**2020-06-08**

Create functions to accompany my Python 3D-PAWS plotter with varying degrees of QC/analysis involved:

1. Highest, most complex level of data quality control, intended primarily for post-processing
2. Moderate level of quality control
3. Basic quality control, intended for quick real-time use

Make each of these functions callable within whatever plotter you are using.

Consider different plotters specific to the variable you want to plot.

Then, create an ALL plotter that calls each of the individual plotters.

Have the QC functions as an option for each plotter.

Create a separate “Performance Analysis Plotter” for visualizing/testing sensor reliability, accuracy, etc. This will be intended for internal R&D purposes.

**2020-06-09**

Create “reader” functions specific to each sensor.

Also create “plotter” scripts that call the reader functions and have two options: plot all data from one sensor, or plot all data from all sensors.

Create your QC methods that output figures of statistics and analysis that also call the reader function

SUMMARY:

* Multiple “reader” scripts
* 2 plotter “scripts” (QC and Raw)

Consider reworking how you read in the data given those random screwy records, so that you can catch these records and either a) fix them, or b) skip them. An error catcher will be necessary. [if you read in line-by-line, you could eliminate the need to add the leading zeros back in after astropy.ascii.read() removes them!]

**2020-06-10**

You may need to account for unequal list/array sizes of data for comparison plots. If any given sensor was having issues for whatever reason, there will simply be no records for the troublesome times, meaning that the number of data points and time will be different from one sensor or another. It may be possible to plot two sets of data with two different time arrays/lists on the same plot. Alternatively, you may be able to match the dates/times in each array then use corresponding indices to fill in the missing data as NaNs for whichever sensor was experiencing trouble.

When you tackle the plot\_ALL program, think about how you want the user to specify where the data are. So long as you/they have not mucked with the folder structure, in theory, you could easily have them simple specify the parent/site folder that contains all the subfolders. You may also want to include an example of the folder structure in your code comments just in case a user mucked with the folder structure.

For the plotter you are working on right now, keep writing it specific to the BMP sensor. It will then stand alone as its own function that will eventually get called by a plotter. It will then serve as a nice template for all other sensor functions to read in their data.

Bmp280\_20181014.dat from the CSA site has records for the same timestamp but different values for variables (see times 07:18 and 07:21 UTC) – this will need to be another thing to check for

Since it appears you now know what erroneous data values look like (i.e. 27.08 80.74 775.61 990.36 29.25 2198.24) at least for the BMP280 sensors, you can use this to flag records or remove records.

Consider, in addition to your readers and plotters, create data file editors as part of your QC methods. Read in a file, run in through your methods and spit out an edited file with flagged and/or modified records named something like bmp280\_YYYYMMDD\_QC.dat

For your future error checker, look for lines read in that have those weird characters (usually at the end of a file the did not get through the entire day [power outages???] and omit them. They break things…

**2020-06-15**

Consider adding the ability to handle lines in a file that were partially overwritten. This could get complicated as there is no guarantee where the break occurs in the line. You would need exceptions to cover all the bases…

**2020-06-16**

Start thinking about a list of simple statistics you want to run on each dataset/variable. These ought to be calculated *after* you clean up the data. Or maybe not! Maybe this could be a quick and dirty way of pointing out anomalous/erroneous data?

Make a plot showing the local minimum/maximum for the BMP280 sensor’s temperature readings (or more simply the minimum and maximum for each data file), to hopefully illuminate the “topping out” problem (temp appears to max out at 27.08 C)

**2020-06-30**

Check if changing the wildcard option to read in only a subset of files works with all the checkers and data cleansing methods you have setup.

Rerun HTU21D and BMP280 programs with the new additions of wildcard testing and missing-report print statement. Test all programs with *both* a different wildcard ending

Consider plotters and statistics calculations that also compare against like sensor measurements from the same system (e.g. compare temp-C from the HTU21D, MCP9808, and BMP280)

Consider adding a simple calculation for the uptime for each sensor/site; use the “missing\_reports” value as well as the expected reporting interval (i.e. 1-minute) and total duration of the given/read-in dataset (i.e. the minimum and maximum time of said dataset).

Consider outputting a number of simple statistics and calculations in a text file: uptime, downtime, minimum value, maximum value, number of values outside of sensor specification range, number of values outside of climatological range, number of files with erroneous characters and/or partially overwritten data lines, names of problematic files

NOTE: the number of missing reports calculation as it stands right now *could* include timestamps that technically exist but were part of a partially overwritten line of data and was therefore thrown out. This is because those timestamps will be skipped but then filled with a NaN later on.

Two possible solutions…

1. The quick fix would be to subtract the number of times a “problematic line” occurs from the number of missing reports calculated using NaNs
2. Find a way to handle the partially overwritten lines so that more data is kept

Develop a method determine the difference between a line with erroneous characters *only*, and partially overwritten lines to determine the frequency of each.

Develop a method to find the files with duplicated timestamps ~~and/or timestamps out of order and~~ print this to the output statistics file for closer examination. -  *Done!*

~~Something like…~~

~~Find where the difference between each time stamp is zero!~~

~~﻿indices = np.where(df.time.diff() == pd.Timedelta(minutes=0))~~

~~This will return the index (indices) of the~~ *~~second~~* ~~occurrence of the same time (the first time a timestamp is seen again), and subsequent occurrences.~~

NOTE: the code as written operates as such. It calculates the downtime and number of missing reports based on the duration of time it was given (i.e. the minimum/first time in the first file read and the maximum/last time in the last file read). It does *not* not account for the first (last) file starting (ending) at a time other than the beginning (ending) of the given calendar day (i.e. 00:00 UTC and 23:59 UTC). It only calculates the data gaps between the very first time read in and the very last time read in.

Do I want to account for less-than “full” data files? – *No, does not make logically consistent sense; unless you install and start running a system at precisely 00Z on a given day, then including less-than full starting/ending files in this count will imply that there was downtime even for a system that runs perfectly after installation, which is not true*

Develop a method for finding/logging the occurrences of timestamps out of chronological order. This will be tricky because it must be done before all the data gap filling and removal of duplicate timestamps, and because data gaps exist. Think on this one for a minute or two… - *Done*

**2020-07-01**

Plotted a few daily plots where a reset of time (and subsequent duplicate timestamps occurred); reveals unrealistic steps/jumps in the data.

The methods used in my code (df.sort\_index()) do not sort the timestamps in any particular order so the resulting plots from data that is sorted then removed of duplicate timestamps have data randomly jumping around for the length of time the data was “overwritten” (see plots) – which is correct?!?!

Check your mintime and maxtime checkers to see if it can catch non-existent dates (e.g. 2017-09-31); for the CSA dataset, this would technically fall within the acceptable time frame, but your checkers should be made to be able to check the validity of the date itself, not just the format the user inputs – *it will if the sorting, removal of duplicates, and NaN-filling occurs, otherwise, it will NOT catch them*

Consider calculating the slope of the temperature data (just taking the difference in temperature between each consecutive measurement) and using a standard deviation threshold to flag spikes and steps; think about how you would handle NaNs where there are data gaps

Check whether all sensors had duplicate/out-of-order timestamps at the same times – *they don’t…*

Be sure to make some plots over days which have duplicate timestamps for the other sensors to see how they behave.

**2020-07-07**

Develop a way to count the number of occurrences that values for X data field are out of spec / spiking (upward or downward).

Given the number of plotting options you may wind up having within your code, consider splitting things up this way…

Single-plots

**2020-07-08**

Wind speed / direction data: data is recorded to the second but measurements are happening every minute (approximately). Consider sorting data with all information preserved, look for duplicate timestamps with all data preserved, remove duplicate timestamps, check for data gaps by the *minute* (there should be one record for each minute), then fill missing minutes with NaNs.

**2020-07-08**

Handling timestamps in wind speed data: consider sorting the raw data, then look for timestamps that are less than 1 minute apart for your first clue

**2020-07-13**

Plot ALL (all sensors; all variables) : Plot Sensor (plot all sensor’s variables) : Plot single variable from single sensor

Winds will be unique in that a user can plot only wind speed, only wind direction, wind speed *and* direction on the same plot using different y-axes (left and right), or plot winds as barbs by reading in and processing both wind speed and direction data

Consider giving a user the ability to plot like variables from all sensors. For example, temperature from BMP280, HTU21D, and MCP9808. Temperature is probably the only variable this is useful for since all others are unique. This could work with pressure from the same sensor, also.

Handling large quantities of data, namely wind speed and direction which are more erratic by nature, but some of these proposed methods could apply to other variables

* Running average (focus on noise reduction and cleaner signal but does not help with reducing high density of measurements for long durations of time and tends to wash out the extremes which is mostly problematic for wind directions plots since 0 / 360 for North will appear to rarely occur)
* Static average (focus on reducing the number of data points while retaining all information)
* Every X number of points (focus on reducing the number of data points without smoothing data points; has potential to be more sporadic than a static average plot)

**2020-07-14**

For analysis purposes, consider encoding the ability for a user to choose what data cleansing options they want to implement: chronologically sorted, remove duplicate timestamps, running average, static average, resampled, all, none, or some.

Find a way to simply comment in (or out) options that one wants to use or (not use). You may need to employ functions for this.

When plotting figures without some of the cleansing methods applied, some other lines of code will break down the chain like when trying to plot with the duplicate timestamps, your NaN filling and ‘mintime’ and ‘maxtime’ checker will fail. For the ‘mintime’ ‘maxtime’ part, you can write the code so that if the specific user-defined ‘mintime’/’maxtime’ is within the time frame read in but there is no record for that time, find the nearest timestamp to the user-defined timestamp

**2020-07-14**

Consider creating a TKinter instance with buttons denoting the various user options for each read/plot/analysis program. Given the number of options you are generating almost daily, changing all these lines of code manually will become cumbersome to users.

Consider changing your line plots to bar plots when plotting raw data. This should eliminate linear interpolation between 1-minute-reporting data points *and* gaps in data (more than 1 minute between report times).

Question: why would the network time resetting cause a jump/step in the data values?

Make some plots of truly raw, unsorted data with time frames surrounding previously known backtracks in time to look for clues. (Turn the date/time array into strings so that they plot as they are in the data file, not in chronological order)

Consider comparison tests with statistics on datasets *before* and *after* data processing to gain a measure of improvement.

Consider imposing a two-pass limits test

1. Pass one includes restricting the range to the sensor’s specifications
2. Pass two includes limiting value ranges based on climatic regimes; this pass should not necessarily remove data points, but flag suspect data points

Think about how to remove anomalous spikes in data; this will be trickier because you’ll have to impose rigid criterion that will still weed out 90% of data spikes. It must be broad enough to catch them all (majority) but specific enough to keep real data

**2020-07-14**

Find the maximum duration of constant wind speed/direction such that zero stations ever exhibit this duration and use that as a threshold for flagging and/or removing suspicious data

Consider assigning zero degrees for wind direction if and only if wind *speed* is below some cut-in threshold (1.0 m/s ? what is it for our particular anemometer?); this should help make wind direction plots clearer in terms of when northerly winds are happening.

**2020-07-16**

Step test is simple enough (calculate difference between ith and ith+1 values, if difference is greater than some threshold, toss the second value), but how does this address full blown offsets (as in, the ith and ith+1 difference is larger, but the ith+1 and ith+2 difference is small and so it the ith+2 and ith+3 difference, ith+3 and ith+4, etc.)

**2020-08-10**

Consider calculating uptime between stations and comparing by latitude to gain a sense of the effect of sun angle on uptime. You’ll have to think about how to make this an apples to apples comparison by aligning the times of year site-by-site based on average, daily incoming solar radiation (i.e. normalize by season; meteorological winter in Kenya is far different than meteorological winter in Minnesota); normalize the daily incoming solar radiation curves, then align the time spans of each site based on percentage (where average daily incoming solar radiation at Kenya is 25% 🡪 where average daily incoming solar radiation at UND site is 25%, for example)

**2020-08-10**

Consider trying to *not* sort the data by time before removing duplicate timestamps. This *should* eliminate the occurrence of “jumpy” data, which seems to be happening only because the sort function in python appears to work somewhat randomly, as in it does not simply move a duplicate timestamp found later in the file immediately below the first occurrence of said timestamp. In some instances, it may put that timestamp *ahead* of the first occurrence. Since the data values associated with these like timestamps are different, the data appears to jump around. Leaving the data unsorted then removing duplicate timestamps (but keeping the first occurrence) *should* result in only one jump/step in the data. Alternatively, we could also keep the LAST occurrence, still resulting in only one jump. This still, however, does not answer the question, which timestamp (set of timestamps) contain the correct data?

See file MCP9808\_20180924 for the CSA site. There is a set of duplicate timestamps that will not be caught by the “times out of order” section in your code because they are not out of order. There are simply 4 of them right in a row in the raw data file! All with the same data values too! Fascinating. They will, however be caught by the duplicate timestamp collector, so you should not just collect timestamps out of order, but also duplicate timestamps after the resorting. **Or better yet, only collect files associated with duplicate timestamps because this will catch both duplicate timestamp occurrences resulting from reset time *and* straight up duplicates.**

~~There is something fundamentally wrong with my method for collecting duplicate timestamps. It did not catch those times mentioned in the paragraph above.~~

~~Apparently the pandas ‘remove duplicates’ method did its job, but my method to collect duplicate timestamps did not catch them for some reason…~~

Scratch all that worry… I was looking in the wrong year for those timestamps… *all is well with the world.*

Cannot compare timestamps to those from the CHORDS site because they are the same, so it would seem that the ability to verify whether or not the time was simply starting where it last left off in some files, is gone… So the mystery as to why a timeframe for the MCP9808 (and other sensors recording temperature) for the 20170828-30 duration does not coincide with the diurnal cycle (nor does is match the Marshall site or KDEN) remains…

For the HTU21D sensor script, consider calculating Fahrenheit yourself in the program from Celcius and compare it to Fahrenheit that gets read in. This should help you verify that it is indeed the Python library used for the conversion and subsequent data logging is to blame for the occasional discrepancies between the occurrences of data spikes between the two variables (temp-F and temp-C)

Consider writing individual plotters for daily, weekly, and monthly plots. Still keep your “user’s choice” plotter.

* For the monthly plotter use this stuff
  + df\_by\_year = df[df.set\_index(‘time’).index.year == 2017]
  + df\_by\_month = df\_by\_year[df\_by\_year.set\_index(‘time’).index.month == 12]
* Parse first by year, then by month
* You can use ‘for’ loops to do this for each year and then each month within each year (a double ‘for’ loop
* You should start by identifying the year and month of the very first timestamp read in, as well as the year and month of the very last timestamp and use those as bounds in the ‘for’ loop somehow
* Also consider how you’ll incorporate your naming convention for the purpose of saving figures; you should have each figure automatically saved with a name the reflects the year and month of the plot

*Consider retaining the ability for a user to specify their own limits for which the daily/weekly/monthly plots will only be created within the timeframe set by the user. This will require a serious number of error catchers (assuring the timeframe is appropriate, etc.)*

**2020-08-10**

Look up the cost of adding a simple inclinometer to the 3D-PAWS stations that logs the pitch and roll as often as the other sensors on board.

See if you can convert the ‘for’ loop and ‘if’ statements in your WIND\_plotter that change speed (direction) values to NaNs if its corresponding direction (speed) value is a NaN to the short hand version for NaN conversion you used in your RAIN\_GAUGE\_plotter; it might speed up the runtime!

**2020-08-17**

Think about possibly creating the METAR-like readings (e.g. 3-hour pressure trend, 6-hour precip, 24-hour precip, etc.)

Compare all duplicate timestamps / moments when time was reset between each sensor. If the same (or pretty much the same), create a model for expected, clear day radiation and compare it to the radiation sensor data. See if it is possible to match the suspicious sections of time with the model to gain a sense of the time correction offset, then apply the correction to the other datasets (again, only if the time resets match closely with that of radiation sensor)

Consider developing ways of deducing snow depth based on the surface conditions from other sensors and the precipitation readings. Going to need lots more data for this especially in locations that receive snowfall

**2020-08-17**

In your daily/weekly/monthly plotter(s), add the ability to skip intervals for which all data within that interval are NaNs and return to the user which intervals were skipped. There is no real point in plotting non-existent data, but perhaps in a large dataset, a user would not know that some intervals do not have data, so it is still useful to tell the user where those instances occur.

**2020-08-24**

Consider shifting the user’s ‘mintime’ choice to the nearest 00Z; I’m really on the fence about this as it is mainly for time-axis aesthetics. It will ensure a label at the beginning and end of the time frame, but poses a lot of programmatic headaches in order to make it work properly. It may not be worth the reward

**2020-08-24**

You could likely modify you ‘maxtime’ adjustment from setting it to “-1”, to finding the index of the last time read in. This would eliminate the need to do anything fancy with the ‘end’ variable in your daily/weekly/monthly plotter loop.

For example:

maxtime = ﻿time.get\_loc(time[-1])

OR

maxtime = ﻿df.index[-1]

…

end = maxtime + 1 #necessary because the range function does not include

OR

end = len(df) #notice: no need to add ‘1’ here

**2020-08-24**

Need to run a few simple experiments with my sensors to see if I can recreate some of the issues I am seeing within the CSA dataset, in particular, the significant bias beginning in late 2018 through May of ’19.

**2020-08-24**

Think about how you could flag data after the station loses power and is not able to collect the network time right away when power is restored. Without the ability to know what date/time it is while powered down, it might come down to a necessity to flag data based on the values themselves. For instance, if power goes down, and returns several hours later, temperature may have changed significantly. This poses a problem for instances such that power goes down for a full 24 hours. Assuming a high-pressure weather scenario, you can expect that the temperature at the same time the next day will likely be pretty comparable. Alternatively, the better indicator might be the radiation sensor. This has a much more consistently predictable pattern on a daily basis. Again, problems still occur for 24-hour power outages, or more dynamic weather patterns present.

Instead of stopping the weekly plotter at the last full week, set maxtime to equal ‘end’! this way it will plot everything!!!

**2020-08-24**

Consider adding timers to the various processes, especially your WIND\_plotter, to see where improvement on your programming could be improved.

**2020-08-24**

Consider a singular processing file that, based on user inputs, “flips switches” i.e. calls certain functions. In other words, make every major processing section of your programs (virtually all of which are universal) a specific, standalone function in a program so that the plotting program will load the program and call the functions within based on user inputs. Some functions will be called no matter what, like handling the duplicate timestamps.

In addition to this “end all, be all” processor program, it might be nice to have a GUI with buttons and places for a user to input things like the y-axis limits and such.

To add to this idea, ensure that the “output” file you end up creating is an option too. There is no need to calculate all of those things every single time and create an output file that stores said information all the time. If a user only wants the output file, then don’t plot. Or vice versa.

**2020-09-02**

Make some error checkers to ensure that the input and output directories that the user specifies actually exists, *and* if they have data…

Need to fix your “uptime” calculation to include instances for 100% uptime (printing the duration)

Some datasets have different number of variables for the same sensors (e.g. WMO\_HIGHWAY\_01 does *not* record seconds in the timestamps for the wind sensors, but the CSA dataset does…). Need to account for these differences.

**2020-09-10**

Consider making a plotter for all sensor variables (e.g. for the BMP280, one plotter including just the measured variables [temp-C and station pressure], and one including *all* variables [temp-C, temp-F, SLP-inHg, SLP-hpa, etc]); do this for all sensors that measure or calculate more than one variable.

Dig into the BMP280’s specification that ±0.12 hPa equates to a relative accuracy in altitude of ±1 m; this ought to change with lower pressures at higher elevations, as in, the accuracy of altitude will decrease (±10 m, perhaps, depending on how high in elevation you are)

Think about documenting the nomenclature used recording the data and for pushing data to CHORDS. This is specifically set within the python programs that run the Pi and sensors, but no where is it documented (that I am aware of). Right now, my RPi was not pushing ALL variables to CHORDS from each sensor. I had to modify the program so that it would.

Use Google-Earth-determined elevations as a baseline for calculating the accuracy of the altitude calculations.

For the BMP280 spikes, consider a sort of two-layer approach for weeding out those 27.08 values:

1. First check if it exceeds some threshold (an arbitrarily chosen limit, or based on a standard deviation) based on the surrounding values. If this threshold is exceeded, then…
2. Check if there are multiples of them (the exact same)
3. Remove those that meet both criteria???

Alternatively, just remove all “27.08 C” values. Perhaps, first count the number of times that that values occurs within the existing dataset (do *not* include missing reports/NaNs), to understand how often they occur. And how often they occur consecutively.

Also, think about calculating the standard deviation of every value for a 5, 10, 20, 30, and 60-min mean. Use this as an assessment for the validity of each value. Note that this could be used for identifying fronts (might be a nice way to develop an algorithm/function for finding/identifying fronts, especially when used in conjunction with multiple sensors.