Analysis v3 - A new data analysis concept

General concept

Conceptually, the analysis_v3 framework consists of an analysis pipeline containing several data processing nodes, which can be either functions or classes (see illustration in Fig. 1). Several conceptual processing pipelines can be defined and run on the same input raw data, or on a subset of it. For example, as shown in Fig. 1, if the raw data consists of single shots acquired on several readout channels, we could create Pipeline 1 (green) for each readout channel, or analyze only channel 0 according to Pipeline 2 (blue) and then Pipeline 3 (purple) in order to quantify the difference between ignoring and including the f-level in the state classification (second node).

To better understand the concept of analysis_v3, let us look at a specific example. Listing 1 shows how to use analysis_v3 to analyze a two-qubit randomized benchmarking measurement. The ProcessingPipeline object (line 11) is used to create the pipeline as a list of dictionaries, where each dictionary contains the input parameters to a node function/class. ProcessingPipeline needs to know the *meas_obj_value_names_map*, which is a dictionary with the name of the measurement objects as keys, and the corresponding value names as values (the latter are created by the detector function based on the UHF channels used by each measurement object). The measurement object can be a qubit, a TWPA, a device, etc.; it is the object which is being characterized. The data processing nodes are then added for each measurement object via the method add_node of ProcessingPipeline (lines 13 - 21). All the nodes process the data from *data_dict* that is specified by the input parameter *keys_in*. Hence, in practice one does not need to write individual ProcessingPipelines for each readout channel in the data. All the nodes for all the channels can be concatenated into a long pipeline, where one specifies in *keys_in* which channel(s) the node to process. In this example, the final pipeline would be: average qb1 - get stderr qb1 - analyze RB qb1 - average qb4 - get stderr qb4 - analyze RB qb4 - average corr - get stderr corr - analyze RB corr. The following theoretical pipelines are also valid:

- processing_pipe = filter resets ch0 - filter resets ch1 - ... - filter resets ch N - Rabi on filtered ch0 - Rabi on filtered ch1 - ... - Rabi on filtered ch N
- processing_pipe = filter reset ch0 - classify qubit ch0 - classify qutrit ch0 - Rabi on output of classify qubit - Rabi on output of classify qutrit
- processing_pipe = average ch0 - ... - average ch N - get stderr ch0 - ...

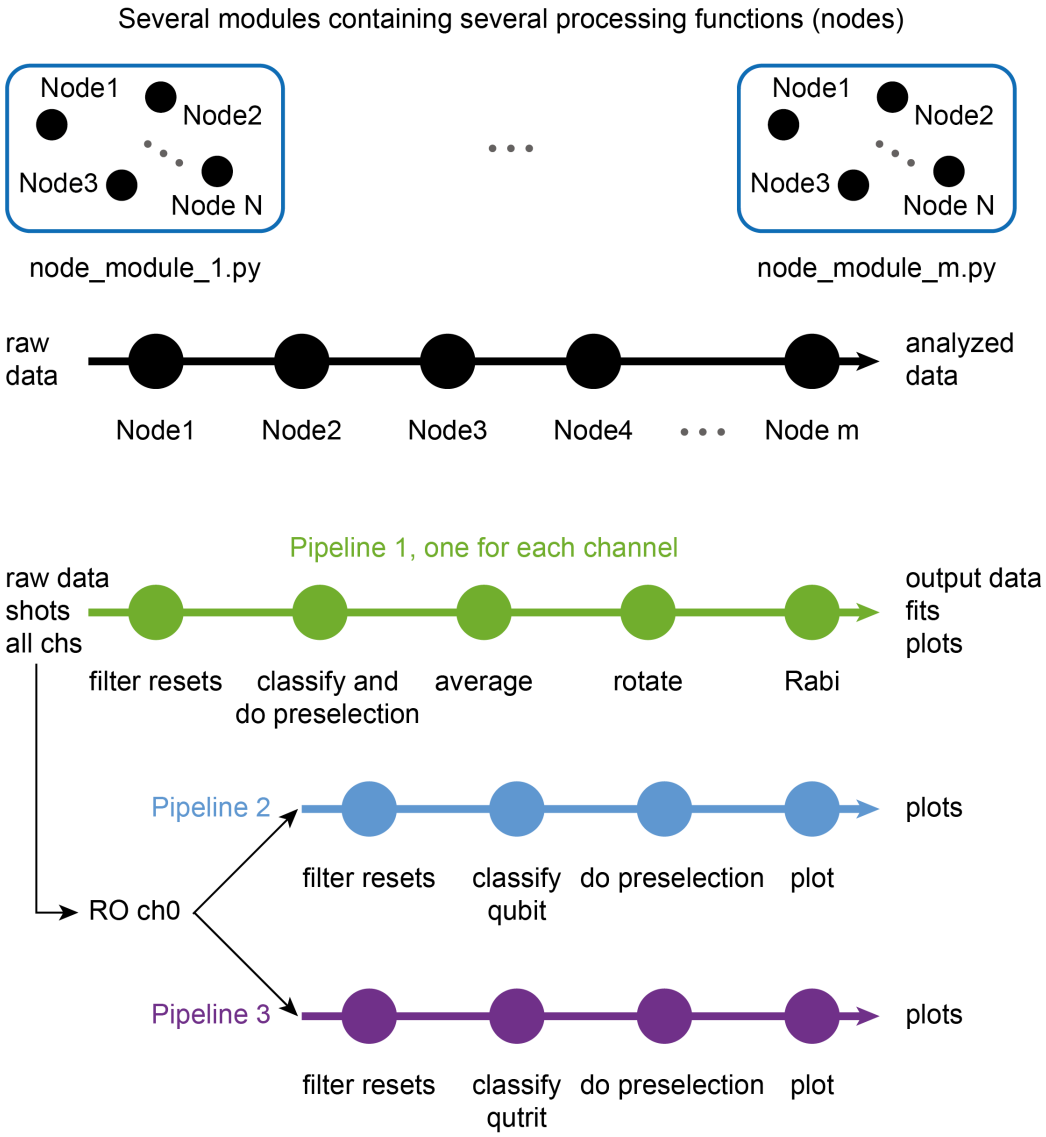

Figure 1: Overview of analysis_v3. The last 3 colored pipelines are examples of how some raw single shot data can be processed in different ways.

get stderr ch N - plot averaged ch N - single qubit RB on averaged ch0 - ... - single qubit RB on averaged ch N-1

```
## Two-qubit Randomized Benchmarking analysis ##

# 3 measurement objects, qb1, qb4 and their ZZ correlator
obj_names = ['qb1', 'qb4', 'corr']
# measurement objects - value names map. The latter are created by the detector function
```

```
6  # based on the UHF channels used by each measurement object
7  meas_obj_value_names_map = {'qb1': ['UHF1_pg w01 UHF1', 'UHF1_pe w01 UHF1', 'UHF1_pf w01 UHF1'],
8                              'qb4': ['UHF1_pg w45 UHF1', 'UHF1_pe w45 UHF1', 'UHF1_pf w45 UHF1'],
9                              'corr': ['correlation UHF1']}
10 # instantiate a ProcessingPipeline object
11 pp = ProcessingPipeline(meas_obj_value_names_map)
12 # add processing nodes for each  measurement object
13 for mobj in meas_obj_value_names_map:
14     # average over the random Clifford sequences
15     pp.add_node('average_data', keys_in='raw', meas_obj_names=[mobj],
16                 shape=(nr_cliffords, nr_seeds), averaging_axis=-1)
17     # get stderr of the results for the different sequences samples
18     pp.add_node('get_std_deviation', keys_in='raw', meas_obj_names=[mobj],
19                 shape=(nr_cliffords, nr_seeds), averaging_axis=-1)
20     # fit to model and plot
21     pp.add_node('SingleQubitRBAnalysis', keys_in='previous average_data',
22                 meas_obj_names=[mobj], std_keys='previous get_std_deviation')
23
24 # pass pipeline in options_dict. Can also be taken from experiment metadata.
25 pla.PipelineDataAnalysis(timestamp='YYYMMDD_hhmmss', options_dict={'processing_pipe': pp})
```

Listing 1: Two-qubit randomized benchmarking analysis with analysis_v3.

The class PipelineDataAnalysis (line 25) defined in the module pipeline_analysis.py extracts raw data and handles the parsing of a single pipeline. The pipeline can be either passed in when this class is initialized via the *options_dict* dictionary with the key *processing_pipe*, or it can be defined in the experiment metadata, which this class extracts from the data file. PipelineDataAnalysis creates the dictionary *data_dict*, where it puts the experiment metadata and the raw data it extracts. The *data_dict* is passed from node to node in the pipeline, and each node saves the processed data into this dictionary with a unique key, specified in the input parameter *keys_out*.

At the moment, **analysis_v3 has support only for 1D data**, and one needs to define a different conceptual pipeline for each measurement object, i.e. there are currently no nodes that handle multiple measurement objects. At this point, it is not clear how to deal with 2D data, or multi-object measurements where the analysis needs to take into account joint information.

If a measurement object is used in the pipeline, the analysis will most likely require that cal_points, sweep_points, meas_obj_sweep_points_map, and meas_obj_value_names_map exist in the metadata. If they are missing in the metadata from the HDF5 files, you can still pass in a dictionary with the missing key-value pairs in the options_dict with the key exp_metadata. This dictionary will be appended to the metadata dictionary extracted from the data file. This requirement already raises some issues in some cases because the framework assumes the variables just mentioned are necessarily defined for the measurement object. Already we ran into issues when using the classifier detector with correlated == True. In this case the detector

creates an additional data column for the correlated data with the value name 'correlation.' This data will be treated as corresponding to an additional measurements object (in addition to the qubits) because currently the node SingleQubitRBAnalysis creates a plot for each measurement object. However, the measurement object for the correlated data does not exist in any of the four variables discussed above.

Guidelines for developers

In order to avoid the limitations of the previous two analysis frameworks, analysis_v3 should be developed by insisting on the following aspects:

Writing nodes **The node functions should be very narrow in scope**, i.e. they should do one task only. For example: data filtering node, do preselection node, average node, rotate IQ node, rotate 1d array node, RabiAnalysis node, etc. Only make the node a class if it's absolutely necessary. So far, the only nodes that are classes are very specialized analyses like Rabi, single qubit RB, etc., which need to extract specific qubit parameters (old π-pulse amplitudes for example), fit, and plot based on the fit results. Maybe we can even come up with a way to not use classes at all!

All nodes must have data_dict as the first input parameter. In addition, all nodes should also have the input parameters keys_in and keys_out. The former specifies the key(s) in the data_dict corresponding to the data to be processed, while the latter specifies the key(s) that the node will create in the data_dict and where the processed data will be placed. Please have a good reason for not using these input parameters in a node.

All the nodes must work with readout channels (specified in keys_in), not with measurement objects (like qubits). Please make the node dependent on measurement object only if strictly necessary (for example if cal points or sweep points are used).

Adding nodes We should try to split up the node modules by concept as much as possible; for example: a module for plotting nodes, another for fitting nodes, etc. Feel free to make new modules, or split up existing ones if you think it makes sense. New node modules need to be imported in pipeline_analysis.py and added to PipelineDataAnalysis.process_data(), which is where the pipeline is parsed.

Helper functions **Please try to write and use helper functions wherever possible!** The advantage of these functions is that a particular functionality is implemented only once. Helper functions should be very narrow in scope. Two important helper functions that should be used in all nodes are get_param and get_data_to_process. The former provides a standard way to get a parameter. This function first tries to get it from they keyword arguments of the node, then from data_dict, and lastly from experimental metadata, and can be told to raise an error if parameter is not found in any of these places. The latter helper function returns a dictionary of the form {key_in: 1d_array_to_process}, which is extracted from the data_dict. The reason this helper function exists is to allow for keys_in to be None, in which case this function takes all the raw data channels, and to allow for keys_in to be a path inside the data_dict, with keys separated by '.' (ex: 'measured_data.raw w0'). Another important helper function is get_cp_sp_spmap_measobjn, which extracts the cal points, the sweep points, the meas_obj_sweep_points_map, and the measurement object (we could consider including the meas_obj_value_names_map here as well, but so far I hardly ever used it). Finally, few useful helper functions for handling plot preparations are get_cal_sweep_points (extends sweep points by number of cal points), get_cal_data (returns the data points that correspond to the cal points), and get_msmt_data (returns the data array without data points corresponding to cal points). All these helper functions are currently in the module helper_functions.py. Feel free to add more helper functions, and/or to make new helper function modules to keep things conceptually separated.