Shipboard Automated Meteorological and Oceanographic System: Merger Program Re-Write

Camill Folsom

July 2019

Major Professor: Sonia Haiduc

Committee: Shawn Smith, David Whalley, Zhenhai Duan

Acknowledgments

SAMOS is base funded by NOAA's Ocean Observing and Monitoring Division (grant # NA11OAR4320199*)* and the U. S. National Science Foundation's Oceanographic Instrumentation and Technical Services Program (grant # OCE-1447797). Since 2013, the Schmidt Ocean Institute (SOI) has funded participation by the *RV Falkor* in the SAMOS initiative.

1 Introduction

1.1 The SAMOS Project

The Shipboard Automated Meteorological and Oceanographic System (SAMOS) project started collecting data from research vessels in 2005. The SAMOS project collects, distributes, quality-evaluates, and archives meteorological and oceanographic observations. Currently SAMOS receives data annually from around 30 research vessels. Each vessel continuously performs automated data logging with sampling intervals of 1 minute or less. These systems are usually mounted on the bow or on the mast over the wheelhouse. The vessels perform navigational, meteorological, and oceanographic observations. The observations then undergo quality control analysis and are distributed to the scientific community. The SAMOS project focuses on improving the quality of these observations.

Each research vessel's data are averaged to make sure each data recording is in 1-minute intervals. All the 1-minute records are then collected for a day packed into a file and delivered, via an email attachment, on a daily basis usually at midnight. Each vessel tracks data for a different number of variables such as time of observation, latitude, longitude, etc. The SAMOS variables can be seen in Table 1. Each variable has data for each 1-minute interval. The daily files for any vessel can have a maximum of 1440-minutes of data, for each variable they track, because this is the maximum number of minutes in a day. Along with the variable data each ship is required to send detailed metadata about the ship and each variable. The ship's metadata is loaded into a ship profile in a MySQL database so the metadata can be tracked for changes such as instrument swaps, new sensors or sensor location, etc.

Primary	**Secondary**
Observed time (UTC)	Vessel Pitch, Roll, and Heave
Latitude	Photosynthetically Active Radiation (PAR)
Longitude	Ultraviolet Radiation
Ship Course Over Ground	Total Radiation
Ship Speed Over Ground	Visibility
Ship Heading	Ceiling
Ship Speed Over Water	Fluorescence
Ship-Relative Wind Direction	Dissolved Oxygen
Ship-Relative Wind Speed	Radiometric Sea Surface Temperature
Earth Relative Wind Direction	Swell and Wind Wave Heights and Directions
Earth Relative Wind Speed	Weather, Cloud Cover, and Cloud Height
Atmospheric Pressure	
Air Temperature	
Moisture, Precipitation	
Shortwave Radiation	
Longwave Radiation	
Sea Temperature, Salinity, and Conductivity	

Table 1: Primary and secondary parameters for a routine data acquisition from a SAMOS vessel. Extracted from: https://samos.coaps.fsu.edu/html/parameters.php

The flow of data processing starts when the SAMOS project receives an email from one of the research vessels containing the observational data for a given day, via an email attachment. The data each vessel sends is in a SAMOS data format which uses a custom key-valued paired comma separated value (CSV) format. Once the data are received it is converted into a network Common Data Form (netCDF) file. The data that is received from each vessel could contain data for only part of a day and there could be multiple pieces sent for the daily file's information such that a single day can contain multiple files of data for a single vessel for a single day. These interruptions in data sending could be caused by a ship being docked for maintenance or by lack of availability of a satellite transmission bandwidth.

The next step involves the CSV files undergoing a series of quality control (QC) steps. The first part of this process is fully automated and produces the preliminary (version 100) netCDF data files. The first step in the quality control process involves validating that the file came from one of the recruited research vessels and that the data is in the proper SAMOS CSV format. Next the file's data are verified, and it undergoes automated QC to apply flags to each variable's 1-minute interval data values. The flags are alphabetic characters A-Z, with the Z flag representing data that has passed all QC evaluations, see Table 2. The automated QC starts by assuming all data values are correct and assigns a Z flag to all data values. As the auto QC process runs, the Z flags are changed into the appropriate lower value flags if needed.

Flag	Definition
B	Original data were out of a physically realistic range bounds outlined.
D	Data failed the T>=Tw>=Td test. In the free atmosphere, the value of the temperature is always greater than or equal to the wet-bulb temperature, which in turn is always greater than or equal to the dew point temperature.
E	Data failed the resultant wind recomputation check. When the data set includes the platform's heading, course over the ground, and speed over the ground along with platform-relative wind speed and direction, a programme recomputes the Earth-relative wind speed and direction. A failed test occurs when the difference between the reported and recomputed wind direction is >20 (or >2.5 m/s for wind speed).
F	Platform velocity unrealistic. Determined by comparing sequential latitude and longitude positions.
G	Data are greater than four standard deviations from the climatological means (da Silva et al. 1994). The test is only applied to pressure, temperature, sea temperature, relative humidity, and wind speed data
H	Discontinuity (step) found in the data. Flags assigned to the maximum and minimum points in the discontinuity.
I	Interesting feature found in the data. Examples include: hurricanes passing stations, sharp seawater temperature gradients, strong convective events, etc.
J	Visual inspection shows the value to be erroneous/poor quality. The value should NOT be used.
K	Data suspect/use with caution – Applied when the value looks to have obvious errors, but no specific reason for the error can be determined. Some data may be useful, but uncertainty would be high, and use is not recommended.
L	Vessel position over land based on reported latitude and longitude.
M	Known instrument malfunction
N	Signifies that the data were collected while the vessel was in port. Typically, these data, though realistic, are significantly different from open ocean conditions.
Q	Questionable – observation reported as questionable/uncertain in consultation with vessel operator.
S	Spike in the data. Usually one or two sequential data values (sometimes up to 5 values) that are drastically out of the current data trend. Spikes occur for many reasons including power surges, typos, data logging problems, lightning strikes, etc.
Z	Data passed evaluation.

Table 2: Flag values for the SAMOS project (Smith et al. 2018)

Each ship could have multiple version 100 files for a single day. To distinguish between multiple version 100 files for a single vessel and day an order number is assigned to all SAMOS versioned files. The order numbers begin at 01 and are incremented for each file present for that day and version number. Every ten days the multiple version 100 files for each vessel and day are automatically merged into a single intermediate (version 200) netCDF data file. The delay allows for the receipt of corrected files from previous days. Due to this a single day for a vessel could have multiple version 100 files that contain all 1440-minutes of data for that day. The last part of the quality control process involves having select ships undergo visual quality control which creates research-quality (version 300) netCDF data files. The overall SAMOS data flow can be seen in Figure 1.

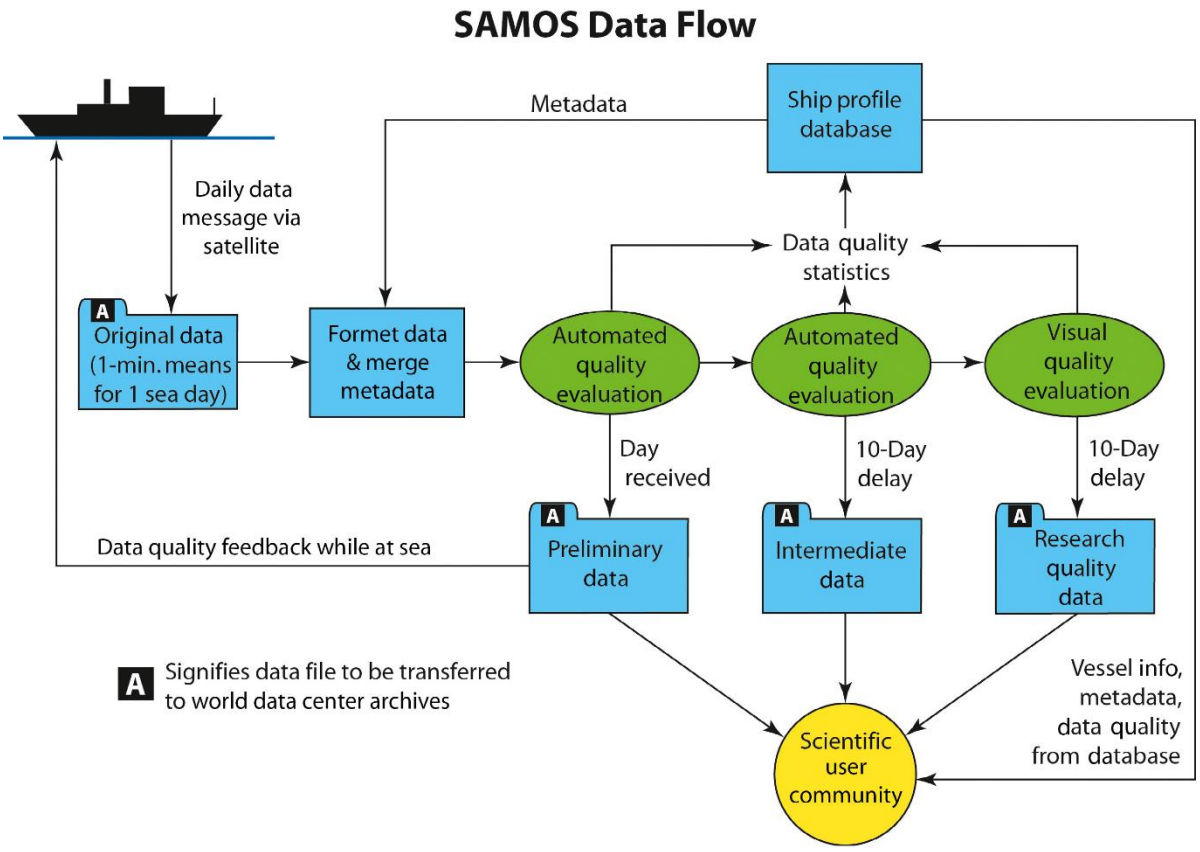

Figure 1: The flow of data processing for a single vessel through SAMOS and to the user community. Each vessel has its own profile in the database. Both the data and metadata are integrated into a single netCDF file that is distributed to the user community (extracted from Smith et al. 2018).

1.2 netCDF Files

A netCDF file contains data for each 1-minute interval of data each vessel collects with files ranging from having a single, up to 1440, 1-minute records. Each variable dimension defines the size of the Numpy array of data each variable is given. NetCDF files have an unlimited dimension that can grow to any size as data are added to the array. Other dimensions allow for only fixed size arrays. A variable may have multiple dimensions in which case it will be a multi-dimensional array.

Each netCDF file, for SAMOS, contains four dimensions time, f_string, h_string, and h_num. The time dimension is unlimited, f_string and h_string define the length of any flag or history string, and h_num represents the total number of history strings that may be stored in the history variable's array of strings. All variables in a netCDF file have their size defined by the time dimension because each variable should have a length equal to the number of time observations in the file. There are a few variables that use dimensions that are not the time dimension, these variables are history and flag. Every netCDF file has metadata attributes for every variable that describes the variable. They also have global metadata for each file that describes the file.

The number of flag strings is determined by the time dimension, but it also has a second-dimension f_string since the flag variable contains an array of characters that represent each variable's assigned flag value at a given minute. The f_string dimension is equal to the number of flagged variables in the file because each flag string must have one entry in the string for each variable for a given minute. For example, if a file only has 1-minute of data then it will contain only a single flag string where the length of the string was determined by the number of flagged variables in the file.

The history variable is the only variable that does not use the time dimension and it is instead defined by h_num and h_string. Here h_num represents the max number of history strings that can be stored and h_string represents the max length the history string can be.

The time of observation variable has data values that represents a single minute since 1/1/1980 at midnight UTC and records the minute when the observations for each variable was taken. The variables time of day and date are not flagged variables since they are derived from the time of observation value. The other non-flagged values are flag and history. Time of day is simply the time of observation turned into a time of day. The date variable is the files date repeated for the entire length of the array which has length given by the time dimension.

With a few exceptions, a variable's data value can be missing, special, or a data value. A missing data value can occur when for a minute in the netCDF file a variables data value does not have a value for that minute. If this is the case it will have a value of -9999 written to the array of the variable's data value where the data is missing that corresponds to that minute. A special value occurs when there is data for a variable's data value for a specific minute however, one of the following occurred. Either the value did not fit in the code range or the value overflowed, and it does not fit in the memory space allocated. Special values are written as -8888. The new program introduces a third fill value, no-decision, that is used when the merger program cannot decide on

the duplicate minute's variable data values. No-decision values will be written as -5555. An example of a netCDF file with a single minute of data can be found in Appendix A.

1.3 The Role of the Current Merger Program

Because each vessel can have multiple preliminary (version 100) netCDF files a merger program is used to merge every preliminary (version 100) netCDF files for a vessel for a single day into a single intermediate (version 200) netCDF file. As the files are merged the merger program identifies duplicate minutes. For every duplicate minute, data values for each variable are determined. Once a data value is identified the value to keep is determined through a series of tests. The first test checks whether the data values are data, missing, or special data values. It chooses from these values using the ordering data > special > missing. If both values are data then the preliminary QC flags are used to determine which value to keep, using the ordering of Z>F>L for latitude and longitude and Z>G>E>B>D for other parameters. If the values have the same flags and the data values are different, then the values to keep is determined using a 30-minute window to calculate the mean for both data values. The data value that produces the smallest mean is kept. If the means results in duplicate values, then no value is written to the file for that data value.

1.4 Motivation for Code Rewrite

The current merger program is written in Pearl and the netCDF library that is used is going to stop being supported soon. The current implementation also has plenty of known bugs where it incorrectly mergers netCDF files (see section 2). The goal of this project is to re-create the merger program using Python without re-producing any of the known bugs in the current program. This also provides a good opportunity to add new features to the original code during the upgrade process. Finally, the SAMOS project intends to go open source, all the code is being re-written in Python, and the project wishes to start using a Git version control system, such as Atlassian Bitbucket.

2 Known Problems

The current program has several known issues where it either fails to merge the files or it incorrectly merges the files. It can cause the correct information to be written in an incorrect order. This problem only occurs when a variable shows up in the first file being merged. If the variable is not in the first file being merged but in any later file being merged, then the file's data can be written out of order.

When merging files that contain a different number of variables the current merger program can have a few problems. It will sometimes mix up the placement of E flags that were once present on the direction and speed variables and it re-assigns these flags to incorrect variables. The next problem it has is it will sometimes not keep the correct flag for a variable turning flags, such as the E flag, into the Z flag. Sometimes it will assign special values to all the data values for a variable that does in fact contain data in the preliminary (version 100) files. The program will sometimes fail to merge the files given that there are a different number of variables and one file

contains 1-minute of data and the other file contains 1440-minutes of data. Finally, a flag can also be written as a non-character given files that have a different number of variables.

If merging more than one identical 1-minute files for a single day together with any number of files that contain more than one minute of data, the current program fails. If this happens the program reports an "unknown error". At least once in this situation the program did not fail to merge the files. However, it incorrectly assigned E and F flags to most of the variables in the file.

3 Re-Design

The re-design of the merger program will follow a software requirements specification developed in collaboration with the SAMOS data quality analysts, lead programmer, and data center manager. A summary of these requirements is provided in this section.

3.1 Inputs

The merger program will take in any number of netCDF preliminary (version 100) files and produce a single intermediate (version 200) file. The order in which the variables appear in a file must adhere to the variable ordering as defined in the SAMOS MySQL database. The merger program must contact the database to retrieve this information.

The merger program will now support an optional command line argument that will allow the user to define a file and variables that are to take precedence in the merge process. The user is required to enter both the file identifier and the variables as a pair, and they can define any number of files with variable precedence. The variables cannot have precedence in more than one file. The variables will also allow for the user to use the '*' wild card character to specify the variables in a file that have precedence. To specify the file a version and order number will be given to specify the file with precedence followed by a comma separated list of variable names.

3.2 Merge Rules

The following validity checks will now be performed on each minute. Any time value must have a Z flag. If any time does not have a Z flag, then that minute will be skipped for all variables present in the file for that minute. Additionally, no time, latitude, or longitude variables values can have anything except data, i.e. no special, missing, or no-decision values will be considered. For the new merge program if a decision cannot be made on duplicate minute data, then a no-decision value will be used for that variable’s value.

The program needs to be able to deal with the several different variables situations. Given a minute that only exists in a single file, then that minute will be kept in the merged file. The code must account for files that have overlapping and/or different times, all the same variables, and a different number of variables. The following variables must be treated as pairs: course and speed over ground, platform relative speed and direction, and true wind speed and direction. The variables longitude and heading are treated as unit vector and are paired with one for their corresponding speed values.

3.3 Deciding Which Values to Keep

To decide on which values to keep given duplicate minutes with conflicting variable data, then for all variables in file1 that are also in file2, for the duplicate minute, following procedure is used to determine which value to keep. Given that the duplicate data values and their corresponding flags are the same, then the first value and flag is arbitrarily kept. The first comparison is made when the values are different is checking if the value is data, special, missing, or no-decision, where the ordering is given as data>no decision>special>missing. The no-decision value is new to this program. Before if a value could not be decided upon it was written as a missing value, -9999. For this program if a decision cannot be made it writes a no-decision value, -5555, for the duplicate minute's variable data value.

After determining whether the data values are data, the program choses one of the variables if it has been given precedence by the user. Next, it moves on to checking whether the flags differ. If the flags differ, then the value with the highest (best) flag will be chosen. The new program uses a somewhat different flag ordering than the previous program. For latitude and longitude, the flag ordering is given by Z>B>F>L and for all other variables the ordering Z>G>E>B>D is used. If the flags are the same, then a comparison test must be run. The decision processes can be seen in Figure 2.

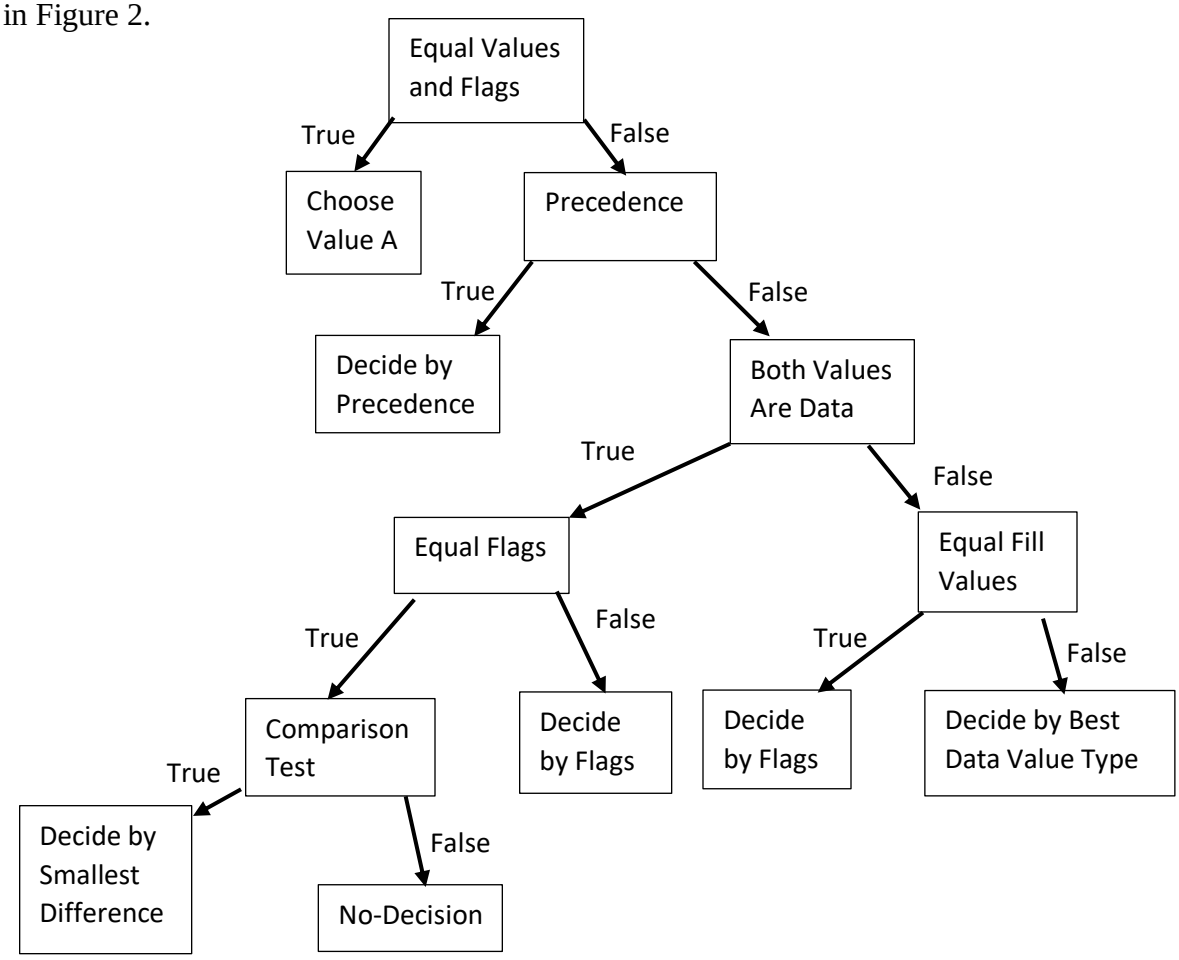

Figure 2: A decision tree for deciding which data value to keep for a duplicate minute.

3.4 Comparison Testing

The old program's last step in its decision-making process was to calculate the mean for a 30-minute window of the data, which is only Z flagged 'good' data, for both data values in question. The window must have a minimum of 15-minutes of data and the program could skip no more than five data values when constructing a valid window. To decide on the value to keep, the old program would calculate the mean on the window without the value in question, the control mean, and again with the value in question added back in. It finally subtracts the control mean from the mean that is missing the value in question and it selects the subtraction that has the smaller value. If the subtraction of the means and control mean result in an equal value, then the old program writes this value as missing.

The new program makes the decision on the data values to keep at duplicate minutes that have the same flag by using a sum of squared differences rather than by using a mean calculation. This test only requires the window to have 10-minutes of Z flagged 'good' data in the window for a calculation to be done. Unlike the previous program this program will not exclude windows with greater than 5 skipped minutes. It also makes sure to center the window on the value in question attempting to get as many points as possible from either side of the value in question with a max of all 30 values in the window being on one side of the value in question. This only happens if no values can be obtained in the window on one of the sides of the center point of the window (e.g., near the beginning or end of the file). Ideally the program tries to get 15 values from either side of the value in question. The formula used to calculate the sum of squared differences is given below:

$$differemce = \frac{\sum(x - x_i)^2}{N}$$

Here x is the value in question, x_i is one of the values in the window i=1,2,3,…,30, and N is the number of values in the window. The sum of squared differences is calculated for each value in question. The window that results in the smallest sum of differences is the value that is chosen. If the sum of differences for both the values in question are within 0.000001 of each other, then no-decision can be made, and a no-decision value will be written to the file for the value in question.

For deciding on the direction variable, a vector is used, and a unit vector is used for longitude and heading. The speeds are always treated as any other variable as they are represented by scalar values, but they are paired with direction to calculate the vector. The directions are also treated as any other variable up until the point where the comparison test must be used to decide.

The old program uses a somewhat lengthy method to achieve this calculation. First it calculates the values u and v using the following equations:

$$u = speed * \cos(270 - direction * (3.14159/180))$$

$$v = speed * \sin(270 - direction * (3.14159/180))$$

Here the 270 value is used to deal with the unusual coordinate system used which flips the x-axis and y-axis. It then calculates the mean on u and v. The means are then converted to a vector for

both the speed and direction. The direction's mean is then used to subtract from the directional value in question and the value that gives the smallest subtraction is chosen. Note that due to the fact the direction is converted back into degrees for use in making the comparison if a value is greater than 180, then the value must have 180 subtracted from it to make sure it chooses the smallest value. This is because the values in degrees 0 are equal to 360. When calculating the unit vector, the only difference is that the speeds have a value of one.

The new program does vector comparisons in a slightly different manner. Like the original program it calculates the values for u and v for all directional values. However, unlike the old program it will use u and v to decide instead of converting these values back to degree values. Because of this the calculation is simplified and we get the equations below.

$$u = speed * \cos(direction * (\pi/180))$$

$$v = speed * \sin(direction * (\pi/180))$$

Here we no longer need the value 270 since we are doing are comparisons using u and v. Note that here the equation uses π instead of 3.141519 since this value was used in the old code but the new code is using the Numpy module's 'deg2rad' function to convert from degrees to radians. The sum of squared differences is then calculated using the formula.

$$difference = \sum((u - u_i)^2 + (v - v_i)^2)/N$$

Here *N* is the number of elements in the window, *u* and *v* represent the vectors in question, and u_i and v_i represent the directional vectors in the window. The directional vector that results in the smallest sum of differences is chosen. For the unit vector we use the same equations except we use a value of one for the speed.

3.5 Logging

The new program will output three log files whenever the program is run. All log files will use the naming format of SHIP_YYYYMMDDvVVVOO_type.log where ship is the callsign for the vessel, YYYYMMDD is the date of the files being merged, VVV is the files version number, OO is the files order number, and type is the log files name, i.e. main, error, or no-decision. The order number for the file is obtained by contacting the database to determine the last version 200 file that was merged, and this file will have an order number that is one greater than the last version 200 file for this ship and day.

The main log file has all logging information written to it. The error logging file will be an empty file unless an error occurs in which case the error is written to the error log file as well as the main log file. The no-decision log file is only written to when no-decision can be made at a given minute. The main log, aside from the data from the error log and no-decision log, also contains all the debugging information that the program logs. Lastly, if no netCDF files open successfully due to an improper file path or filename, then the three log files will still be generated but will use a fall back naming scheme where the name will be YYYYMMDDHHMMSS_type.log where YYYY is the year, MM is the month, DD is the day,

HH is hours, MM is minutes, and SS is seconds. This time and date represent the current local time in which the program failed to open any netCDF files.

3.6 Metadata

The netCDF files contain a history variable that is used to track the creation of the netCDF files. When merging the files together the history variable must gain the history from each file that is being merged and there will be a single line written to history by the merger program that records the creation of this version 200 file. The history variable must not exceed the size of its dimensions. If the history approaches the max number of history lines, then there is always a single line reserved by the merger program so that it can write its own history value.

Every variable in the netCDF file has its own metadata. This metadata includes data such as the actual range of the values, the long name of the variable, the instrument used for the observations, etc. When merging the files, the metadata for each variable will be taken from the file with the most recent metadata modification date if that metadata attribute exists. All variables shall also receive a new variable metadata attribute that tracks the name of the file from which it was retrieved. The program will also update the actual range of the variable’s data values with the new range after the merge process has finished.

Every file has its own global metadata. The global metadata for the new file shall be obtained from the file with the most recent metadata modification date. The file's metadata and data modification date shall both be updated to the current local time in the ISO 8601 standard format. The global metadata will have a few attributes added. An attribute that tracks all the files that were merged to create the new version 200 file. It will also add an attribute to track the program’s version number and commit hash from a git repository.

4 Implementation

The re-creation of the merger program will be written completely in Python. To read in the command line arguments the argparse module was used. This module allows for the user entered command line arguments to be easily parsed into the program. The user may enter any number of path/filenames to be merged or they can enter path/filenames with wildcard characters to specify multiple files in a single line. The user can also optionally define the variable precedence.

The log files are created and managed using the logging module. The logging module is set up so any logging can be done using the logging base class. This is accomplished by getting an unnamed logger and then file handlers are added to the base logger. This allows the base logging class to be used to log all messages.

4.1 Opening the netCDF files

Each netCDF file is read into a list using the netCDF4 python module, using the Dataset class. Upon opening a netCDF file as a dataset, the property 'set_auto_mask' needs to be set to false, by default this property is set to true. When this property is true any values that are considered fill values are represented with a boolean value rather than a numerical value. This would cause a problem as the program needs to check for any number of fill values that have an ordering given

to them. For this implementation the fill values are no-decision, special, and missing with values -5555, -8888, and -9999, respectively.

Once the first dataset is opened, the name of the output version 200 file can be determined as well as the name for the log files. To create the name of the log file, the program must first get the vessel's call sign and the date for when the time of observation started. This information can be retrieved from the first opened dataset. The order number for this file can be retrieved by querying the database using the vessel's call sign and the date to get the vessel's last version 200 order number for that day. The order number for the new file will be the old order number with one added to it. While the connection to the database is opened, the ordering that the variables are to appear in a file is read from the database.

Once the order number for the current file has been determined, the program creates the name for the error log file and then it creates the error log file by calling a function that adds a file handler to the logging class. The error file has the logging level set to warning so that all error messages will be logged to the error log file. The log files for the main and decision logs are created by replacing the word error with main and decision, respectively, in the log files name and then adding a new log handler with level debug for the main log file and level info for the decision log file. Because the decision log is set to logging level info, any error logs would be written to this file if a string was logged to any error level. To solve this problem an 'info_filter' function was created to make sure only strings written to info would be written to the decision log file.

As the other netCDF datasets are read into the list, they are checked to make sure each call sign and date match for each file being merged. If any of the files being opened do not match the first opened file's date and call sign, then the program will exit with an error and log the error to the error and main logs. The program then logs the call sign, date, and filenames involved in the merge to the main log file. Each time the program opens a dataset it uses the Python 'os' module to get the filename from the path used to open the dataset. It maintains a list of filenames as it opens the datasets. If a filename exists in the list of filenames before it is added, then the two files with the same name are the same file and the program will exit with an error. Otherwise, the program will append the filename to the list of filenames.

At this point the merge function is called to merge all the datasets. It returns an in-memory dataset that contains the merged file information. A new non-in-memory datasets is created, and it has all the data from the in-memory dataset copied to it. Once this is finished the new merged file is closed. The in-memory dataset is also closed, and the list of datasets has each element closed. At this point the program has finished the merge and it exits.

4.2 Merging the Datasets

The first thing that is done when merging the dataset is to check that the datasets are all version 100 files that are being merged. If any of the files are not version 100 files, the program exits and logs the error. It then checks to see if the user specified any variable precedence. If they did, this function calls another function to parse the user entered precedence. The precedence function returns a dictionary with the index into the list of datasets as the key and the list of variables that have been given precedence by the user for that file mapped to it.

The merge function then calls a function to create an in-memory dataset to hold the merged data. The function merges all the data from every dataset in the list of datasets, except for the data values for each variable. It creates a netCDF file with dimensions, variables, and metadata, but the variables are assigned no values. After retrieving the no data value dataset, the auto_mask must be set to false or when writing data to the file that is a fill value the value will not be written.

Next, the times that need to be skipped are determined. These times are times in which the time value has a non-Z flag. The only other reason a time is skipped is if the time, latitude or longitude has any value that is a fill value as these attributes cannot have fill values. When a minute is skipped it means no data from any variable for that specific minute is used in the merge.

Now the merge function has all the information it needs to create a merger_dataset object. A merger_dataset is a custom-made class that is used to parse the data from a single dataset to make the merging of data easier and more efficient. The class has a function built into it for merging one merger_dataset with another. It also has a function that takes in a list of merger_datasets objects and it returns a single merger_dataset object with all the variables from each merger_dataset object merged into a single merger_dataset object. The merger_dataset keeps a dictionary of variable names mapped to another dictionary of time values mapped to a tuple of (value, flag), i.e. {variable_name:{time:(value, flag)}}.

Once the data has been merged into a single merger_dataset, the data needs to be read from the merger_dataset into the in-memory dataset. First, the history is read in because it does not contain any flags. Each history string from each merger_dataset is loaded into empty spots in the history array when the merger_datasets are merged together. The history value just needs to have the new creation time stamp from the merger program added in.

The remaining data in the merger_dataset's dictionary needs to be written in the proper order described by the variable order obtained from the database. This is because when writing the flag values for a variable, the variables are appended to the flag array, column wise. If the ordering is not considered, some flags will show up in an incorrect location. On the first iteration of the loop through, the variable data time is retrieved from the dictionary by unzipping the dictionary to get times values. Time has all Z value flags since all other flags were excluded for time so a numpy array full of Zs with length equal to the time dimension will represent the time flags. The values are retrieved by unzipping the nested dictionary to get arrays with all the times, values, and flags for a variable name. These data arrays are then used to set the no value netCDF dataset's values.

The time of day and date are left out of the loop since they are calculated on their own. For time of day the values are retrieved from the time values converted into a time of day. The date is the files date repeated for time dimension number of times. The new start and end dates are written to the merged datasets global attributes. Finally, the function updates the merged datasets actual ranges for each variable by getting the arrays min and max values and then the dataset is returned by the function.

4.2.1 Determining File/Variable Precedence

The parse precedence function first loops through each dataset and gets the order and version numbers from each dataset's global attributes and maps it to its own position in the list of datasets. It uses the order and version numbers to make an 'version,order' number string. This string is used as a key into a dictionary that maps to the index in the list of datasets where it appears. A check is performed to make sure none of the 'version,order' number strings are in the dictionary before adding the string to the dictionary. If the string exists in the dictionary before adding the string and the string belongs to a different dataset than the one already added, then more than one file contains the same version and order numbers. If this happens, then the program exits and logs the error.

The function then loops through the precedence values entered by the user. The precedence dictionary is constructed using a temporary dictionary to parse the precedence for each file and then the precedence dictionary can be updated with the temporary dictionary. It uses the 'version,order' number string the user entered to get the index for the dataset from the dictionary created earlier. The variable precedence entered by the user is expected to be entered as a comma separated list of variables. Because of this the variable string entered for the precedence is split on commas. If any of these variables do not contain a '*' wild card character and their name is not in the dataset's variable names, then the program will exit and log an error since the variable does not exist in the dataset where it was given precedence.

If the variable name contains a '*' wild card, then the 'fnmatch' module is used to find all the variables found for the '*' wild card. The results from the 'fnmatch' filtering, a list of variable names, are then put into the dictionary using the dictionary's function 'fromkeys' to map the variables to their corresponding dataset index. If the 'fnmatch' filtering returns an empty list, then the program will exit and log the error since the precedence does not describe any variables.

If the user entered variable string does not contain wild cards, then the variable string is split and 'fromkeys' is used to map each variable to the dataset index. The precedence and temporary dictionaries have the variable name keys intersected and the intersection is checked to see if the variables they have in common have different dataset indexes. If they do, then the user gave a variable precedence in multiple files and the program will exit and log the error. If this does not happen, the precedence is updated using the temporary dictionary. To make the precedence easier to use, the dictionary is then looped through mapping all the variables to their given dataset's index, i.e. {dataset_index:[var1, var2, …]}.

4.2.2 Creating a No-Data netCDF Dataset

The first thing that needs to be done is determine what order the variable metadata will be obtained from the datasets. Each dataset has its metadata modification date read from the dataset's global attributes. The date and time are then turned into a 'datetime' object. The datetime, dataset, and the filename are then put into a tuple and appended to a list. If the dataset does not have a variable that gives the metadata's modification date, then the datetime object is replaced with none in the tuple and it is appended to a different list. The list of datetime tuples is then sorted in reverse to make sure the most current date is the first value. If there are any

datetime tuples that have none written instead of a datetime object, then the sorted list of tuples is extended with these tuples that have none instead of a datetime object.

A dictionary is created to map each variable name to a tuple of the dataset's variable and the files name. A variable is only added once to the dictionary since a variable only needs to be created once. Now that there is a dictionary that contains all the names of all the variables in each dataset, the dimensions for the file can be created. The dimension that determines the length of the flag strings value can be obtained by subtracting the number of non-flagged variables from the total number of variable name keys in the dictionary. The dimensions are obtained from the open datasets. The flag string length dimension can be set to the calculated value.

Now that the dimensions are created the variables can be created. Since this is the dataset that will ultimately become the new file and netCDF datasets order variables based off the order they are created, the variable ordering defined in the database is enforced here. Before creating the variables, the difference of the variable names and variable ordering names is checked to make sure the two sets have no variables that are in the variables dictionary that are not in the variable ordering. If this fails, the program exits and logs the error. The sorted function is used to make sure the variable names are looped through in the proper order. Each variable is created based off the variable's name, the dataset's variable data type and dimensions. The metadata is grabbed from the dataset for this variable and the variable has a retrieved from attribute added to it that records the filename from which the metadata was retrieved.

Each variable has an attribute that defines its quality control (QC) index. This index is equal to the variable's position in the file, if the variable has this attribute, except for time of day and date which must have a QC index equal to times QC index. The QC index is maintained by a count variable that is used to count the variables as they are created. The program must separately add flag and history to the end of the file than the other variables because they are currently not given an ordering in the database.

The last issue that must be handled is the global metadata attributes. The most recent dataset's global metadata is copied, from the dataset with the most recent datetime, and attributes are added with new information. The version and order numbers are updated to reflect the merge. The metadata and date modification dates in the global metadata attributes are updated to reflect the time when the file was created by the merge program. The merger program keeps a global variable with the programs' version number. The version number is written as a new attribute. Lastly the Git commit is written to a new attribute if it can be obtained from the .git folder. Otherwise, unknown is written for the commit_hash attribute. The in-memory no-data netCDF dataset is then returned by the function.

4.2.3 Merger Dataset Class

This class was created to facilitate the merging of the netCDF dataset's variable data. Each merger_dataset creates a variables dictionary that holds each variable's data values, flags, and a bool that determines whether or not this variable and minute has precedence. This tuple is mapped to the corresponding time minute as a key, i.e. {variable:{minute: (value, flag, bool)}}. This dictionary is used to link each variable name, time value, data value, and flag since this data

is needed to accomplish the merging of variables. This dictionary is made by using the ‘zip’ function.

Each variable in the dataset has all its data values in the Numpy arrays. Every variable, except history, that we wish to merge has the same dimensions defined by time. Because of this we can zip the variables data values with its flag values. Due to the odd way SAMOS netCDF datasets handle the flags to get each flag for a variable, its position in the dataset must first be determined, then every flag from that column can be read to be zipped with the values. Next, the time values Numpy array is zipped with the zipped values and flags, and the zip is cast as a dictionary. Finally, the newly created dictionary is mapped to the variable name key in the variables dictionary.

There are a few variables that do not get an entry in the variables dictionary. These variables are time, flag, time_of_day, and date. Time is not included since it is essentially the keys for each dictionary that is mapped to a variable name. The same is true of the flags as they are kept paired with the data values, so there is no need to maintain a key for the flag variable name. Time_of_day is derived from the time variable so there is no need to include it in the dictionary. Date is just a repeated number that represent the date for which the file describes data for and does not need to be handled by the merger_dataset merge function.

The history variable is needed but since its dimensions are not time it has to be handle differently than all other variables. The merge program wants to track what file the history data come from so the file name is made the first string in the history array. The history is then simply mapped to its name.

A second good_data dictionary is also maintained. It is almost the same dictionary as the variables dictionary. However, it does not track the flag values or the history variable. This is because this dictionary is used to do comparison testing. When doing the comparison test, we only want to have good data in a window so this structure only holds values that have been flagged by the best flag value, currently the best flag is Z but in the future this could change. Most of the variables have the variable name mapped to the minute key, which is mapped to the data value, i.e. {variable:{minute:value}}. The exception here is the vectors and unit vectors. For the unit vectors they simply have their values paired with 1 to indicate this should be processed as a unit vector. i.e. {variable:{minute:(value,1)}}. For any vector it has its corresponding scalar value paired with it if it exists, i.e. {variable:{minute:(value1, value2)}}, if it does not, then the value is paired with 1 and the vector is treated as a unit vector. For SAMOS the vectors are directions and they are paired with scalar speed values (e.g., value from DIR, value2 from SPD).

This class contains the function that merges one merger_dataset with another. During the merge process the calling object is updated with any information from the merger_dataset that is passed into the function if that data does not exist in the caller or if it has been decided the passed in variable has the better value. To accomplish the decisions sets are used heavily. The times for a variable are intersected to determine if there are any duplicate minutes for a variable. If there are not the non-duplicate minute are added to the caller’s variables structure. If there are duplicate minutes, then a difference is taken between the minutes in the passed in object and the caller to

get the minutes that are not in the caller's variables structure, but they are in the passed in objects structure. These values are added to the caller's variables structure.

Next the items of each dictionary are intersected, these items are tuples with the form (minute, value, flag). If there are any values in this intersection, then these are exact duplicates and we keep the value already loaded into the caller's variables structure. Next, the unique elements are determined in the passed in objects structure by taking the difference of the set of items in the passed in object minus the duplicate items found. If the difference is empty, then all the elements in the passed in object are duplicated in the caller's structure. If there are values that are not complete duplicates, then the non-duplicate items are intersected with the minutes that both merger_datasets have in common. These minutes are then looped through and the decision-making process from Figure 2 is used to select a value. When making a window, it is done on the good_data structure. Due to this, the window may be created using array slicing since the goal is to simply create a window centered on the minute value in question.

Once the merge is complete the caller has the merged data for itself and the merger_dataset with which it was merged. The class has a static method that takes in a list of merger_dataset objects. It then grabs the first merger_dataset from the list and then resets the list to a slice of the list that does not include the first element. Next it loops through each merger_dataset and using the first merger_dataset from the list as the caller, it merges that object with all other merger_datasets in the list. Finally, it returns the merger_dataset that now contains all the merged variable data.

4.2.4 Orderer Class

To facilitate the comparisons of flags and fill values an orderer parent class was created. It is a simple class that maintains an empty array. Its main function is to create the comparison operators for the class such that the higher position an element is in an array the greater value it has, i.e. the first element in the array is the greatest value. The classes used for flag and fill value comparisons are derived from the parent orderer class. The child classes flag_order and fill_val_order simply define the array for the parent class. These classes are used to do comparisons for flags and fill values using the comparison operators.

4.3 Challenges

In the creation of this program it was discovered that Python is very slow at indexing into arrays. This was first discovered when attempting to create a structure that mapped the minute value to a list of (file_index, value_index) tuples. This caused the programs execution time to increase by a factor of about 4 going from taking ~0.5 seconds to it taking ~2 seconds to run. This limitation was overcome by using sets. It was discovered that the increase in runtime was being caused by three arrays that were being indexed on each iteration of the loop. This can be seen below.

```
for f_index, ds in enumerate(datasets):
    time_index = list(ds.variables.keys()).index(time_name)
    for val_index, time_val in enumerate(ds.variables[time_name][:]):
```

```
flag_val = ds.variables[flag_name][val_index][time_index]

lat_val = ds.variables[lat_name][val_index]

lon_val = ds.variables[lon_name][val_index]
```

Here getting the flag, lat, and lon values is what is causing the runtime to increase by so much. It is the actual indexing that is taking so much time the dictionary key access, i.e. [flag_name], happens in a fraction of a second.

The merger was originally going to use the structure created above to create another structure that mapped the variable name to a flag value which was mapped to a list of values for that flag. Creating this structure however exploded the program's execution time taking the average run time from ~1 second to ~50 second. It was discovered that this was being caused by two arrays being indexed on each iteration of the loop. The files being tested had around 27 variables with 4 files that contained all 1440 minutes of data. This means the indexing was being done at most 2*(27*4*1440) = 311,040 times and it was taking nearly 1 minute to do the indexing. By moving the indexing to happen as few times as possible this brought the execution time down to ~30 seconds. Removing the indexing all together brought the runtime down to less than 1 second.

The merger_dataset class was created to help get around this indexing limitation. When the class merges the merger_datasets, it goes to great lengths to avoid indexing whenever possible. This is done by using sets to get intersections and differences between sets to make sure only values that must be inspected are indexed and even when they are indexed, they are index with a dictionary key as opposed to an array index. This has a significant impact on the execution time where the runtime of the entire completed program is only between roughly 1-3 seconds.

5 Testing

5.1 Plan

To test the program the following strategy was used. Some files where the old merger program had error in the merge process are used to make sure those bugs are no longer present. We wish to test the following cases:

1. A ship that has platform speed over water (PL_SOW) as a variable is reporting the values correctly.
2. Merging a single version 100 file.
3. Merging files with different number of parameters.
    a. A file with a single minute of data merged with a file with multiple minutes.
    b. Multiple 1-minute files being merged with multiple minute files.
    c. File that has non-overlapping minutes.
    d. File with some of the minutes overlapping.

5.2 Results

This program was tested using real data from the SAMOS project. Where for each of these files tested, they have some single minute files being merged with files that contain multiple minutes of data. These files also contain a different number of variables between the files being merged. They also contain the PL_SOW variable. Some of the files being merged have overlapping minutes and some minutes do not have overlapping minutes.

The merged file is compared against the old merger programs results. For the first test, there are six version 100 files the first three files and the fifth file contains a single minute of data and all these files contain the same minute of data. The fourth and sixth files contain 1439 minutes of data that start at the minute after files 1, 2, 3, and 5. When merging this file the old program incorrectly merged the PL_SOW2 value by setting each data value as special when they were not special values. The old program also changed two flags from B to Z that should have been B flags. The new program does not re-produce these errors and it correctly merges the files.

The next test was done on another six files for a different day. The first and fifth file contains a single minute of data. Files 2, 3 and 4 have the whole day's data, 1440 minutes. The sixth file contains 1439 minutes of data excluding the minute from files 1 and 5. These files also have a different number of variables between the files and some of the files contain PL_SOW. The old program produces errors for PL_SOW2 assigning special values where data values are present. It also incorrectly changes a few 'E' flags to 'Z' flags. The new program correctly merges this case and it does not reproduce any of the old program's bugs.

The next test once again uses six files. The first and fifth files contains the same single minute of data. The minute in the second and sixth files start at the minute after the one in file 1 and has 1424 minutes in the files. The third and fourth files contains 1425 minutes of data starting at the same minute as the first file. Some files have PL_SOW and the files have a different number of variables between them. This is the case the old program struggled the most with. Once again it incorrectly assigns special values to PL_SOW2. It also incorrectly changes 326 'B' flags to 'Z' flags. It also changes some 'F' and 'E' flags to 'Z'. The new program correctly merges this case and does not re-produce any of the bugs from the old code.

The next test contained four files to be merged. All the files contained 728 minutes of data and start at the same minute. The PL_SOW variable is present in some of the files and the files have a different number of variables between them. The old code changes 16 PL_SOW2 values to special that in fact have data. It also incorrectly changes 39 'E' flags to 'Z' flags. The new program handles this case correctly and does not re-produce the old code's bugs.

The last test involves merging a single file by itself. The new program successfully handles the merging of a single file. The file that is produced is a copy of the first with the new metadata values added, a new history string added, and the name updated to reflect it is now a version 200 file. Also, if the single file being merged had any minutes that are non-Z flagged or it has a time, latitude or longitude value that is any fill value. Those minutes are then excluded from the new file.

The average execution time was determined by running a single merge for a day that had six files to be merged. The time it took to merge those files can be seen in Table 3. These runs were all done with the same files and the test was run on the COAPS servers so the times greater than 1.5 seconds could be caused by the current load on the server.

Run	Runtime (seconds)
1	1.685
2	1.505
3	2.711
4	1.519
5	1.515
6	2.523
7	1.529
8	1.516
9	1.511
10	1.516
Average	1.753

Table 3. The runtime for the new merger program.

Unfortunately, I cannot get an exact average for the old program, but it is assumed the old program ran in around 1-3 seconds. So, the new program runs in about the same amount of time as the old program, which was written in Perl.

6 Future Work & Summary

There are currently plans to expand the code to allow it to merge netCDF file that have a higher version than 100. This would allow for research quality files (version 300) or intermediate (version 200) files to be merged with each other or the version 100 files. To accommodate this the program will need to handle new research level quality control flags. Currently when the program encounters an error it logs that error to a log file. In the future the ability for the error to be emailed to the program operator will be added. The program will undergo further acceptance testing before becoming an operational part of the SAMOS data processing system.

In summation the new merge program accomplishes what it set out to do. It mergers together any number of version 100 netCDF files into a single version 200 neCDF file. It does this within roughly the same amount of time the old program took. However, it does not reproduce any of the bugs the old program had.

References

daSilva, A.M., Young, C.C., & Levitus, S. (1994) Atlas of Surface Marine Data, Vol. 1: Algorithms and Procedures. NOAA Atlas NESDIS 6, U. S. Department of Commerce, NOAA, 83pp. Avail-able from: NOAA/NODC, Customer Service, E/OC, 1315 East-West Highway, Silver Spring, MD 20910.

Smith, S. R, Briggs, K., Bourassa, M. A., Elya, J., & Paver, C. R. (2018) Shipboard Automated Meteorological and Oceanographic System Data Archive: 2005-2017. *Geoscience Data Journal*, 73-86pp.

Appendix A: Example netCDF File

There are two things to know about reading the netCDF data. First notice that all the variable data is tied to each minute in the time array. Also, the file records the flags for each variable in a somewhat odd manner. Because the number of flag strings is determined by the time dimension the number of flag strings is always equal to the number of minutes in a file. A flag string represents each variable's flag in the order they appear in the file for a given minute. Such that the first flag in the string represents the time variable since it is the first variable in the file.

Below is an example of a netCDF file that contains a single minute of data. Note the variable metadata has been cut because all the variables metadata was taking up around 12 pages. Also the a good number of "", was deleted for history because it there was an entire page of "",. Note that the history array files blank spots in the array with "" if there is no string for that index.

netcdf WTDF_20190510v10001 {

dimensions:

time = UNLIMITED ; // (1 currently)

f_string = 25 ;

h_string = 236 ;

h_num = 50 ;

variables:

int time(time) ;

time:long_name = "time" ;

time:units = "minutes since 1-1-1980 00:00 UTC" ;

time:original_units = "hhmmss UTC" ;

time:data_interval = 60 ;

time:observation_type = "measured" ;

time:actual_range = 20698560, 20698560 ;

time:qcindex = 1 ;

float lat(time) ;

lat:actual_range = 43.39f, 43.39f ;

lat:long_name = "latitude" ;

lat:units = "degrees (+N)" ;

lat:original_units = "degrees (+N)" ;

lat:instrument = "Leica MX420" ;

lat:observation_type = "measured" ;

lat:average_method = "average" ;

lat:average_center = "unknown" ;

lat:average_length = 60s ;

lat:sampling_rate = 1.f ;

lat:data_precision = 0.01f ;

lat:qcindex = 2 ;

float lon(time) ;

lon:actual_range = 290.14f, 290.14f ;

lon:long_name = "longitude" ;

lon:units = "degrees (+E)" ;

lon:original_units = "degrees (-W/+E)" ;

lon:instrument = "Leica MX420" ;

lon:observation_type = "measured" ;

lon:average_method = "average" ;

lon:average_center = "unknown" ;

lon:average_length = 60s ;

lon:sampling_rate = 1.f ;

lon:data_precision = 0.01f ;

lon:qcindex = 3 ;

…

…

int date(time) ;

date:long_name = "calender date" ;

date:units = "YYYYMMDD UTC" ;

date:observation_type = "measured" ;

date:actual_range = 20190510, 20190510 ;

date:qcindex = 1 ;

int time_of_day(time) ;

time_of_day:long_name = "time of day" ;

time_of_day:units = "hhmmss UTC" ;

time_of_day:observation_type = "measured" ;

time_of_day:actual_range = 0, 0 ;

time_of_day:qcindex = 1 ;

char flag(time, f_string) ;

flag:long_name = "quality control flags" ;

flag:A = "Units added" ;

flag:B = "Data out of range" ;

flag:C = "Non-sequential time" ;

flag:D = "Failed T>=Tw>=Td" ;

flag:E = "True wind error" ;

flag:F = "Velocity unrealistic" ;

flag:G = "Value > 4 s. d. from climatology" ;

flag:H = "Discontinuity" ;

flag:I = "Interesting feature" ;

flag:J = "Erroneous" ;

flag:K = "Suspect - visual" ;

flag:L = "Ocean platform over land" ;

flag:M = "Instrument malfunction" ;

flag:N = "In Port" ;

flag:O = "Multiple original units" ;

flag:P = "Movement uncertain" ;

flag:Q = "Pre-flagged as suspect" ;

flag:R = "Interpolated data" ;

flag:S = "Spike - visual" ;

flag:T = "Time duplicate" ;

flag:U = "Suspect - statistial" ;

flag:V = "Spike - statistical" ;

flag:X = "Step - statistical" ;

flag:Y = "Suspect between X-flags" ;

flag:Z = "Good data" ;

char history(h_num, h_string) ;

history:long_name = "file history information" ;

// **global attributes:**

:title = "HENRY B. BIGELOW Meteorological Data" ;

:site = "HENRY B. BIGELOW" ;

:elev = 0s ;

:ID = "WTDF" ;

:IMO = "009349057" ;

:platform = "unknown at this time" ;

:platform_version = "unknown at this time" ;

:facility = "NOAA Marine Operation Center- Atlantic" ;

:data_provider = "Operation Officer" ;

:contact_info = "Center for Ocean-Atmospheric Prediction Studies, The Florida State University, Tallahassee, FL, 32306-2840, USA" ;

:contact_email = "samos@coaps.fsu.edu" ;

:fsu_version = "100" ;

:receipt_order = "01" ;

:start_date_time = "2019/05/10 -- 00:00 UTC" ;

:end_date_time = "2019/05/10 -- 00:00 UTC" ;

:EXPOCODE = "EXPOCODE undefined for now" ;

:Cruise_id = "Cruise_id undefined for now" ;

:Data_modification_date = "10- 5-2019 0:11: 1 UTC" ;

:Metadata_modification_date = " 9- 5-2019 20:11: 2 EDT" ;

data:

time = 20698560 ;

lat = 43.39 ;

lon = 290.14 ;

PL_HD = 318.12 ;

PL_CRS = 319.23 ;

DIR2 = 180.78 ;

DIR3 = 175.68 ;

PL_WDIR2 = 296.26 ;

PL_WDIR3 = 283.82 ;

PL_SPD = 5.668688 ;

PL_SOW = 6.131648 ;

PL_SOW2 = -0.056584 ;

SPD2 = 5.360048 ;

SPD3 = 6.039056 ;

PL_WSPD2 = 3.940304 ;

PL_WSPD3 = 3.688248 ;

P = 1027.48 ;

T = 7.82 ;

RH = 71.9 ;

TS = 8.23 ;

TS2 = 8.52 ;

SSPS = 31.87 ;

CNDC = 3.37 ;

RAD_SW = 2178.46 ;

RAD_LW = 13070.76 ;

date = 20190510 ;

```
 time_of_day = 0 ;
 flag =
  "ZZZZZZZZZZZZZZZZZZZZZZZBB" ;
 history =
  "NETCDF file created Friday May 10, 2019 00:11:00 U",
  "Prescreened on 10- 5-2019  0:11: 1 UTC with code version samospre_v018",
  "# flag changes by pre:    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    1    1",
  "",
  "",
  "",
  "",
...
...
  "" ;
}
```