Section 1: Using LiQCS

LiQCS is intended to be an all-in-one lidar quality control suite. While this means it can be run as a standalone Python program, it also makes LiQCS a centralized collection of smaller tools that individually operate on lidar files.

Say you were working on some Python program that requires information stored in a lidar file's header. In that case, you can treat LiQCS as a module, and include an `import liqcs` statement to make use of its `parse_header` function. Further along in the program, you decide that you also want to generate a tile index for that lidar file. Just call the `generate_tile_index` function already available from your previous import statement, saving yourself the hassle of digging around a drive for another piece of source code.

In most cases though, LiQCS will be run from the command line using the Python interpreter. Passing the `-h` flag as an argument will yield a full breakdown of arguments, tests, and usage examples, but the general structure of a LiQCS call is as follows:

```
>> python liqcs.py -i <indir> -o <outdir> -t <tests>
```

The only technically mandatory argument is the input directory following the `-i` flag; output will by default be funneled to a new folder in the input directory, and if no `-t` flag is specified, every test will be run.

Something to note is that there is a considerable runtime bottleneck associated with fully reading large lidar files, even more so for .laz files. Most of the current tools can work around this by making use of `parse_header`, but some of the tools, namely lasinfo and the gridding functions, cannot. What this means in a practical sense is that if you don't need to do a certain type of gridding or don't require lasinfo.txt files, you probably shouldn't run those tests. It'll save a lot of runtime.

Section 2: LiQCS Public Functions

These are the functions you may want to access if importing LiQCS as a module.

`generate_grids``(type, infile_path, grid_path, epsg_code)`

> Generate a density and/or intensity grid for the lidar file found at `infile_path`, and save them in `output_path`. The `type` parameter can be ‘dens’, ‘intens’, or ‘both’, to indicate which grid type is to be generated.

`generate_tile_index``(infile_list, epsg)`

> Generate and return a GeoPackage containing a set of tiles, where each tile is a 2d bounding box around one of the input lidar point clouds in `infile_list`

`make_density_histograms``(lasinfo_lr_glob, lasinfo_g_glob, outdir)`

> Plot two point density histograms using globs of last return and ground point filtered lasinfo.txt files. Saves the resulting histogram.png in `outdir`.

`lidar_summary``(list_of_files)`

> Given a list of lasinfo.txt files, summarizes the contents into a dataframe and then returns that object, which can then be turned into a csv file with `df.to_csv` as is done in `main()`.

`void_grid``(file, outdir, shapefile=None)`

> Detects non-water point voids in the lidar file defined by the path `file`, and saves the relevant “void grid” in `outdir`. If one or more bodies of water are known to be in the region covered by this lidar file, a shapefile defining the geometry of the water can be optionally passed in so that those areas are not treated as voids.

`parse_header``(filename, verbose=False)`

> Given a .las or .laz from `filename`, parse the file’s header into a readable struct format. This function allows access to lidar file header information without having to actually unpack the whole file, which is a big deal for .laz files.

Section 3: Guidelines for New Code:

1. Ideal code structure for new tools:
    a. The code should generally try to adhere to the Python Style Guide as outlined in PEP 8 to keep everything consistent and readable.
    b. The tool should be contained within a single callable function, which may or may not call other functions internally.
    c. The tool should receive as parameters a file or file path and any other information needed, and return only necessary results from the test.
    d. The tool should not do any input or output on its own; that's the job of LiQCS. However, Including an optional `verbose=True` keyword argument that causes the tool to print useful intermediary information can be a good idea for testing and debugging purposes.

2. Steps to add new code to LiQCS:
    a. Obviously, add the new function[s] to the liqcs.py source code.
    b. Pick a new character to specify this test after the `-t` flag when running the suite from the command line. You can see a list of which characters are already in use in the argparse description near the top of the file.
    c. Update the argparse description, then add the new argument to the argparse object, writing a help string and default value if needed. Use the other arguments as a reference if unfamiliar with argparse.
    d. If the new test has a dependency on the results of another test, that should be included in the logic of the `set_test_flags()` function.
    e. Call the function from `main()` if the relevant flag is set, and output the results. In a perfect world, it should look similar to this:
    ```
    if char in test_flags:
        test_result = new_function(input_files)
        outfile.write(test_result)
    ```
    Obviously it won't always be this clean, but this is what we should aim for to keep `main()` as readable as possible.

3. Documentation:
    a. Write some docstrings/comments so it's clear what your functions do.
    b. Update the above API documentation to reflect your new addition.