #

           #create the CTD insert sql command for each row

           #

           fields="("

           values="("

           for field in CTDfields:

              value=rec[field]

              if value is None:continue

              if isinstance(value,basestring):

                 values+="'"+value+"',"

              else:

                 values+=str(value)+","

              fields+=field+","

           fields=fields[:-1]+")"

           #print fields

           values=values[:-1]+")"

           #print values

           sql="insert into CTD "+fields+" VALUES "+values+";"

           #print sql

           #

           #insert the CTD record

           #

           cur.execute(sql)

        #

        #check for duplicates. Check if a platformPosition record with the same time and platform of the
first row exists
```

```
        #
        sql="SELECT * FROM platformPosition WHERE platformId="+str(IVERrecs[0]["platformId"])+"
AND time='"+IVERrecs[0]["time"]+"';"
        cur.execute(sql)
        duplicate=cur.fetchall()
        if len(duplicate)>0:continue
        for rec in IVERrecs:
          #
          #create the platformPosition insert sql command for each row
          #
          fields="("
          values="("
          for field in platformfields:
            value=rec[field]
            if value is None:continue
            if isinstance(value,basestring):
              values+="'"+value+"',"
            else:
              values+=str(value)+","
            fields+=field+","
          fields=fields[:-1]+")"
          #print fields
          values=values[:-1]+")"
          #print values
          sql="insert into platformPosition "+fields+" VALUES "+values+";"
          #print sql
          #
          #insert the platformPosition record
          #
          cur.execute(sql)
#
```

```
#It doesn't really happen until we do a commit

#

con.commit()

print "Finished"
```