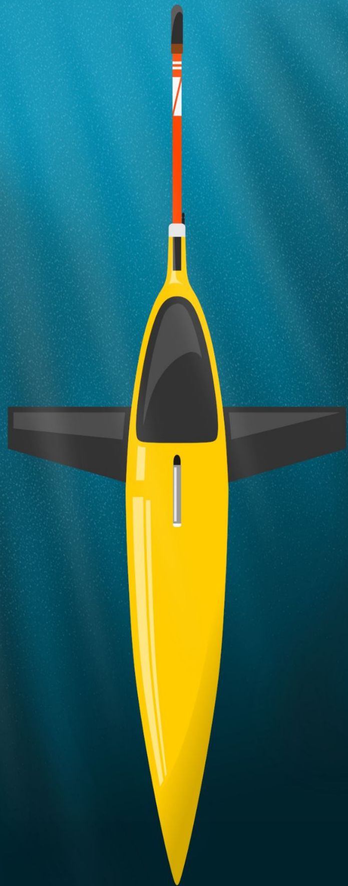


SEAGLIDER

IOP office hours

Integrative Observational Platforms group (IOP)

iopsg@uw.edu

June 9, 2026

UNIVERSITY of WASHINGTON
Applied Physics Laboratory

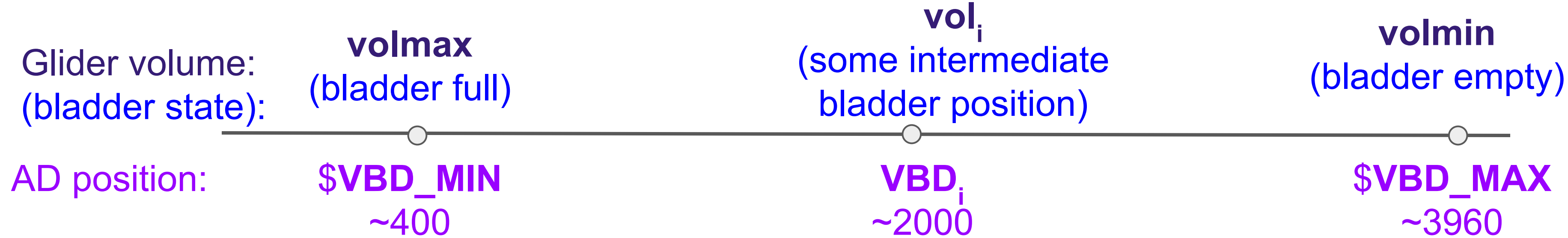
What we are going cover/who asked us questions

1. A brief note on topics we are covering for all office hours
2. Ballasting/estimating volmax from a tank: steps to prep a glider to go out for its sea trial.
3. Best practices for mission metadata/calibration/hardware tracking: What info is critical for tracking for each glider, how do you organize all of that, and how do you make sure all gets updated in a timeline manner by different people in the team?
4. Overview of sensor integration with Seaglider
5. Overview of plotting/adding custom plots

1. Estimating volmax from a tank

IOP's process for preliminary ballasting before the sea-trial

(1) Weigh glider to get scale mass (***M***)
(2) Soak overnight in a freshwater tank that is large enough to fully submerge the entire glider
(3) Compute water density in the tank ($\boldsymbol{\rho}_{\mathbf{tank}}$) based on temperature (and salinity if a saltwater tank)
(4) With an accurate scale, measure the glider's weight in water ($\boldsymbol{W_i}$) at an intermediate VBD position ($\boldsymbol{VBD_i}$) – e.g., 2000 AD counts
(5) Compute **volmax** (see next slide for math)
(6) Repeat at a few VBD positions (we do 2000,2250,2500, 2750,3000) and average them
(7) Compute the target mass for a target thrust and density as usual - see **Tools → Ballast worksheet** in vis
(8) (Note! This gives only a rough estimate of volmax - perhaps ±100 cc)



Difference in volume between bladder full and bladder empty

$$= \mathbf{volmax} - \mathbf{volmin}$$

$$= (\mathbf{VBD_MIN} - \mathbf{VBD_MAX})*\mathbf{VBD_CNV} \qquad (\mathrm{VBD_CNV} = -0.2453\ \mathrm{cc/AD})$$

Intermediate bladder volume

$$= \mathbf{volmax} - \mathbf{vol_i}$$

$$= (\mathbf{VBD_MIN} - \mathbf{VBD_i})*\mathbf{VBD_CNV}$$

We can measure the intermediate volume by measuring the Seaglider mass (**M**) and its weight in water ($\mathbf{W_i}$) of density $\boldsymbol{\rho}_{\mathbf{tank}}$:

$$\mathbf{vol_i} = (\mathbf{M} - \mathbf{W_i}) / \boldsymbol{\rho}_{\mathbf{tank}}.$$

Rearranging gives:

$$\mathbf{volmax} = (\mathbf{M} - \mathbf{M_i}) / \boldsymbol{\rho}_{\mathbf{tank}} + (\mathbf{VBD_MIN} - \mathbf{VBD_i})*\mathbf{VBD_CNV}$$

2. Best practices for mission metadata/calibration/hardware tracking

We keep a maintenance log (a custom searchable sqlite database) in which we log:

- Any service performed
- Sensors or other parts (e.g., antenna) moved from one glider to another
- Ballast numbers, including calculations and scale weights
- “To do” items
- Scanned documents (checkout sheets, ballast sheets, refurb notes, etc)

We also track the following separately:

- Science sensors - serial number, calibration date, calibration data (scanned calibration sheets), status (e.g., “broken”, “needs cal”)
- Pressure sensors - serial number, span, associated $PRESSURE_SLOPE

3. Overview of sensors in the glider and Basestation

- A brief overview of of sensor integration into the Seaglider, with a focus on data flow from the Seaglider and data handling/processing on the basestation.
- Documentation (a work in progress): https://github.com/iop-apl-uw/basestation3/blob/master/docs/sensor_integration.md

Sensor classes: serdev

- Instruments may be attached to the glider "truck" or main motherboard directly.
- Instruments on the truck are treated as spot sampled: turn on, wait for a warm up, read data, and turn off.
- A serdev **.cnf** file contains information on how to interact with the instrument and on how to extract and store data from the instrument into the Seaglider's **.dat** file (the truck generates a single **.dat** file per dive)
- https://iop.apl.washington.edu/iopsg/serdev.txt for details on authoring a serdev file

Sensor classes - serdev (cont)

- Selected sections from a serdev file (**legato.cnf**):

```
prefix=rbr
format="%d-%d-%d %d:%d:%f %00 %01 %02 %f %f %f %f %03"
column=conduc(10000,0)
column=temp(10000,0)
column=pressure(1000,0)
column=conducTemp(10000,0)
```

- The `prefix` must be unique - preferably across all Seagliders everywhere - but it must be unique for all instruments configured on a single basestation. The `prefix` is combined with the name portion of the `column` to form the name of the data column in the gliders **.dat** file:

```
columns:
rec,elaps_tms,depth,heading,pitch,roll,GC_state,mag.x,mag.y,mag.z,rbr.conduc
,rbr.temp,rbr.pressure,rbr.conducTemp,
```

- The remaining portion of the `column` spec is a scaling factor and an offset value. The glider will subtract the offset, then multiply by the scale to transform the data read from the device. This is important to get correct, since data columns are transmitted as integer values. Furthermore, getting the scaling such that it yields values as close to zero (on the positive side) is desirable, because the glider will compress the data with a first order difference, so keeping the values close to zero means a smaller difference to be encoded.

Sensor types: loggers

- The other class of instruments are generally described as “loggers”. Loggers are also attached to a port on the Seagliders motherboard.
- Unlike serdev instruments, loggers are instruments that are generally turned on at the start of the cast and off at the end.
- Loggers are expected to handle their own data sampling schedule, and store any data internally and/or return data to the glider at the end of the dive.
- Loggers’ software support exists in two broad categories:
    - Custom loggers (where all behavior is built into the glider's firmware)
    - logdev loggers (where the interface is defined by a **.cnf** file)

Sensor types: logdev

- Documentation in https://iop.apl.washington.edu/iopsg/logdev.txt
- Selected **.cnf** contents:

```
prefix=mr
datatype="u"
xmodem="%F%r%[$]sx -k /home/debian/links/%f%r%[now.]"
```

- `prefix` is always exactly two letters and must be unique from any other loggers - preferably across all Seagliders everywhere - but it must be unique for all instruments configured on a single basestation.
- While a logdev device can create generate files of any name, the files that are to be transmitted from the glider to the basestation and processed by the basestation must conform to the Seaglider file naming convention, part of which is made up of the `prefix`.
- The `datatype` is one of the known file packing or compression types (see below) – typically `z` or `u` for gzipped or uncompressed data. This specification is used to composite the `%f` expansion to define a filename in the glider's namespace.

Sensor types: scicon

- Documentation here: https://iop.apl.washington.edu/iopsg/Scicon_manual_June2024.pdf
- The scicon is a particular custom logger, that itself has devices (serial or frequency counted) attached to it.
- The scicon runs its own sampling schedule, which is largely independent of the glider's science sampling and is much more flexible.
- As with the glider, instruments are spot sampled.
- Scicon produces one data file per instrument, per dive, per cast (dive, loiter, climb)
- There are three config files that control the scicon operation:
  - **scicon.ins** - The instruments' interface definition file.
  - **scicon.att** - How the instruments are attached to the scicon hardware.
  - **scicon.sch** - How the instruments are sampled.

Sensor types: scicon (cont)

- Selected lines from scicon.ins

```
tridentebb700bb470chla470 = {
        format = %d, %00, %01, %02
        column = bb700(100000,0)
        column = bb470(100000,0)
        column = chla470(1000,0)
       }
```

- The `column` name in the final **.eng** file produced by the basestation will be formed by the sensor name - `tridentebb700bb470chla470`- and column name - `bb700`
- Just as in the serdev devices, data will be offset and scaled before a first order diff is applied prior to data transmission.
- Unlike the serdev, the scale and offset are included in the transmitted **.dat** file.

Basestation sensor extensions

- While not universally true, almost all code related to the data flow from **.dat** files to the **.nc** files is housed on a “sensor extension”. These are individual python modules that are called by the basestation code during processing to perform specific transformations or configurations and metadata for the instruments the sensor support.
- In the case of serdev instruments, there is a built-in set of machinery that can use a **serdev.cnf** file to drive some of this processing.

Names and metadata

- The process of writing variables out to netCDF files is driven by metadata for each variable - such as the type of the variable, the units, standard name and what dimension and time variable the sensor is associated with.
- Sensor extensions provide this metadata, typically as part of the sensor initialization.
- Serdev devices can supply the metadata as part of the **.cnf** file in specially formatted comments.
- Scicon-based instruments and loggers require a python-based extension to provide this metadata.

Data flow and basestation processing

From a sensor data standpoint, basestation processing can be boiled down to:

1. Transforming incoming **.dat** files (to **.asc**) to **.eng** files.
2. Reading the **.eng** files:
   a. Reading in the contents from all **.eng** files on disk from all sensors (truck, scicon, loggers) from a single dive
   b. “Rehydrating” the files from an existing netcdf file
3. Running automatic science calculations and quality control processes
4. Writing the results into a **.nc** (netCDF) file
5. Generating plots from the **.nc** files

The distinct steps may seem cumbersome, but there is a reason. The basestation code will only go back “as far as needed” when processing, starting with steps 1, 2a or 2b.

- 1 is the start only when there are new/updated files transferred from the glider
- 2a is used only when a new/update .eng (or .log) file associated with the dive is updated.
- 2b is used otherwise - **Reprocess.py** (and FlightModel) for example, allowing reprocessing of data with only the netCDF files.

Basestation processing: **.dat** to **.asc**

- Only for truck and scicon
- For the truck and the scicon, each **.dat** file is a row/column format with a single time vector. The conversion to an **.asc** file is the mechanical undoing of the first order difference compression that the Seaglider performed.
- For the truck, the **.asc** file is actually written out to support later processing, while for the Scicon instruments **.asc** format is maintained internally.

Basestation processing: **.asc** to **.eng** for serdev

- **.asc** to **.eng** involves undoing the scale and offset transformations and adding some additional metadata. The resulting file contains the data as captured on the glider or scicon.
- Since the scale and offset transformations are not encoded in the in the truck **.dat** file, the scale/offset transformation must be informed by the sensor extension.
- A **serdev.cnf** file can be used to provide the information.
- For a python sensor extension, the transformation is encoded in the **asc2eng** function. If no extension is able to handle the scale/offset conversion, the data column is propagated to the **.eng** file without conversion.

Variable name remapping for serdev

- Sensor extensions can provide for variable renaming as part of the asc to eng transformation by providing the **remap_engfile_columns_netcdf** function.
- This only works for sensors on the truck. The purpose of this remapping is to be able to process data sets that were generated with sensor names that don't match the current naming conventions.
- As list of remappings supported are in https://github.com/iop-apl-uw/basestation3/blob/master/sg000/sg_calib_constants.m
  - Of the form `remap_<general_instrument_class>_eng_cols`
- Remapping is documented here for completeness - new sensor extensions should not need to make use of the facility.

Basestation processing: **.dat** to **.eng** for loggers

- The logger sensor extension's **process_data_files** function is responsible for the transformation from **.dat** to **.eng**, if any.
- Many extensions elect to simply rename the file to have a **.eng** extension, and put logic for reading data format for the next step.
- Logger **.eng** files need not be human readable.

Basestation processing: reading **.eng** files

- For the next step of the process, the basestation will collect all **.eng** files for a single dive that have the same `prefix` (`sg`, and `sc` examples for seaglider and scicon) and call the associated eng reader (the name of the reader is registered at sensor extension initialization time).
- The reader processes the **.eng** file(s) and returns any data to be used downstream along with the name the for each data column (this is the name that will appear in the netCDF file).

Basestation processing: reading **.eng** files - serdev and scicon

- For serdev
  - Names are transformed from `sensorname.columnname` to `eng_sensorname_columnname`
- For scicon
  - The reader will merge all the casts together to form a coherent timeseries.
  - Variables are renamed from `sensor.columnname` to `sensor_columnname`
- For loggers
  - There are choices here, depending desired output.
  - Some elect not to house data in the netCDF file, so the `**eng_file_reader**` returns nothing.
  - Others can choose to propagate the data to the netCDF file, so the reader needs to do any transformation of the logger data to match the name and shape as defined in the metadata.
  - An example: for a logger that returned data in a **.mat** files(s), the extension might read the **.mat** files and build up single time series for each variable in the **.mat** file
- For all three classes of sensor, the read data is returned to the main processing basestation code

Basestation processing: actual data processing

- An extension may include code that performs science or quality assurance code that acts on sensor data - this is done through the **sensor_data_processing** function.
- The extension has access to all variables that are destined for the netCDF file.
- This function in the extension is called after the CTD data has been processed (hydro model and its output are available as well).
- The contract is new data columns may be introduced, and existing columns left alone/not modified.

Basestation processing: writing out netCDF files

- There is no sensor extension interaction during data writing - all the writing is driven off the data columns in memory (as read from the **.eng** file or added during processing) and its associated metadata.

4. Overview of plotting in basestation3

- A brief overview on the generation of plots in basestation3, with a focus generation plots that are not available in the default basestation3 package
- Documentation (a work in progress): https://github.com/iop-apl-uw/basestation3/blob/master/docs/plotting.md
- Plots and vis discovery
- Plot file format
- Ways to generate plots

Plots and vis discovery

- All plots reside in the `plots` sub-directory under the mission directory
- Vis will scan this directory for new/updated plots
  - When signaled from the normal processing in **Base.py** (at the end of dive process and at the end of mission processing)
  - When the browser is refreshed
- Any files meeting the naming convention will be viewed as a plot
- For dives - `dvdddd_<plot_name>.<extension>`
  - `dddd` - dive number
  - `plot_name` - name of the plot
  - `extension` - image type (next slide)
  - Displayed only in the specific dive context
- For mission plot - `eng_<plot_name>.<extension>`
  - `plot_name` - name of the plot
  - `extension` - image type (next slide)
  - Displayed for any dive

Plot file formats

A few general formats

- Static formats
    - **.webp**, **.png**, **.jpg** all work
    - Output size will be scaled automatically for display
    - A good default size is 1058 pixels wide and 894 pixels high
- Vector format
    - **.svg** works
- HTML content
    - Basestation3 used Plotly (https://plotly.com) to generate **.div** files that are loaded by vis
- If you have two versions of the same file (dv0001_ctd.webp and a scalable format (vector or html) - dv0001_ctd.div as examples), the static file will be used in the thumbnail ribbon and the scalable will be used in the main page.

Ways to generate plots

If there is no current plot that meets your needs, generate one of your own:

- Decoupled from basestation processing
- Using a hook script
- Writing a basestation extension
- Writing a basestation plotting extension
- Using one of the configurable plotting features

Ways to generate plots (cont.)

Decoupled from basestation processing

- If a new plot file shows up in the plots directory, vis will find it and display it (eventually)
- Possible approaches
    - cronjob triggered process
    - Generate a plot off machine and copy into the directory
    - N.B. - make sure the file permissions are set so the vis process can read the plot

Ways to generate plots (cont.)

Use a hook script file

- Hook scripts are general executable files that you supply
- They are called at various points throughout the conversion process
  - https://github.com/iop-apl-uw/basestation3/tree/master#additional-hook-scripts for list of hook scripts and script arguments
- **.post_mission** is a good candidate for where to run your own plotting code
- Hook scripts are run run in the user context of the conversion (the seaglider account or runner account, depending on how your basestation is setup
  - Code run needs to be accessible to the conversion user account
  - Output files (into the plot directory) have the correct user and group ownership

Ways to generate plots (cont.)

Using a basestation extension

- Basestation extensions are pieces of python code that can be executed at various places during the conversion process
- Theses can be used for a wide variety of of purposes from data processing, new file type generation to - in this case - generate plots
- Extensions can use the current basestation software infrastructure to do work (e.g. the machinery to generate plots using plotly)
- The need to be installed in the mission directory, the `group_etc` directory or the `/usr/local/basestation3/etc` directory
- There is a sample extension - **SimplePlotExtension.py** - included in the basestation3 source that can be used as a starting point for your own extension.

Ways to generate plots (cont.)

Using a basestation plotting extension

- Basestation plotting extensions (not to be confused with general basestation extensions) are the most closely coupled with the built-in basestation plotting routines
- All plotting extensions reside in `/usr/local/basestation3/plotting` or in `/usr/local/basestation3/plotting/local`. Files in the latter location may be symlinks to other locations in the file system.
- Dedicated plotting extensions vs. generic extensions
  - Are available without any **.extension** configuration
  - Gets access to more of the plotting output configuration setup and internal notification
  - Generally good for plots that are likely to be used for many missions/permanent addition
  - No difference in the number/type of plots you can generate
- Sample extension forthcoming

Ways to generate plots (cont.)

Using one of the configurable plotting features

- For single dives plotting, https://github.com/iop-apl-uw/basestation3/blob/master/sg000/divescience.yml allows for single or multiple variables to plotted as down and up profiles against gliderdepth.
- For a series of dives, https://github.com/iop-apl-uw/basestation3/blob/master/sg000/sections.yml allows for the plotting of single variables as a heatmap as time vs. depth for the entire mission