Code documentation

1. **app.py, Dash and Flask initialization**

The app runs with UI through Dash, and Dash by default accesses a localhost server for your app to run on through Flask. Flask can be used outside of the automatic integration with Dash to port your app to an online server, which is what I did with hosting the app on a DigitalOcean domain. I paired Flask with OAuth and used Azure's backend IDs for @astroa.org to force any user who wants to access the site to go through ASTRO's Azure authentication system before they're allowed in. The app itself doesn't have a different web address for each page within the app, since it's all controlled by a group of buttons which technically loads each layout onto the same page as the initial one, but I use web address redirects through server routing in Flask for the authenticator and for downloading files off the database. To account for cases where the org wants to allow other users access to the database I coded a guest link generator, and locked the guest link generator to only "admin" sessions (@astroa.org users) so that "guest" sessions can't distribute the link.

2. **Dash layout-callback flow*: layouts__.py, callbacks__.py***

2a. Dash layout-callback flow overview

- The core of Dash are the layouts and callbacks. Layouts are basically html which you code using Dash html components, and the callbacks respond to elements of the layout when you interact with the layout. Each layout member has an id, and a callback with Input or State arguments tracking that id can respond to different properties of the layout member (i.e. a button Input argument would track 'n_clicks' of the button associated with that button id, or a dropdown State argument would track the 'value' of the dropdown id). Outputs of callback arguments map to the returns of the functions which the callbacks decorate.

2b. Implementation of **2a**

- Essentially the way my code structure breaks down is I have a ton of different files which house functions which the callbacks call when the user interacts with different parts of the layout and triggers certain callbacks. I.e. *backend_top.py* and *backend_deep.py* house functions for uploading, processing and retrieving files which the user would upload to the database, and these functions are for the most part triggered in *callbacks_uploaddata.py*, which reacts to the relevant layouts that are written in *layouts_uploaddata.py*. This kind of flow is consistent for every part of

the database, and I kept it decently well organized so that general categories of functionality are grouped in the same files.

- *utils/* houses a bunch of files like *tools.py, plot_tools.py, data_compiler.py, helpers.py*, etc. and these are loosely organized utility files whose contents are referenced by different functions for the backend processes of the database. Nothing actually runs out of these files but the functions within them are used many times in the callbacks and other higher level files. The way I thought was intuitive to organize my file workflow was layouts and callbacks at the top layer, then within callbacks there are calls to bigger parent functions housed in files living in *utils/* (mostly) which do the heavy lifting of the code. Then files like *tools.py* generally have smaller, shorter functions which are called all over the place and these would be considered the lowest level.

3. Database mechanics

3a. Processing input

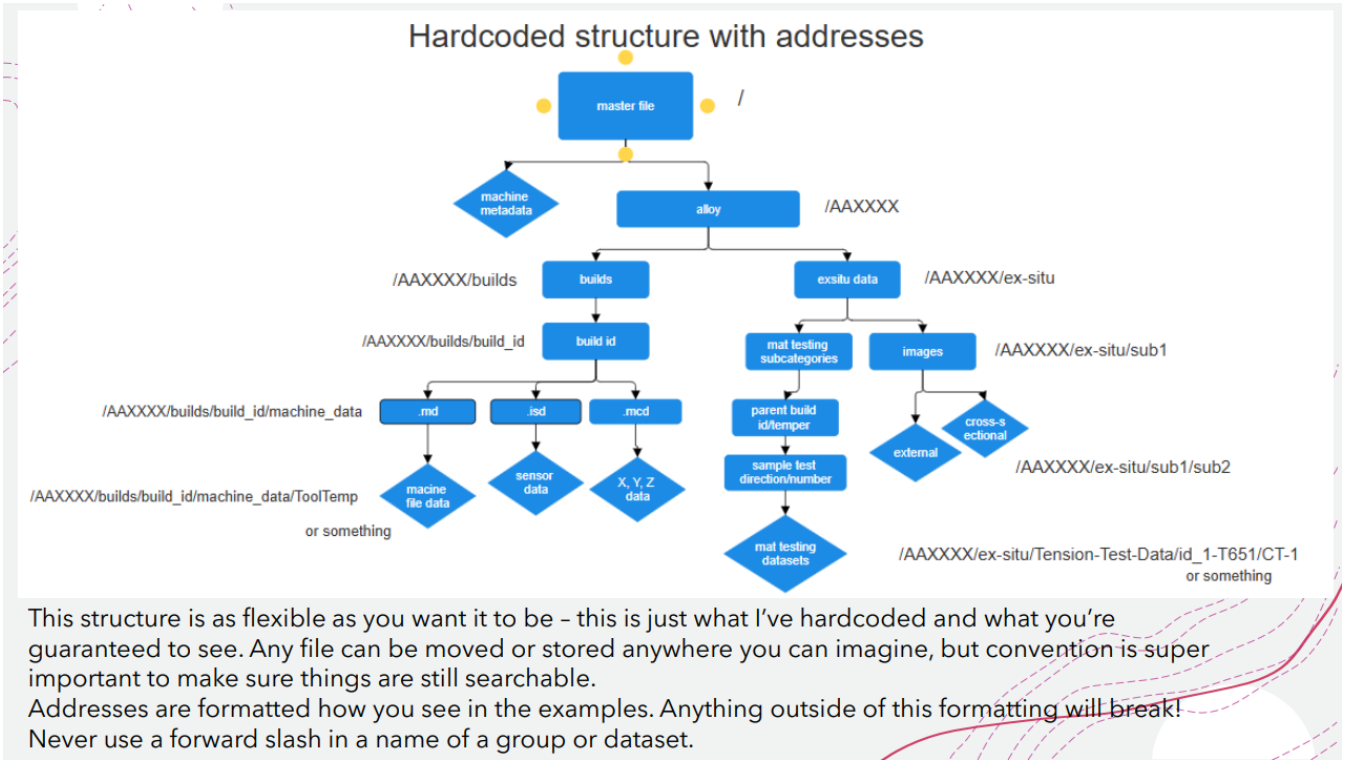
Most files that need to go into the database are .csv files which are taken off the machine, the in-situ monitoring, etc. and I use the *pandas* library in python to handle these. Csvs are processed as pandas DataFrame objects and injected into the database as one group with a given name that houses *h5py* Dataset objects which are created from the columns of the DataFrame. This is convenient for data analysis because you can grab these datasets from the database and vectorize them using *numpy* to be used to compare statistics, generate plots, etc.

- Machine, in-situ and motion capture CSV files are automatically ran through processing functions which use *pandas* and *numpy* to clean data and headers, standardize time, adjust/standardize sample rate, and add some useful information like an on/off column for example. This makes the files more unified per build so analysis becomes simpler.

Generally any file is handled according to its filetype (sounds obvious but that's literally how the code flows). Dash encodes every file upload by default in base64, so my code decodes that from base64 and reads the filetype from the header of the Dash file upload 'contents' property. For text/csv or ms-excel (different save type of .csv) files, they are processed according to the previous bulletpoint's procedure, image files are stored as byte arrays, and text files are stored as 'utf-8' decoded strings.

3b. Storage

HDF5 addresses are passed around in the code as strings and look like master/group/subgroup/dataset or something. Every query into the database must go through the master file, and so I use “with .. as.. :” syntax to open the master file and create a node by referencing a certain path within the master file which then either represents a group (*h5py.Group*) or dataset (*h5py.Dataset*) object. Accessing the database like this allows me to manipulate database objects in my code like other variables and perform operations, extract information, etc. as I need.



Taken from the documentation slides. Groups are boxes, datasets are diamonds.

This image also gives insight into storage considerations. I hardcoded the organization of the build files so that the user would just fill in a template of information about each build and it would auto generate under the appropriate address, and same thing for ex-situ/post-processing data which is parallel to build data but connected through attributing. With that being said, the user still has the ability to move files to arbitrary locations as well as upload arbitrary data to any address it wants, but having this much structure auto-generated seemed nice and intuitive to me from a databasing perspective to standardize structure and make it easy to parse.

3c. Attributes

The *h5rdmtoolbox* library (https://h5rdmtoolbox.readthedocs.io/en/latest/) provides streamlined attribute functionality which I took advantage of to create an attribute system within the database. My attribute system has two primary elements: statistical attributes and general ones, both used for different things.

Metric	Min	Max	Avg
Actuator Force (lbs)	6	80	54
Feed Velocity (in/min)	0	13	8
Thermocouple 1 (deg C)	0	0	0
Thermocouple 2 (deg C)	28	197	149
Thermocouple 3 (deg C)	0	0	0
Thermocouple 4 (deg C)	25	198	149
PathVel	0	95	12
Spindle Power (W)	-70	9916	5330
SpinSP	0	360	351
Spindle Torque (ft-lb)	-9	409	228
Spindle Speed (RPM)	0	284	162
Tool Temperature (deg C)	14	373	352
XTrq	-17	17	0
YTrq	-19	17	0
ZTrq	-31	9	-15

Above is an example of statistical attributes, displayed in a formatted way. These attributes are generated by reading the data from the machine file (which is first uploaded to initialize the build file) and outputting statistics for important columns which characterize the build. These attributes are assigned at the build level (/alloy/builds/build_id), whereas similar statistical attributes for sensor data are assigned to the sensor data parent group itself (/alloy/builds/build_id/in_situ_data or similar).

Other Attributes:

- alloy: AA7075
- build_id: hub_concentric_TBuild_6
- csv_path: hub_concentric_TBuild_6_md.csv
- failed: true
- sample_rate: 20

Other attributes can be any information the user would want to add to a build file or any file in order for that file to be pulled through some kind of search query later on. In the picture, these are all examples of attributes which my code auto-assigns for different backend purposes. For example, the build_id attribute is assigned to all data related to one build anywhere in the database, so using the search tab the user could search for all builds with

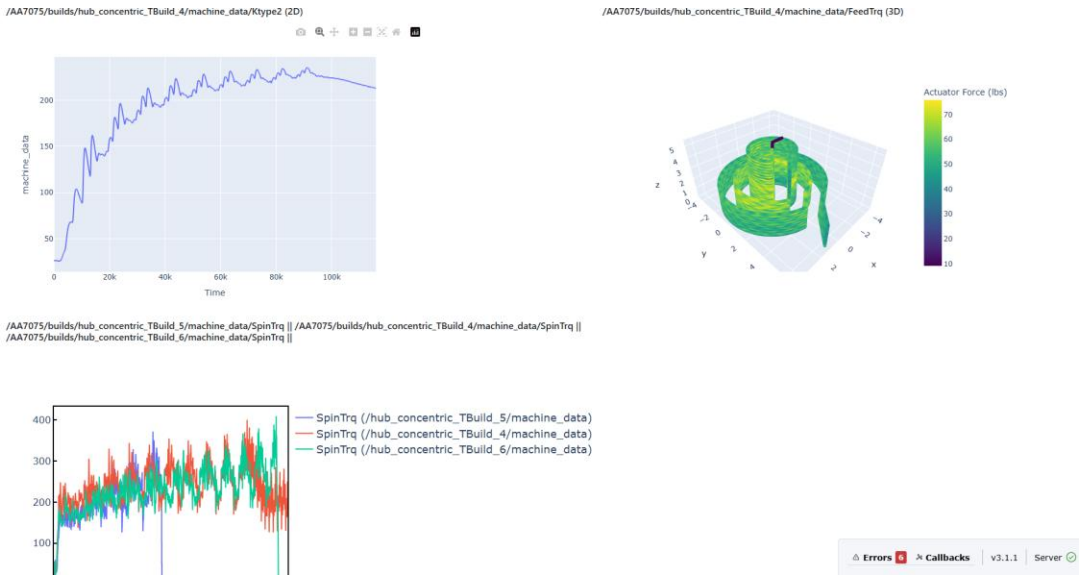
“build_id”: “interesting_id” and they’d be able to pull all information about that build stored in the database. The sample_rate attribute is automatically assigned by the data cleaner and used to standardize sample rates of files across a build in the database in order to allow for seamless analysis through combined and exported files, 3D and 2D graphs, etc.

There’s a dedicated tab in the database for searching by attributes. The functions use built-in *h5rdmtoolbox* methods to find files with matches for attributes by exact value or lower/upper bound for numerical attributes, and if used intelligently can give a way to access any interesting file based on the attributes it has. This ability to quickly parse the whole database is one of the big benefits of the HDF5 architecture, especially for ASTRO’s use case. What’s really cool about the search tab, aside from the actual search function, is that you can immediately have access to details about any dataset within a group if the result of a search is a group, view all the attributes associated with it, and if it’s a dataset you can view the raw dataset, a 2D plot if applicable, and attributes associated as well. All of this is accomplished in the code by taking an address of the object returned by h5rdmtoolbox search function, navigating to that data, creating the node by accessing it from the master file and then gathering information using hdf5 built in methods.

4. Analysis tools

4a. Graphing interface

The graphing interface can take any dataset and plot it against time or position as a 2D or 3D plot respectively.



Screenshot taken from "Analyze data" tab where I loaded in different plots as examples.

The graphing interface was an absolute trip to code, especially because it was one of the first things I made. It took an ungodly amount of trial and error to get through and it's not the smoothest, but it definitely works. It has three cases: plot all, plot individual, and plot a stacked 2D graph. In each case the database accesses datasets at the address queried and plots them against either time or position depending on the dimension of the graph. For 2D plots, the database finds the time column in the sister datasets stored underneath the same parent (i.e. the time column for in_situ_data when asked to plot total force vs. time) or the position data from machine_data in the same build file to generate a 3D plot. There's a ton of code on the backend going into queuing, storing, formatting and labeling the plots within the relevant files but it's not worth diving into. The plots are interactive *plotly.express* objects which allow for rescaling and zooming in both dimensions.

4b. Analyze data tab

The analyze data tab has two primary parts, build comparisons by attribute and showing a stored list of graphs to look at them side by side if you can't do so in the graphing interface (i.e. Feed Torque plot from two different builds). Data is loaded into this tab either from the graph or search interface. The build comparison is the interesting part of that tab in my opinion, and it has the ability to either load in a build as a benchmark from the search tab and compare it to a list of any arbitrary number of other builds, or to create a custom benchmark with selected metrics and compare only those metrics against any other build loaded in from search tab.

[in_situ_data] vs [in_situ_data]

Metric	% Difference
Force_1_Force_avg	6.3
Force_1_Force_max	235.5
Force_1_Force_min	-34.3
Force_2_Force_avg	23.4
Force_2_Force_max	196.3
TC_5_Temperature_max	-23.4
TC_5_Temperature_min	7.7
TC_6_Temperature_avg	-5.2
TC_6_Temperature_max	-4.8
TC_6_Temperature_min	-5.4
TC_7_Temperature_avg	545.1
TC_7_Temperature_max	611.6
TC_7_Temperature_min	11.8
TC_8_Temperature_avg	-4.1
TC_8_Temperature_max	5.3
TC_8_Temperature_min	-5.4
Total_Force_avg	74.6
Total_Force_max	106.6
Total_Force_min	77
build_id	Different
Normalized difference:	**1123.6**

[in_situ_data] vs [in_situ_data]

Metric	% Difference
Force_1_Force_avg	-28.9
Force_1_Force_max	-147.5
Force_1_Force_min	-32.6
Force_2_Force_avg	-1.7
Force_2_Force_max	-5.8
TC_5_Temperature_max	-10.9
TC_5_Temperature_min	247.9
TC_6_Temperature_avg	16.3
TC_6_Temperature_max	19.5
TC_6_Temperature_min	11
TC_7_Temperature_avg	19.6
TC_7_Temperature_max	8.4
TC_7_Temperature_min	41.6
TC_8_Temperature_avg	10.7
TC_8_Temperature_max	11
TC_8_Temperature_min	10.2
Total_Force_avg	91.2
Total_Force_max	110.7
Total_Force_min	95.9
build_id	Different
Normalized difference:	**200.2**

[in_situ_data] vs [in_situ_data]

Metric	% Difference
Force_1_Force_avg	-1
Force_1_Force_max	234.7
Force_1_Force_min	-38.3
Force_2_Force_avg	13.2
Force_2_Force_max	196.2
TC_5_Temperature_max	-34.9
TC_5_Temperature_min	10.7
TC_6_Temperature_avg	6.1
TC_6_Temperature_max	2.5
TC_6_Temperature_min	10.1
TC_7_Temperature_avg	14.5
TC_7_Temperature_max	8.7
TC_7_Temperature_min	31.7
TC_8_Temperature_avg	-0.3
TC_8_Temperature_max	4
TC_8_Temperature_min	-2.7
Total_Force_avg	61.6
Total_Force_max	102.2
Total_Force_min	70.8
build_id	Different
Normalized difference:	**1285**

Example of comparison summary with normalized difference at the bottom (labels are not informative on the table but the full addresses are at the top of the page).

To store data and pull it for comparison, global *dcc.Store* (storage bin object from Dash core components) objects are used session-to-session to avoid multi-user interference. They are loaded in the highest level layout so that they can be used by any tab within the database. Something that's in progress as I'm writing this documentation is the ability to search the database for a match based on a generated benchmark table and a tolerance for normalized difference, which might be simple or might be egregiously complicated and I'm not exactly sure yet.

4c. Browse database tab

While not really an analysis tool per say, I figured it was worth mentioning somewhere because this is probably the most useful tab on the database. It has dynamic callbacks to generate dropdowns as the user clicks through each group, so that you can find any file in any location within the database, and then has the ability to copy the address so the user can use that address elsewhere without having to type a long HDF5 format address themselves. Can "show details" of anything the user finds (same as search tab) and also export datasets as .csv files or merge machine, in-situ and motion capture data for a build into one .csv and export.

5. General considerations

I tried to split up my codebase to keep files under 300-400 lines for the most part, so that everything was organized and easy to find. This ended up backfiring a little bit because I started running into circular imports or forgetting where functions are since there's so many files, but the nice thing about Python is that in the imports section at the top of every file you can see where things come from so it doesn't end up being that big of a deal. If you want to parse my codebase for something specific and don't know where to look for it, Chat GPT will take a .zip of *database_root/* and find it for you.