Berkeley **H**igh **R**esolution (**BEHR**) Retrieval: Readme

Josh Laughner

October 17, 2017

Summary:

The Berkeley High Resolution Retrieval is a high resolution NO_2 retrieval based on the NASA OMNO2 product from the Ozone Monitoring Instrument (OMI) aboard the Aura satellite. This retrieval improves the standard OMNO2 product in several ways:

1. A higher resolution terrain product (GLOBE Terrain Database) is used to calculate the terrain pressure of the pixels
2. A more frequent and finer resolution albedo is used, taken from the Moderate Resolution Imaging Spectrometer (MODIS) instruments aboard the Terra and Aqua satellites.
3. A MODIS cloud fraction is available to use in rejecting cloudy pixels, as the OMNO2 cloud fraction tends to overestimate cloud fractions if highly reflective surfaces are present

This document will describe the current state of BEHR, including its file structure and a changelog. As of this writing, the code is maintained in a Git repository on my machine. I will try to keep the main matter of this document up-to-date; however, if there is a discrepancy between the changelog and the main matter, defer to the changelog.

Contents

1 Authors

BEHR was initiated by Ashley Russell, who completed her Ph. D. in 2012. She showed that using high-resolution albedo and terrain data improved the OMI NASA Standard Product retrieval (see her papers in the Literature section below). Check the group website for her current contact information.

Luke Valin also completed his Ph. D. in 2012; he helped Ashley run the WRF-Chem simulations needed to get the high-resolution NO_2 profiles.

Josh Laughner (joshlaugh5@gmail.com) took over development in 2013. He upgraded the surface reflectance from the MODIS black sky albedo to the MODIS BRDF product and instituted daily *a priori* profiles for as many years as possible. The monthly profiles were also recomputed including lightning NO_2 and a more advanced chemical mechanism.

If you contribute to the development of BEHR, add your name and contribution to this list and update the change log. The **bold** name is the most recent contact if you need help getting things up and running.

2 Literature

The following are important papers in the history of BEHR:

Other literature will be cited in the relevant section; see the full references list beginning on pg. **??**.

3 Getting started

3.1 Before you begin

You'll want to make sure you have the following software or utilities installed:

- **All platforms:**
  - MATLAB
  - Git version control system
  - Python (optional but recommended)

- **Windows:**
  - An SSH client like puTTy
  - Cygwin - a bash shell for Windows. Includes capability to make SSH connections.
  - wget - a method for retrieving remote files, install by selecting it as a package during Cygwin setup
- **Mac:**
  - Find the "Terminal" app, this is where you can SSH from and use commands like `wget`
- **Other:**
  - Get an account on the computing cluster. Currently this is the Savio cluster (http://research-it.berkeley.edu/services/high-performance-computing/). As of 31 Jul 2017, this requires filling out the "Additional User Account Request/Modification Form" at http://research-it.berkeley.edu/services/high-performance-computing/getting-account. Ron will need to approve it.
  - Get access to the group file servers at 128.32.208.13 and cohenwrfnas.dyn.berkeley.edu. Ask other group members for the credentials, or reach out to the contact listed in §1.
  - Get access to the satellite download computer at 128.32.208.11. You'll likely need to reach out to the contact person in §1 to get access, as it is only accessible by SSH using an SSH key (no password) and not something that the group at large has access to.

MATLAB is used to run BEHR itself. Python is another language that the PSM gridding code and some utilities for BEHR are written in currently. Bash is a shell—basically a text based OS interface—that Unix based systems use. It's the easiest way to download large amounts of satellite data with the command line utility `wget`, and is used in much of the BEHR automation and is necessary to work on the cluster. SSH is a secure protocol that allows remote, command-line access to machines set up to accept SSH connection (this includes the file server, Savio cluster, and satellite download computer).

3.2 Quickstart

1. Clone the `BEHR_GitRepo`, `MiscUtils`, and `BEHR_MatlabClasses_GitRepo`, and `Matlab-Python Interface` repositories (Table 1, pg. 10) and add them to your Matlab path.
2. In the PSM directory, build the `omi` package. In the `omi` subdirectory containing the `setup.py` file, run the command `python setup.py install --user`.
3. Mount the `share-sat` share on the Synology NAS at 128.32.208.13 (you'll need the login credentials from another group member).

4. Run `BEHR_initial_setup.m` under in the root directory of the BEHR repo to create the file `behr_paths.m` that will point to various directories on your computer and the file server that BEHR needs. Check that all the paths listed in `behr_paths.m` are correct. You can validate them by running `behr_paths.ValidatePaths()` in Matlab, it should return `1` (`0` means at least one path is invalid).

5. Try running `read_omno2_v_aug2012.m` for one or two days. Make sure it saves to some temporary directory and *not* the main `OMI_SP` directory on the file server.

6. Do the same for `BEHR_main.m`, also saving to a temporary folder.

7. Do the same for `BEHR_publishing_v2.m`, also saving to a temporary folder.

4 Retrieving NASA data

4.1 Group server file locations

A shared file server has been dedicated to storing satellite data locally. It is located at the IP address 128.32.208.13. To connect to this server, your computer will need to have a UC Berkeley IP address. Connecting through an ethernet wall port is the recommended method, but you should be able to access it by connecting to the AirBears2 wifi network or from anywhere as long as you are connected to the Berkeley VPN.

If you are working with satellite data, you will want to "mount" the file server as a network share on your computer; this will let your computer treat the files as being on an external hard drive connected to it, meaning you will not need to copy the files to your computer in order to access them. Instructions to connect are included in the PDF on the shared group Google drive folder, or by clicking on this link

BEHR requires 4 satellite products:

1. OMNO2 level 2 (the NASA OMI NO_2 product)

    - Stored in `/volume1/share-sat/OMI/OMNO2` on the file server.
    - Organized by year, then month.
    - The year links in this directory always point to the most recent NASA product version.

2. OMPIXCOR (a NASA pixel corner product)

    - Stored in `/volume1/share-sat/SAT/OMI/OMPIXCOR`
    - Organized in the same manner as OMNO2

3. Aqua-MODIS cloud product (MYD06)

    - Stored in `/volume1/share-sat/SAT/MODIS/MYD06_L2`
    - Organized by year
    - Have not been retaining old versions

4. Combined MODIS BRDF product (MCD43C1).
    - Stored in `/volume1/share-sat/SAT/MODIS/MCD43C1`
    - Organized by year

4.2 NASA versioning system

First, there are a number of "levels" of data available for the OMI output:

- **Level 0** is the raw instrument data; it has not been calibrated or geolocated.
- **Level 1b** are the "calibrated Earthshine spectra." I believe this means that the reflected/backscattered light from Earth has been both geolocated and calibrated to the solar irradiance spectrum (**??**). Additionally, a single pixel actually consists of four exposures across different rows; these are averaged together at this step (**?**). No I do not know where level 1a went.
- **Level 2** data is where we start. It converts the raw spectra into slant and vertical column densities, and handles stratospheric subtraction. Data is kept at the native pixel resolution; where the pixels are the result of averaging the four exposures mentioned previously (**???**).
- **Level 2G** can be thought of as binning the Level 2 native pixels to a consistent $0.25° \times 0.25°$ grid. Any pixel whose center falls within the grid cell is binned to it. Up to 15 pixels per day will be stored; the data is kept separate for each pixel (**?**).
- **Level 3** is the product resulting from averaging the Level 2G data. This provides a single daily average field (**?**).

These levels just represent the amount of processing that has been done to the satellite data, and should be consistent across all versions.

The versioning describes how the algorithm has changed in time; there are two components to the NASA versioning system for the level 2 product: "collection number" and "data product version number."

The collection number refers to the version of the data processing used to get from Level 0 to Level 1b. As of 23 Nov 2016, the NASA product uses collection 3. This is the number contained in the filename (i.e. `v003`) and in the metadata.

The data product version number refers to the version of the algorithm that produces the Level 2 product. This number does not seem to be stored in the filename or metadata currently. The current version is also 3.0.

Sources for the various data product versions:

- **?** describes version 1 of the level 2 algorithm.
- **?** describes version 2.1 of the level 2 algorithm.

- **?** describes one of the changes between version 2.1 and version 3; namely the revised procedure for fitting the Earthshine radiances to obtain slant column densities. **?** covers the version 3 algorithm itself.

- See **?** as well for information on changes to the chemical transport model used to generate the NO_2 a priori profiles in version 3.

4.3 Downloading from web sites

- **OMNO2:** You will need an EarthData account to download OMI data. As of 31 Jul 2017, you can register at https://urs.earthdata.nasa.gov/.

  NASA OMNO2 data (the NASA OMI NO_2 product) can be downloaded at http://mirador.gsfc.nasa.gov. Search for the keyword "omno2" and specify your date range. We usually leave the location empty. This will return several products; BEHR uses the "1 orbit L2 swath" version. Click on "View files," then on the following page "Add all files in all pages to cart." We don't need any special options, so choose "Continue to cart" and then on the next page "Checkout." On the next page choose "URL List (Data)," this will open in a new tab or window. Save these URLs in a plain text file in the `download_staging` folder under OMNO2 on the file server.

  To download, navigate to the `download_staging` folder in the appropriate `version_x_x_x` folder under the OMNO2 path. and run `wget -i "<list file>"` where `<list file>` is the file you just saved. Keep the window open until this finishes. To sort these into the proper folders, run `sortscript.sh` by typing `./sortscript.sh`. All the OMNO2 files will be moved into the appropriate year/month folders.

- **MODIS data:** Go to https://search.earthdata.nasa.gov and search for the MODIS product abbreviation (MYD06_L2 or MCD43C1). This should bring up a list of products underneath the map. Click on one to access it (for MYD06_L2, we use the regular one, not the near real time one). You can further subset to time period and geographic location using the controls next to the search bar. For MYD06_L2, geographic subsetting to the US really helps cut down on the number of files to download.

  Next, click "Download Data" on the right side under the map. When the next menu comes up, choose "Stage for Download," then click "Submit." You will receive two emails at the email address you provided when you registered for EarthData, the first is confirming that the request was send to the data center that actually serves the MODIS data, the second will be an email from v2lads with instructions to download your order. You can either follow those instructions, or use wget from the command line (Terminal on Macs, cygwin on Windows):

  ```
  wget --ftp-password=<password> -r ftp://<ftp address>/<orders directory>
  ```

  where `<password>` is the password given in the email (usually your email address), `<ftp address>` is the address given in the first line of the FTP commands in the email, and `<orders directory>` is the directory they tell you to `cd` to for example,

  ```
  wget --ftp-password=myemail@email.com -r ftp://ladsweb.nascom.nasa.gov/orders/501162485
  ```

This will download the data to a `ladsweb.nascom.nasa.gov` subdirectory in the current directory. (You can change that with the `--cut-dirs` and `-nH` options.)

If you choose "Direct Download" instead, you can open a page containing the links (which you'd then have to click on) or generate a download script. I have not had luck getting the download script to work (access denied errors).

4.4 Automated downloading

The necessary satellite data for BEHR is being downloaded automatically by the black Compaq computer in room B47. Each week, this computer queries the relevant archives for new files, compares the remote and local file lists, and retrieves any new files from the remote archive. It will not intelligently acquire new versions, so it will need updated accordingly as new product versions are released, or additional satellite products are required by BEHR.

It is also running `read_main.m`, `BEHR_main.m`, and `BEHR_publishing_v2.m` each week to produce new BEHR files. This is keeping the website (§**??**) as up-to-date as possible.

Here the downloading process itself will be described. All scripts can be found in the `Downloading` folder of the BEHR git repository (see §5). For information on how the computer and its OS were configured for this, see Appendix **??**.

- **OMNO2 and OMPIXCOR:**

  Automatic OMNO2 and OMPIXCOR downloads required a data subscription with GISC Mirador (the current one was setup by Josh Laughner in his name). The process consists of two steps:

  1. `order_omi.sh` - this script uses the subscription to request OMNO2 or OMPIXCOR files from the last 120 days. It then waits for the Delivery Notice to be send by Mirador to the *nasa* user set up on this computer specifically to receive these notices by SFTP (see Appendix **??**), then copies these to one folder as a record and to the `download_staging` folder on the file server.
  2. `get_omi.sh` - this script waits for `order_omni.sh` to copy the delivery notice to the appropriate `download_staging` folder, then downloads each file (including metadata) that does not exist locally (using wget). As it downloads each file, it sorts it into the proper year/month folder.

- **MODIS:**

  Currently, albedo and cloud files are handled differently; this may change. `get_modis.sh` searches for both sets of files for the last 90 days. In both cases, it looks for remote files not present locally. For the albedo, it does so by using a feature of wget which can retrieve remote directory listings. It checks the listings on the LADSWEB FTP server against local files, and then retrieves those files not present locally (also using wget).

  Cloud files were more involved, because these become very large as many of them are downloaded. Since we look at individual MODIS granules for clouds (instead of the global gridded product for albedo) we want to restrict the cloud files to the lat/lon

boundaries of the US (or other region of interest), plus a 5-10° buffer to ensure all granules are retrieved.

To do this, we use the *Simple Object Access Protocol* through the Python module SOAPpy to send a request for all MYD06_L2 files in the given time and space range, using the Python script `automodis.py`. This places a list of URLs into a file that `get_modis.sh` can then check against local files.

- **Reading the files:**

  Finally, `run_read_main.sh` will find the last OMI_SP_yyyymmdd.mat file produced, the last albedo file retrieved, and run the MATLAB function necessary to import all satellite data between those two dates into a format MATLAB can work with. `run_behr_main.sh` runs the main BEHR algorithm, and `run_publishing.m` converts the .mat files into .hdf and .txt files and places them in the appropriate folder for the website to access them.

5 Version Control

5.1 Repositories

The core code for BEHR is contained in a Git repository on our group's Synology DS1813+ NAS file server. If you are not familiar with Git, I recommend reading chapters 1-4 at http://git-scm.com/doc, which includes information on installing Git on your system as well as the basic commands. If you are working on a Windows machine you make also want to look at http://guides.beanstalkapp.com/version-control/git-on-windows.html and follow their recommendations, although as I've never used Git on a Windows machine, I can't say for certain how well that works. I have also created video tutorials available by clicking on the links here:

- Command line
- Mac, using the SourceTree GUI
- Windows, using the SourceTree GUI

Further instructions will proceed assuming you're using the command line, since that is the same on any machine.

Once you have Git installed and working on your machine, navigate to the folder that will be your working directory for the project in either Terminal (Mac) or Command Prompt (Windows) and run the command:

```
git clone ssh://RCCohenLab@128.32.208.13/volume1/share-sat/SAT/BEHR/BEHRGitRepo.git
```

This will mirror the repository as a new folder in that directory. A second repository, at `128.32.208.13/volume1/share-sat/SAT/BEHR/MiscUtils.git` contains some general utility Matlab scripts that I have found useful. Some were downloaded from the Matlab

file exchange, many were functions I found myself needing rather often. A third repository is `128.32.208.13/volume1/share-sat/SAT/BEHR/AircraftProfiles.git`, with functions to work with aircraft data sets. Additional repositories will be added to Table 1. I recommend that these be downloaded to folders called "BEHR", "Utils", and "NO2 Profiles" within your main Matlab directory (which you can check with `userpath` at the Matlab command prompt. This way, any internal links that I've set up to be robust as `fullfile(userpath,x,y,z,...,file)` should work smoothly on either a Windows or Mac platform.

Repo location	Rec. folder	Contains
`BEHR_GitRepo.git`	BEHR	All the code needed to run BEHR
`MiscUtils.git`	Utils	Miscellaneous utilities not needed for BEHR but generally helpful
`AircraftProfiles.git`	NO2 Profiles	Functions to work with aircraft data sets, including code to validate satellite data against such data sets.
`BEHR_MatlabClasses_GitRepo.git`	Classes	Certain MATLAB classes I wrote to help manage certain tasks (e.g. error messages)
`https://github.com/firsttempora/MatlabPythonInterface.git`	Python Interface	Matlab functions I wrote that handle converting Matlab types into Python types and vice versa.

Table 1: Summary of the IP addresses, recommended folders within the main Matlab directory, and contents of the three Git repositories.

If you have not worked with Git (or another version control system) before, it may seem like an overly complicated way to go about keeping track of files. It's not. It's far more complicated to try to do it manually. The advantages of a version control system are:

1. *It keeps everything in one place, and makes it easy to keep everything up to date.* As long as you make changes in the directory that the Git repo consists of (and commit the changes periodically), those changes are tracked.

2. *You can see how the code has changed over time.* So, you can roll back to an earlier version if something breaks, or see what code you (or I) changed.

3. *It provides traceablility.* Each of the major steps of BEHR stores a hash that points to the Git commit that was the HEAD when that step was executed. That way if a user has any concerns over a bug in the code, we can trace back to the approximate state the code was in when it produced the data they are looking at.

4. *Parallel development.* If multiple people are developing BEHR at one point, each person can create their own branch and develop simultaneously, while still being able to update the project on the server, and eventually merge the development lines together.

5. *Code sharing.* Conversely, if two (or more) people are both using BEHR, this makes it easier to synchronize code when and if you want.

5.2 Best practices for version control

Read this section before you start changing things! As I'm writing this, the version control for BEHR is fairly straightforward as I'm the only user on the project. If multiple users are developing the code simultaneously, adhering to some good practices will help keep everyone sane.

- **The master branch should only contain stable code.** Generally in Git, the "master" branch is just as it sounds: the main branch with reliable code. This branch should contain the version of BEHR that produces the *published website* data.

- **Changes to the master branch should be understood by all parties.** Before merging any work into the master branch, everyone involved in developing BEHR should sit down together and be sure they understand the changes to be merged. Ideally, this would include a full code review where everyone satisfies themselves that there are no lingering bugs; but that may not be doable within everyone's available time.

- **Each student on the project should create their own development branch.** This will serve as a sort of "personal master branch," i.e. the branch that all the rest of your branches split from and that you merge back into. Ideally, only this branch should then be merged into the true master branch. Conversely, whenever the master branch gets updated, be sure to merge the master branch into this branch.

- **Before merging into the master branch, merge the master branch into your main dev branch.** This should result in a fast-forward merge, if you've been keeping up with updates to the master branch. If not, you'll need to resolve any conflicts and test the updated algorithm to verify everything works.

- **Prepend remote branches with your name.** For example, if I had a local branch `lnox-fix` and you push it to a remote, I should make the remote branch's name `josh-lnox-fix`. This will require specifying both the local and remote names in the push, e.g. `git push -u origin lnox-fix:josh-lnox-fix`. This will help keep straight whose branch is whose.

- **Not every local branch needs a remote branch.** Git is really useful because you can create these branches to develop new code while leaving a stable state alone. However, not every local branch needs to have a corresponding remote branch. To keep the remote repo from getting too crazy, only make remotes when you need to sync that branch across multiple computers.

- **Clean up remote branches when finished with them.** Once you merge or delete a development branch, make sure the remote that went along with it gets deleted.

- **Finally, don't check out someone else's dev branch without telling them.** And definitely don't push back to it, as then they have unexpected code coming into their dev branch.

Along with having good practices for version control, be sure to update the file `Changelog.txt` under the Documentation folder of the BEHR repository. This should contain itemized, human-readable descriptions of the change for each new version or revision. To help this, you should keep a "Development" section at the top of it where you can record significant changes as you make them. When this merges into the master branch, change the header from "Development" to the new version number. Make sure to post the updated change log on the website (it is uploaded as a regular text file and linked under the "Documentation" section of the "Download BEHR data" page on the website.

5.3 Versioning scheme

The BEHR versioning scheme combines the NASA SP version with an internal version:

X.Y[A,B,C,...](revN), e.g. 2.1Arev1

- X.Y is the NASA Standard product version number. Note that the full NASA version number includes the OMNO2 collection number; we just use the standard product version.

A,B,C,... is a letter, i.e. A, B, C, etc. denoting the major BEHR version based on the NASA SP version. The first BEHR version based on NASA SP version 2.1 would be 2.1A, the second would be 2.1B, and so on.

- Optionally, a BEHR revision number can be added to indicate a minor revision. If this is omitted, it is equivalent to revision 0, i.e. 2.1A and 2.1Arev0 are equivalent versions.

When a new BEHR version changes the published data in a significant way, one that could potentially affect the science done with BEHR, the major BEHR version (the letter) should be incremented. Only minor changes should increment the revision number; this is primarily meant for formatting changes (changing fill value, adding metadata, etc.). The only changes to the data that may be considered revisions are bug fixes that impact a small number of pixels ($< 0.1\%$). If you have any doubt, make it a major version change.

Whenever NASA updates, we reset to version A, so when NASA updated from version 2.1 to version 3 of their code, we changed from v2.1C to v3.0A. Likewise, the revision number resets for each BEHR version.

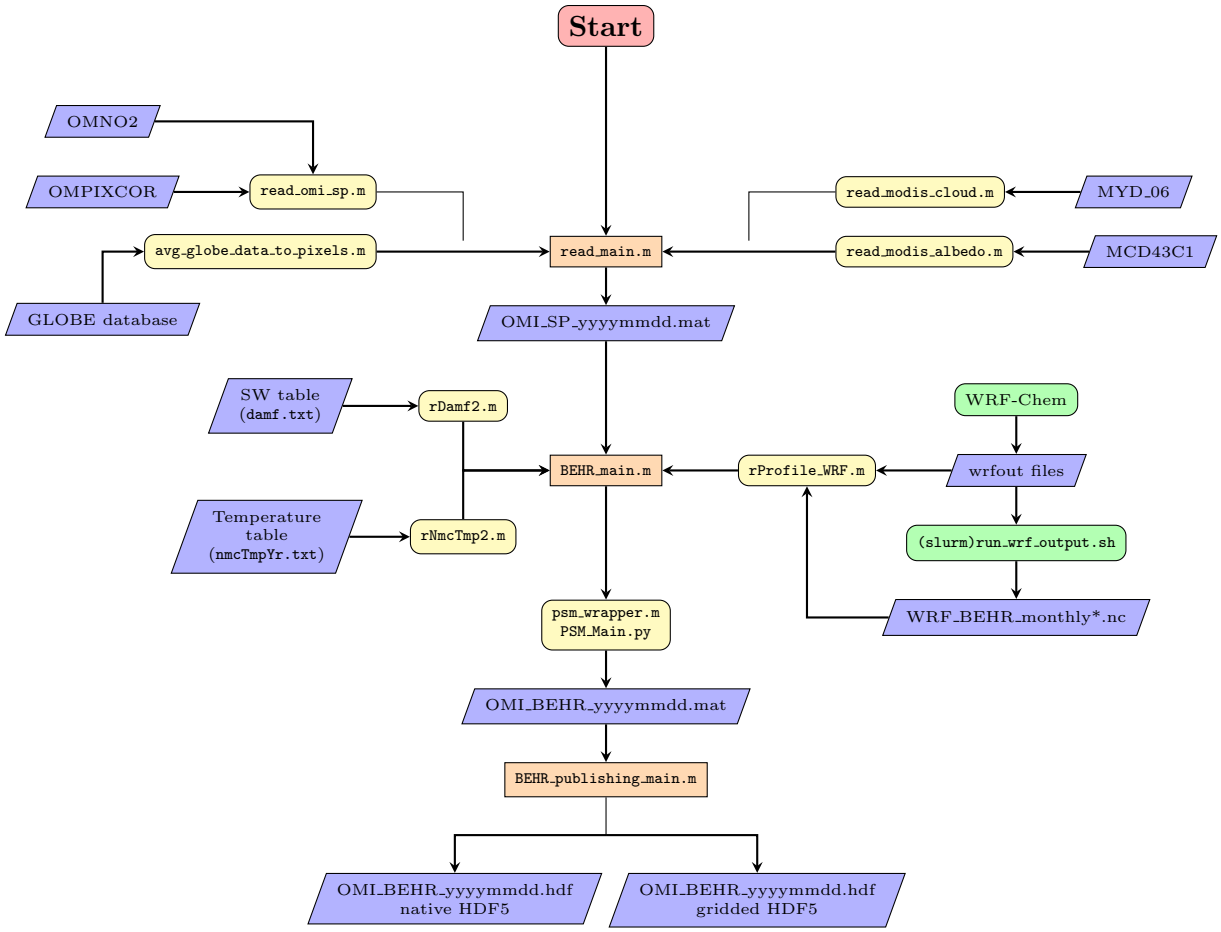

Figure 1: The structure of the BEHR program. Orange rectangles indicate the primary MATLAB functions that need to be executed; yellow rounded rectangles are other functions called internally by those main functions. Green rounded rectangles are external programs not called directly by the BEHR algorithm. Blue trapezoids represent input or output data.

6 Program structure

This section describes the key files of code in BEHR to make it run. Instructions for running it contained within this section are predicated on the idea that you're running BEHR on a desktop computer. For instructions on running it on a cluster, see §**??**.

The general program flow is outlined in Fig. **??**. The first step is reading in the OMNO2 and ancillary data using `read_main.m`. The result of this function is taken by `BEHR_main.m` which computes and applies the BEHR AMF, as well as grids the data to a $0.05° \times 0.05°$ grid. Finally the results can be published to the website using `BEHR_publishing_v2.m`.

6.1 Read data

The first step in running BEHR on new data is to read the NASA files into Matlab. This is done using the `read_omno2_v_aug2012.m` file. It takes a number of parameters as input, but defaults are coded into the file, so that you can (if desired) run it using the "Run" button

in the editor without passing those parameters, and just setting them in the code. Having these options as parameters makes it easier for external driver functions to pass what they need to to run it in a specific mode of operation.

The "`v_aug2012`" part of the name indicates that this file is intended to work with OMNO2 v. 3, which was released in Aug. 2012 (or at least the technical specs for it were).

This function serves several purposes:

1. It reads all relevant variables from OMNO2 files into Matlab.
2. It reads pixel corner data from the OMPIXCOR product and matches it up to the standard product data.
3. It averages the MYD06_L2 cloud product data to each OMI pixel.
4. It calculates the BRDF albedo from the MCD43C1 BRDF parameters for each pixel.
5. It averages the GLOBE terrain data to each OMI pixel, converting from altitude to terrain pressure.

Each of these operations is carried out by a separate read function (Fig. **??**) so `read_main.m` is primarily a driver function that iterates over the necessary data.

For each day that this function processes, a `.mat` file is saved with a single variable, `Data`. This variable is a data structure, in which each OMI swath for that day is stored under a different top-level index (i.e., `Data(1)` refers to the first swath of that day, and `Data(2)` the second, etc.). These data structures contain the data read from the OMNO2, MODIS, and GLOBE files as matrix fields. Each element of the matrix corresponds to an OMI pixel.

Production (i.e. not testing) files output from this script will generally have the name `OMI_SP_vX.YZ_yyyymmdd.mat`. Filenames with additional information are generally testing files I have created in the course of various debugging runs, and should not be used to produce NO_2 data.

I have tried to make this function and its subsidiary function general enough to handle future updates to the input products gracefully, however, some editing may be needed in the event a new product is released. This is also the place to make changes to import more variables from the input products or read additional ancillary products.

6.2 Recalculate AMF and Tropospheric Column

This is handled by the `BEHR_main.m` function in `BEHR/BEHR_Main/`. This is the key component of BEHR. It goes through several steps:

1. It uses the TOMRAD look up table from NASA's OMNO2 product to generate scattering weights (a.k.a. box AMFs) for each pixel, but using MODIS albedo and GLOBE terrain pressure. This is done for the clear and cloudy cases.
2. Reads in NO_2 profiles generated from WRF-Chem and bins them to each OMI pixel.
3. Calculates new AMFs by combining the WRF profiles and scattering weights.

4. The new AMF is then applied to the tropospheric slant column, found by multiplying the NASA AMF with their vertical column.

5. Finally selected data fields are gridded onto a $0.05° \times 0.05°$.

New in version 3 of BEHR, daily NO_2 profiles are read directly from `wrfout` output files from the WRF-Chem model. `rProfile_WRF` can read from any `wrfout` file that has the variables `no2`, `XLONG`, `XLAT`, `P`, and `PB`. It also will read in profiles from `WRF_BEHR` files produces by either `run_wrf_output.sh` or `slurmrun_wrf_output.sh`. These are shell scripts that use command line NCO (netCDF operator) tools to subset and average WRF-Chem output files. Currently, these are only used for the monthly average profiles.

The results of `BEHR_main.m` are BEHR files that contain two variables: `Data` contains the original OMI pixel data, with the new BEHR AMF and NO_2 column density appended. `OMI` contains the variables gridded to $0.05° \times 0.05°$. Both retain the same structure wherein the top-level index represents a single swath. However, in `OMI`, every swath's grid covers the entire region of interest, and cells with no data for that swath have a fill value.

You may wonder that the grid is smaller than the regular OMI pixels; this technique is called *oversampling* and allows us to take advantage of the fact that the OMI pixels do not overlap exactly day-to-day to obtain an effective resolution greater than the pixel size if we average over longer time periods.

This function should require minimal upkeep even in the event a new NASA product is released, but is where major changes to the algorithm will occur.

6.3 Weight pixels and map

This step is used when creating maps of BEHR NO_2 data. The existing function to handle this is `no2_column_map_2014.m`. It can be called with a GUI by using `no2colmap_wrapper.m`. Either way, this uses the gridded BEHR data to do time averaging and `omi_pixel_reject.m` to filter out pixels with cloud contamination or that are affected by the row anomaly.

When doing temporal averaging, each grid cell has an areaweight associated with it; this is the sum of the reciprocal of the areas of each pixel that overlapped that grid cell. By weighting by the reciprocal of the area, it gives more weight to grid cells that got information from small, more representative, pixels. Therefore, any temporal average should be weighted by this field.

Plotting can be accomplished using the `m_map` package (which is used in the above functions, and which should be included in the BEHR Git repo). MATLAB also has built in mapping functions which can be used if desired. Generally a `pcolor` plot of some sort is the best option.

6.4 Unit testing

"Unit testing" is a software engineering technique where small units of code are tested to ensure that they are behaving correctly. We have implemented some unit testing functionality for BEHR that should be run *at least* prior to releasing a new version.

Unit tests for BEHR are stored in the `Production tests` subfolder. There are currently two broad types of tests contained here.

6.4.1 Standard unit tests—`unit_test_driver.m`

`unit_test_driver.m` is the primary function that should be run when a new version of the code is ready. When run, this code will issue interactive prompts that allow you to determine its operation.

1. This code is capable of executing `read_main.m` and `BEHR_main.m` itself for a few specified test days which are chosen to test the code for a number of standard days, plus days which have caused issues in the past. The generated files are placed in a directory `ProductionTests/UnitTestData/ProducedYYYYMMDD` where YYYYMMDD is the year, month, and day that the test is run on. It will also produce a file, `GitReport.txt` which contains information about the state of the code at the time the files were produced.

2. Whether or not you generate the data files, it will next run `reading_priori_tests.m` and `behr_priori_tests.m`. These are not tests of the *a priori* profiles necessarily; just tests that can be run without knowledge of any previous versions of BEHR. For example, one of the things that `reading_priori_tests.m` checks is that the corner coordinates of the pixels make sense, that they contain the pixel center. Generally, as you find bugs in the code *or* if there's something that you're concerned about that comes up as you're coding, see if there's a test that can go here to ensure that the bug does not recur.

3. Finally, it runs `behr_unit_test.m` on the Data structures produced by `read_main.m` and/or `BEHR_main.m`. It checks the structures to determine if the same fields are present and have the same values in both old and new versions. Since you will most likely *expect* some differences between versions, this will probably result in a "failed" unit test. *However*, the important thing is to make sure that only differences that you are expecting occur.

6.4.2 Detailed data tests—`behr_prod_test.m`

At present, `behr_prod_test.m` is a separate utility from `unit_test_driver.m`, though I may integrate it in the future. This function is aimed at giving a statistical summary of the differences in certain fields between an old and new version of BEHR data. It will randomly sample a number of old and new files, look at the fields specified by the variable `fields_to_check`, and generate statistics of the differences between them. It not only checks that differences in values themselves, but the occurrence of fill values.

6.5 Publish

Once a production quality version of BEHR is ready to go, it needs to be published to the website (see §**??**). The data is published in three formats:

- Plain text CSV files that list the important variables for each pixel. The 2D structure of the swath is lost. This is considered at the native OMI pixel resolution.
- HDF5 files containing the same variables as the plain text files. However, because HDF files can store data in arrays with any number of dimensions, the 2D structure of the swath can be preserved. This is also at the native OMI pixel resolution.
- HDF5 files containing a subset of the variables at the $0.05^\circ \times 0.05^\circ$ resolution.

The publishing is handled by the function `BEHR_publishing_v2.m`. It is set up to produce a single one of the above three types with each run. Which type and the dates to produce for can be specified as inputs to the function or by modifying the corresponding values in the "Set Options" part of the code and running it without any inputs. It also retains the ability to produce some specialized versions of the product, mainly those that used *a priori* profiles derived from the DISCOVER-AQ research flights.

7 Code style

I'll apologize now if this seems pompous, but having a consistent coding style makes the algorithm easier for newcomers to read, and easier for us all to understand what's going on at a glance because, i.e. knowing that single, lower case letter variables are always loop indicies makes it easy to identify, at a glance, that, e.g. `no2_profiles(:,a)` is being subset according to some loop variable `a`.

Some of these rules grew up naturally as the algorithm developed, others are the result of bad experiences trying to understand the program flow.

7.1 Functions vs. scripts

Always use functions (or classes), never scripts. Matlab code can be contained in scripts or functions, the latter start with the keyword `function` and the name of the function must match the file name. (You can get a function template from the Matlab editor under the dropdown menu for "New".)

Scripts all share the same workspace, running a script is conceptually identical to copying and pasting its contents in the command window. This has two main problems:

1. variables with the same name in both scripts will conflict
2. since variables are not passed explicitly into and out of scripts, it's impossible to tell from the call to the script what variables in the calling script are needed or modified.

7.2 Comments

1. Every function should start with a comment block that describes the operation, inputs, and outputs of the function. See `state_outlines.m` in the general Utils repository and `BEHR_main.m` itself for ideal examples. These code blocks should consist of:

(a) First line: name of the function in all caps followed by a one-line description of its purpose.

(b) A description of each possible syntax to call the function (excluding parameters, see #??) where the outputs, function name, and function inputs are written in all caps followed by a paragraph describing that syntax.

(c) A list of any parameters, including for each acceptable values, what its effects are, and the default value. (Parameters are inputs for which the order does not matter, but each value must be preceded by a string that names which parameter is being given.)

(d) Optionally, examples.

2. Use comments surrounded by `%` to delineate sections of the code, e.g.

```
%%%%%%%%%%%%%%%%%
% INPUT PARSING %
%%%%%%%%%%%%%%%%%
```

With some blank lines before it, this is easy to see and helps the eye find major sections of the code.

3. Reference any papers (with at least author, journal, DOI or author, journal, year, pages) or websites relevate to the code. This includes e.g. stackexchange or Matlab forum discussions that code is obtained from.

4. Explain any constants or conversion, e.g.:

```
no2 = ncread(wrf_file, 'no2');
% NO2 from WRF is in ppm, we want unscaled mixing ratio
no2 = no2 * 1e-6;
```

There should never be "mystery numbers," when constants show up in code, they should almost always have a comment explaining why they are needed.

5. If you are having trouble working out how to code something, write down your thought process in comments. Having it written down will help you, and likely help whoever comes after you understand why you made the decisions you did.

6. If a change to long standing code is made to fix a bug, comment why the change was necessary.

7. If in doubt, comment.

7.3 Variable and function names

1. Single letter, lower case variables are reserved for loop indicies, e.g. `a`, `i`.

(a) I typically start with `a` for the outermost loop, unless another variable makes more sense, e.g. `d` for a loop over dates.

(b) Longer names may be appropriate when a large number of loops are occurring simultaneously, e.g. `i_sza`, `i_vza`, `i_raa`, `i_alb`, and `i_surfp` if you were looping over all the variables of the AMF scattering weight lookup table. Starting with a single lower case letter followed by an underscore helps denote these are still loop variables.

2. Double lower case variables (e.g. `xx`, `yy`) are to be used for logical indexing or the result of the `find` function, e.g.:

```
xx = lon > lon_lim(1) & lon < lon_lim(2);
```

3. All other numerical, string, and cell array variables should be given pothole case (e.g. `amf_clear`, `wrf_file`: all lower case letters with words separated by underscores) names.

4. Function names should also be written in `pothole_case`.

5. Structures and instances of classes should be written in camel case (e.g. `BehrGrid`: each word in the name is capitalized).

6. Class names should also be written in `CamelCase`

7. Subordinate the capitalization of "BEHR" in names to these rules. Older functions may not follow this rule (and renaming breaks the history in Git) but new functions and variables going forward should.

8. All variables names (other than loop or logical index variables) should be descriptive enough that it is obvious what the significance of that variable is: `grid_lon` is better than `glon`, `wrf_profile_path` is better than `wp`, etc.

7.4 Single, authoritative representation of data

Constant values that may be references multiple times should have a function that returns that value. This way, if that value changes, all instances of it in the algorithm change as well. This is why:

- The version identifier is contained in `BEHR_version`
- The fill value is stored in `behr_fill_val`
- The minimum allowed AMF value is contained in `behr_min_amf_val`
- etc.

7.5 Logical organization through functions and sub-functions

Use functions to help logically organize the code. This has two main flavors:

1. Avoid long, repetitive, copy-paste code by using functions written to take what changes between each copy-paste chunk as inputs.

2. Move large but logically connected sections of code out of large, monolithic files into separate function files, or at least a sub-function within the same file.

Both of these can be seen in the changes to the main reading code between version 2.1C (`read_omno2_v_aug2012`) and v3.0A (`read_main`). In version 2.1C, each variable read in had a copied chunk of HDF reading code. This became very difficult to maintain. In v3.0A, this was replaced with the `read_omi_sp` function. This loses some flexibility, but ensures the same behavior is applied to all data read in.

Likewise, in v2.1C, the blocks of code for reading in MODIS cloud, MODIS albedo, and GLOBE terrain elevation data were all contained directly in the main reading function. By moving these to their own separate function, it is easier to obtain a high-level overview of the main reading function by seeing that it calls, in turn, functions that read in SP data, MODIS cloud data, etc. The details are then contained in each of these functions.

Note: this benefit only happens if the called functions are clearly named, so be sure to give your functions descriptive names!

8 Running BEHR on a compute cluster

8.1 BEHR in Matlab

8.1.1 Parallelization

Starting in Jan 2015, the main BEHR code (`read_omno2_v_aug2012.m` and `BEHR_main.m`) had simple parallelization added to the code body. This is only intended to be used when the code is run on a cluster because in order to run operations in parallel, Matlab must send all relevant variables from the main instance to the parallel "workers" actually executing the code. Given the size of the variables routinely used in BEHR, this communication overhead can result in overall slower work than a serial execution if too few cores are used. As of 29 Jan 2015, I have yet to run any sort of rigorous benchmarking tests, but anecdotally, running BEHR in parallel with only 2 cores seemed to result in a slower execution than running it in serial.

There were numerous small changes to the code to enable parallelization, most importantly, the `for` loop over days was replaced with a `parfor` loop. `parfor` in Matlab allows multiple iterations of a for loop to run in parallel, and automatically handles the distribution of data. Compared to an `spmd` block, we give up control over how and when the data is distributed in exchange for Matlab handling it automatically.

Compared to previous versions of BEHR, most of the changes in the code (too numerous to list individually) were mainly to allow Matlab to "slice" variables appropriately for inclusion in a `parfor` loop. The change that is significant for the user is that a number of global variables were added to control both the action of the parallel loop and the paths to data.

The reason these variables were made global variables instead of inputs to the function was to allow them to be set in a run script once. This makes the run script more "bash like" (programs compiled from a shell like bash—including GEOS-Chem and WRF-Chem—often reference environmental variables to determine how they compile).

The two variables controlling the action of the parallel loop are `onCluster` and `numThreads`. `onCluster` should be set to true in a run script whenever the script is being executed on a cluster. `numThreads` is, by default, set up in the run script template to take its value from an environmental variable, `MATLAB_NUM_THREADS` set in the bash shell that calls the Matlab instance running it. This was done because it is possible to set this environmental variable by referencing another that relates directly to how many CPU cores are available. See the example in §**??** for an example.

The path variables are all defined in the .m files, and are set up such that if `onCluster` is true, the functions will look to global variables setup in the run script for those paths. (If any aren't defined, an error is thrown.) This is so that you only have to edit the run script to use BEHR on a cluster. If `onCluster` is false, the function looks to the paths coded into the functions.

8.1.2 Running it

The code was set up to be executing using a "runscript," which is a Matlab script file that can be called from the command line as:

```
matlab -nosplash -nodisplay -r "run('runscript.m')"
```

The argument `-nosplash` tells Matlab not to show the splash screen on startup, and `-nodisplay` tells Matlab that it shouldn't start the GUI interface. (Since we're working from the command line, we don't need it, and we don't want it to try to open it if we can't see it anyway.) `-r "..."` tells Matlab to execute the command given in quotation marks as soon as it starts up. In this case, that is to execute the runscript in the current directory.

A template for a BEHR runscript can be found in `BEHR/Run_Scripts/`. You'll notice that the runscript template does several things:

1. Sets the global variable `onCluster` to `true`. This lets any scripts that use that variable know that it is running on a cluster and should activate any parallel elements in the code.

2. Sets the variable numThreads. By default this is set to the variable of the environmental variable `$MATLAB_NUM_THREADS` from the shell that executed this instance of Matlab. More on why this is preferable below.

3. Detects any active parallel pools and shuts them down before exiting.

4. Everything is wrapped in a `try-catch` block that will, in the event of an error, prints the error information to the console, closes any active parallel pools, and exits with status code > 0.

This is a very general template, so you can use it to call any function you parallelize using the global variables `onCluster` and `numThreads`. For BEHR, you'll also have to set all the global path variables for `read_omno2_v_aug2012.m` and `BEHR_main` in the runscript.

8.1.3 Submitting to the cluster queue

Like any job you want to run on a computing cluster, you'll need to write a shell script that is put into the queue for the cluster. Since the Savio cluster that I use operates with the SLURM scheduler, this section will be written from the point of view of submitting to SLURM using the bash shell. (I assume that if you use tcsh or another shell that you know what you're doing well enough to make the necessary adjustments.) Below will be an example submit script, each important line will be described afterward.

```
1  #!/bin/bash
2  #
3  # Job Name:
4  #SBATCH --job-name=BEHR
5  #
6  # Partition:
7  #SBATCH --partition=savio
8  #
9  # Account:
10 #SBATCH --account=ac_aiolos
11 #
12 # QoS:
13 #SBATCH --qos=condo_aiolos
14 #
15 # Number of nodes and processors per node
16 #SBATCH --nodes=1
17 #SBATCH --ntasks-per-node=20
18 #
19 # Wall clock limit:
20 #SBATCH --time=24:00:00
21 #
22 ## Run command
23 export MATLAB_NUM_THREADS=$((SLURM_NTASKS_PER_NODE-1))
24 cd /global/home/users/<me>/<behr_run_dir>
25 module load matlab
26 matlab -nosplash -nodisplay -logfile "runlog.txt" -r "run('runscript_behr.m')"
27 MATLAB_EXIT=$?
28 exit $MATLAB_EXIT
```

Line 1 Lines starting with a `#!` are called a *shebang* in bash-speak, this one tells the computer to run the script using the bash shell. Including this is a safety measure; it ensures the script is always run using bash if another shell interpreter is active.

Line 2 A `#` indicates a comment in bash

Line 4 The `#SBATCH` at the beginning of this line tells the SLURM scheduler that a setting is being passed. Bash ignores it because of the `#`, but SLURM does not. In this case, we're setting the name of the job that will appear in the queue.

Line 7 Tells SLURM which partition of nodes to run on. We use "savio" unless we need lots of RAM.

Line 10 The SLURM account that would be charged for running (I think). Ron is under the "aiolos" account; the "ac" indicates that it is the account.

Line 13 "QoS" determines what rules the job is run under and how compute time is charged. The part after the underscore will always be the account, "aiolos." The part before the underscore determines the rules. "condo" means that we can use up to 8 nodes at a time and won't be charged for it.

Line 16 How many nodes to to. A node is the computing unit of the cluster—each node will only run one job at a time.

Line 17 How many cores to use on each node. Each node has two 10-core processors for 20 cores maximum per node.

Line 20 How long to let the job run before forcing it to quit. Here, we set it to 24 hours. It's good practice to do this to prevent a job from running indefinitely.

Line 23 `export` in bash means "set this as an environmental variable" which programs executed from this shell can access. The `$(( ))` in bash means to evaluate an arithmetic expression and return the result. So here we're saving one less than the number of tasks per node in the `MATLAB_NUM_THREADS` variable. I do this to be a little bit careful to leave a core free for the main Matlab instance, although I don't know if that's strictly necessary.

Line 24 Change to our run directory (just in case the job starts somewhere else). `<me>` and `<behr_run_dir>` are just placeholders.

Line 25 Savio organizes applications by modules, here load the matlab module to be able to run matlab.

Line 26 Execute matlab with command line arguments. The only new one is `-logfile "runlog.txt"` which saves all Matlab command window output to the file runlog.txt. One mistake to avoid is including the `-nojvm` flag, because (apparently) the parallel computing toolbox needs Java to work. Don't ask me why.

Line 27 The `$?` is the last given exit code. We save this to a variable...

Line 28 ...and then exit this script with that exit code. This will let you know if the script succeeded or failed.

If this script is named e.g. `matrun` then typing `sbatch matrun` on the cluster will submit it to run. You can check the status of all jobs with `squeue`.

The reason that we use the variable `MATLAB_NUM_THREADS` to pass the number of available cores to the run script is to be a little careful about not causing Matlab to request more cores than we've set aside. By deriving `MATLAB_NUM_THREADS` from the SLURM variable indicating the number of cores to be used per node, we ensure that a change to the SLURM settings is propagated through to our Matlab instance without any intervention on our part.

8.2 Resources

1. Matlab documentation on parfor loops: http://www.mathworks.com/help/distcomp/parallel-for-loops-parfor.html
2. Matlab parfor loops, classification of variables: http://www.mathworks.com/help/distcomp/classification-of-variables-in-parfor-loops.html
3. The High Performance Computer (HPC) User Guide: http://research-it.berkeley.edu/services/high-performance-computing/user-guide

4. SLURM Documentation: https://computing.llnl.gov/linux/slurm/documentation.html

5. List of SLURM parameters (i.e., what can be set in the bash run script on the lines beginning with `#SBATCH`: https://computing.llnl.gov/linux/slurm/sbatch.html

9 Maintaining the website

9.1 Gaining administrative access

To be able to edit the website, you'll need to make an account. At the top right of every page, there should be a register link. Complete that to create an account.

Once you have an account, you'll need to contact an existing administrator to have your account promoted. Existing administrators are (try in order):

- Josh Laughner (joshlaugh5@gmail.com)
- Anna Mebust (annamebust@gmail.com)
- Ashley Russell (ashley.ray.russell@gmail.com)

Go ahead and add yourself to the top of this list once you've been promoted. If none of the existing admins can help, you can also contact Stephen dos Remedios (steven@meetthere.com), he is the one who helped set up the website initially, and can help with most issues, although he usually prefers to have one of the other admins handle adding new admins.

9.2 Editing the content

Log in with your newly promoted admin account. At the top of the site, you should now see the Dot Net Nuke Community bar. To edit, change the dropdown menu at the top right from "View" to "Edit." Each section in each page should now have a "Manage" ghost button when you hover over it. Hovering over that in turn brings up a menu, one option is "Edit content." This is how you modify the text and most of the content in the website.

Some small files should be uploaded directly to the website, things like the change log, user guide, etc. In the edit content window, the planet with a chain link button is the hyperlink manager. This lets you insert links to other websites, but also to uploaded files. The button next to the URL field bring you to the Document Manager, where such files can be uploaded. *This is not how to upload new versions of BEHR, see* §**??***!*

9.3 Providing new data

The website is set up such that a URL pointing to `/behr` (no `http://` or `www`, this is a local path) points to the directory `/volume1/share-sat/SAT/BEHR/WEBSITE/webData/` on the file server at 128.32.208.13. The most important subdirectories are noted in Table **??**.

Website URL	Server folder
/behr/behr_hdf	/volume1/share-sat/SAT/BEHR/WEBSITE/webData/behr_hdf
/behr/behr_txt	/volume1/share-sat/SAT/BEHR/WEBSITE/webData/behr_txt
/behr/behr_regridded_hdf	/volume1/share-sat/SAT/BEHR/WEBSITE/webData/behr_regridded_hdf

Table 3: Important data paths on the website and file server.

The files publicly available on the website are produced by `BEHR_publishing_v2.m` in the `HDFtools` folder in the BEHR repo. Note that starting with version 2.1Arev1, we include the version number in the file names (both for the published data and our internal .mat files) and as an attribute for each swath in the HDF files. This version string is defined in the `BEHR_version.m` function in `Utils/Constants` in the BEHR repo. Make sure to update this version string before reproducing the full data record so that it is stored in all the proper places.

10 Checklist for releasing a new version

- All development work should be done on a development branch of the repo, *not master* (§5.2).
- Once the code is ready, run `unit_test_driver.m` and, if prudent, `behr_prod_test.m` on the development branch (§??). Ensure that it passes the priori tests and that only expected changes fail the old/new comparison.
- If there are others actively working on the BEHR project, set up a code review meeting. Go through the differences between the new code and the current master, as well as the results of the unit tests. This both ensures that all developers are aware of the incoming changes to master (and how it will affect their development/research) and provides additional eyes to check for errors.
- Update `BEHR_version.m` to the new version number (§5.3) and modify the changelog to reflect the changes made.
- Merge changes into master branch.
- Ensure that the prior version's `OMI_BEHR*.mat` files are backed up to Box. (The `BoxMirror` script from https://github.com/CohenBerkeleyLab/BoxUpload makes this much easier.)
- Reproduce the data record with the new version, probably best to use the cluster for this (§??). This should generate new `OMI_SP*.mat` files, `OMI_BEHR*.mat` files, and the published files (`*.hdf`, `*.txt`).

  - Double check in `BEHR_publishing_v2.m` that any new variables have been added to the `vars` variable in the `set_variables` subfunction.
  - Variables that should be included in the gridded products are listed under `gridded_vars` in the `remove_ungridded_variables` subfunction.
  - Add the variables to the attribute table in the `add_attribute` subfunction.

- Move these files to `share-sat` on the `128.32.208.13` file server. The published files (`*.txt`, `*.hdf`) should go in `/volume1/share-sat/SAT/BEHR/WEBSITE/staging` for now. The `*.mat` files should go in staging directories as well, usually these staging directories are inside the folders those `*.mat` files are normally stored.

- Check with `behr_prod_test.m` that the published files have the expected changes. Also manually verify that the entire data record was reproduced (no files missing).

- If everything is ready, replace the old versions' files with the new files. See Table **??** for the paths.

- Upload the new changelog to the website. Update the user guide if necessary, and upload too. Both are uploaded with the file manager on the website (under the Admin button).

- Update references to the version string on "The BEHR Product" and "Download BEHR Data" pages of the website. This includes changing the blurb on the "Download BEHR Data" pages to describe the changes pertaining to the current version. Check that the table of variables on "The BEHR Product" is still accurate.

- Tag this commit in the master branch of the Git repo with the version string.

- Update the satellite download computer to the new version of the code.

There is one additional file on the file server that must be updated. It will be updated automatically the next time `run_publishing.sh` runs on the download computer. That is the `behr_version.txt` file under `/volume1/share-sat/SAT/BEHR/WEBSITE/webData`. It just contains the version string with no newline at the end. This is used in the utility `get_BEHR.sh` for users to download BEHR in batch.

11 Additional utilities

11.1 Verification

11.1.1 Reading

- ?

- ?

- ?

11.1.2 Background

One method of describing the validity of a satellite retrieval is to compare it against *in situ* aircraft measurements. In general, there are two ways of calculating a column from aircraft measurements:

1. Use a profile that extends over most of the troposphere and integrate this directly, making whatever corrections are needed at the top and bottom of the profile.

2. Use measurements taken at the interface of the boundary layer and free troposphere, assuming each are well mixed, and integrate up and down from there.

#?? is much less common, but was used in **?** to validate the original BEHR product. There have been a large number of aircraft campaigns which can be used for validation, available online at http://www-air.larc.nasa.gov.

11.1.3 A caveat

I'm writing this description about a year after I last worked with this code, so I might forget some details. If there's a discrepancy between how the code works and what I say here, don't be terribly surprised.

11.1.4 Code base

This code is not actually contained directly in the BEHR repo, instead it is in the `AircraftProfiles.git` repo under the `satellite_verification_scripts` subfolder. There are several components to this code base, we will take each in turn.

Reading in merge files The data you usually want to download from the LARC website is called a "merge" file, meaning that all the different instruments on the flight have their data merged into a single file. These are text files in a specific format, so we first want to read them into a format Matlab can handle. The function `read_merge_data.m` in the `read_icart_files` subfoler does so, saving a .mat file with a `Merge` structure as well as a data table. The `Merge` structure is used in the remainder of the code.

I've already run this for most of the campaigns, and customized the data a little bit. This is saved (as of 24 May 2016) on the file server at 128.32.208.13 in `/volume2/share2/USERS/LaughnerJ/CampaignMergeMats` (also look in the archive directory if I've already graduated). That in turn is organized by campaign, platform (aircraft ID or ground), and time resolution. Much of the other code expects this organization.

Different campaigns can have slightly different names for important variables, so `merge_field_names.m` in the `utility_scripts` is the central location where these are stored. It will return a structure with the proper field names for each variable, along with the directory of that campaign. If you add a new campaign, you should include it in `merge_field_names.m`.

Spiral/profile verification The most common method of verification is to use full profiles. This has two further subcategories: the DISCOVER campaigns were geared specifically towards satellite validation and so have aircraft spirals up or down over a small area (and more importantly identified in the data with a profile number). Other campaigns do not focus on satellite validation, and so you will need to manually identify what parts of the data constitute a profile.

Manually identifying profiles ranges is done with the `select_campaign_ranges.m` script in the `GUIs` subfolder. This uses the `select_changing_altitude.m` GUI, which presents you with a graph of the aircraft altitude vs. time. You can select periods where the aircraft is consistently climbing or descending and these will be stored in the Ranges structure. This approach was used because it seemed more reliable than trying to have the computer decide, as the aircraft does not necessarily monotonically ascend or descend. Plus it allows you to look for other flight patterns. The Ranges structure should be saved in the main directory for each campaign and added to `merge_field_names.m`.

To actually execute the comparison, use the `Run_Spiral_Verification.m` function in the `satellite_verification_scripts` subfolder. This is a driver script for `spiral_verification_avg_pix2prof.m`, which has so many input options it became easier to enumerate them in an outside script than to try to remember to set them all. Plus this iterates over all days in the campaign and cleans up the output afterwards. (There is, or maybe used to be, another script called `spiral_verification.m` that instead of averaging together all pixels intersected by a profile did the opposite—compared the profile against each pixel individually. I realized that this wasn't as useful, but left the code around for reference. Don't use it though.)

Boundary layer verification This one requires a few more steps. The core of the code is the `Run_BL_Verification_w_Heights.m` script, which is the driver for `boundary_layer_verification_presel_heights.m`. What makes this more complicated is that is requires you to identify the boundary layer height from profiles in the flight for each day. This requires two steps:

1. Find the ranges of time during the flight where the aircraft is consistently ascending or descending using `select_campaign_ranges.m`.

2. Give the Ranges structure (along with other inputs) to `select_BL_heights.m` in the `GUIs` folder. This function will make a best guess at the boundary layer height using `find_bdy_layer_height.m` in the `plotting_scripts` folder, then allow you to examine, modify, accept, or reject it.

You then need to point `Run_BL_Verification_w_Heights.m` to the file where you've saved the output from `select_BL_heights.m`. As in **?**, it will then find parts of the flight in the boundary layer and extrapolate down to the ground to compute the column. Also as in **?**, the three fields you usually want to use to determine potential temperature are $[NO_2]$, $[H_2O]$, and potential temperature.

Other utilities One other goal of the code in this repository was to experimentally verify the conclusion in **?**, that the relative position of aerosol and NO_2 layers controls the impact aerosols have on the NO_2 retrieval. This involved subsetting the profiles in DISCOVER-AQ and SEAC4RS campaigns based on the relative position of the two layers and the extinction of aerosol present, then running `Run_Spiral_Verification.m` for each subset of profiles, and seeing if the correlation was different. That is why `Run_Spiral_Verification.m` accepts profile numbers as inputs.

This ended up not generating any very exciting conclusions, as the uncertainty from other sources seemed to overwhelm the aerosol effect at the AODs observable. The code is still available for future use.

Key functions are:

- `multiple_categorize_profile.m` which handles the categorization of each profile into aerosol above, NO_2 above, or coincident. This is superior to `categorize_aerosol_profile.m`, which it calls several times with different criteria to get the best guess at the proper categorization.
- `Run_all_aer_categories.m` then is a driver script for the driver script (yeah, it's Inception but with code) `Run_Spiral_Verification.m`, which it calls with each subset of profiles.

There are also several functions that take figures from **?** to compare my experiment against her theory.

It was for this project that I obtained the aerosol extinction in the blue wavelength from Lee Thornhill at Langley, which was appended to the Merge structure. Therefore my existing Merge structures do have extra fields not present in the standard merge files from the LARC site.

11.1.5 Summary

1. Read in Merge files if necessary using `read_merge_data.m`
2. (if using a campaign without profile numbers) Identify profiles using `select_campaign_ranges.m`. Save the resultant Ranges structure and add it to the `merge_field_names.m` file.
3. (if doing boundary layer verification) Identify boundary layer heights with `select_BL_heights.m` and save the results.
4. Running, do either:
   (a) Modify `Run_Spiral_Verification.m` with the desired settings and run it, pointing to the proper range file if necessary (note that if done properly, `merge_field_names.m` can ask interactively which range file to use).
   (b) Modify `Run_BL_Verification_w_Heights.m` with the desired settings, pointing to the proper heights file and run.

11.2 EMG fitting

11.2.1 Reading

- ?
- ?
- ?
- ?
- ?

11.2.2 Background

The idea behind EMG (exponetially modified Gaussian) fitting is that by taking satellite NO_2 observations, rotating them so that the wind direction is aligned to the positive x-axis, and integrating across the plume (perpendicular to the wind direction), you get a line density that is a one-dimensional representation of the NO_2 concentrations as you move downwind of the city. By fitting this with an exponentially modified Gaussian function with 5 fitting parameters, one can extract information about emissions and chemical lifetime.

11.2.3 Code base

There are four main files for this analysis: `calc_line_density.m`, `fit_line_density.m`, `calc_fit_param_uncert.m`, and `compute_emg_emis_tau.m`. All are stored in the folder `BEHR/Emissions`, where `BEHR` is the BEHR repository folder; the latter two are further in the `Plotting` subfolder.

As you might expect, `calc_line_density.m` produces the actual line densities and returns them along with a lot of other information that can be useful for calculating uncertainties. It expects that you have satellite data stored in Data structures, as in the BEHR .mat files. It also requires that you pass in a vector of wind directions (and speeds if you want to subset days by wind speed) that is the same length as the number of days to average. This allows you to use this function with whatever wind data you want rather than tying it to a specific way of getting the wind data necessary to rotate the plumes. This function can take a fairly long time to run, so it's well worth running it once (if possible) and saving the output rather than running it every time a line density is required.

`fit_line_density.m` then takes the output of `calc_line_density` and fits the EMG function to it. It only needs two inputs: the x-coordinates and values of the line densities itself, but it has a number of additional options. Some (like the ability to turn off the iterative output of `fmincon`) are useful for switching between running interactively and in batch mode. Others are useful for exploring the uncertainty of the fitting parameters. In general, it is set such that the defaults are those used in **?**.

`calc_fit_param_uncert.m` then takes the 5-element vectors of the fit parameters and their fitting uncertainties, along with some other optional inputs, and computes the total, absolute uncertainty in each parameter. This is based of the uncertainty calculations in **?**

and **?**, but I have adjusted the assumptions slightly, as detailed in the comments in that function.

`compute_emg_emis_tau.m` takes the values of a, x_0, their uncertainties, and the subset of wind speeds considered to compute the values of emissions and lifetime and their uncertainties.

There are some other functions that may be useful. In descending order of usefulness:

- `Plotting/fit_var_plotting.m` has several miscellaneous plotting functions. As of 24 May 2016, this is also where the calculation of full uncertainty in the fitting parameters is contained (see the `plot_3_fits` subfunction). I may move that to its own function eventually.
- `preproc_WRF2Data.m` was meant to preprocess WRF-Chem output into Data structure which could be used by `calc_line_density.m` to make line densities. Works fine, just didn't end up using it because the results weren't helpful at the time.
- `fit_line_density_variation.m` was a way to test how the fit would respond if each of its parameters was fixed to a certain value and the others reoptimized. Good for understanding the uncertainty in the parameters in more detail.
- `Scripts/run_many_line_densities.m` is a function used for **?** to automate the creation of multiple line densities for different cities, with different wind bins, using different *a priori* profiles in the retrieval. Can serve as a prototype for your own automation.
- `Scripts/run_many_emg_fits.m` is likewise a function to automate the creation of many line densities.
- Other files in the Scripts folder (if they're there) are older, less general versions of these last two.
- Many other files in the main `Emissions` folder involved using Monte Carlo sampling to try a different way of estimating uncertainty. It never quite worked, so take these with a grain of salt.

11.2.4 Summary

1. Prepare your wind information by creating a vector of speed (in whatever units you prefer) and direction (in degrees counterclockwise from east, north is $+90°$, south is $-90°$).
2. Point `calc_line_density.m` to the proper directory and files, give it the wind data vector, wait for a while.
3. Give the output of `calc_line_density.m` to `fit_line_density.m` to generate the fit.
4. See `fit_var_plotting.m`, subfunction `plot_3_fits.m` for how to compute emissions, lifetime, and uncertainties.

Appendices

A Running downloads in the background

The simplest way to run downloads in the background, if using the command line on a Mac or Linux machine is to use the `screen` utility. (If you are working on a Windows machine, and having trouble with this in Cygwin, you can always log into the satellite download computer to do it.) `screen` is a utility that lets you start a terminal-within-a-terminal session that will stay alive even if you close the actualy Terminal application, or log out of a computer you are SSH'ed in to.

`screen` should already be install on any Linux system; for a Mac, you'll need to use **Homebrew** or **Macports** to install it. These are linux-like package managers for Mac. Generally you want to install only one. (I prefer Macports, though Homebrew is getting more popular lately.) On Windows machines using Cygwin, if `screen` is not installed, you should be able to install it by rerunning the `setup-x86_64.exe` (or `setup-x86.exe` if you are by some random chance of fate still using a 32-bit machine.)

Start a `screen` session from Terminal by typing the command `screen`. (You can also name your screen session with `screen -S <name>`, replacing `<name>` with whatever you want.) Execute your download command exactly how you would normally. Then, press **Control+a** followed by **d**. This will "detach" your screen session. You can reattach to it with the command `screen -r`. (If you have more than one, you'll need to specify which one either by the name given with the `-S` flag mentioned above, or with it's number. `screen -ls` will list all active screens.) You will see that the download continued without interruption. As long as your computer is on, screen sessions can continue running in the background. When the download finishes, reconnect to the screen session and type `exit` to kill the screen session.

B Setup for automatic downloads

B.1 OMNO2

Sources:

- http://disc.sci.gsfc.nasa.gov/additional/faq/general_user_services_faq.html#subscription
- http://disc.sci.gsfc.nasa.gov/additional/scienceTeam/s4pa_mri.html

Automatic downloads for OMNO2 are the trickiest part of this because they require a data subscription in order to be downloaded automatically. I chose to use an inactive subscription in which our computer makes a request, receives the list of URLs, and executes the download. This gives us the maximum control over when the download process occurs.

Because of the nature of these systems, I recommend that automatic downloading be carried out on an Ubuntu or other Linux-based OS. These tools are easiest to configure in such an OS, and these instructions are meant for those.

1. The computer must have **openssh** installed. I will be very surprised if it doesn't, but you can make sure by checking that there is a folder `/usr/lib/openssh`. If not, install with:

   ```
   sudo apt-get install openssh
   ```

2. The way that the machine request interface works is that upon a request (delivered by wget'ing a formatted URL), it will produce a delivery notice with the URLs to the files requested via secure file transfer protocol (sftp) to the designated computer at a fixed IP address. I have the users `nasa` set up on the black Compaq computer at 128.32.208.13 for this purpose.

3. For security, the computer has password authentication for SSH disabled, only asymmetric key login allowed. Authorized public keys need to be pasted into the `~/.ssh/authorized_keys` file (one per line) for the correct user. The OMNO2 DISC sent their public key to me upon request, this should go in the `authorized_keys` file for the `nasa` user. (Password authentication is disabled by changing the line in `/etc/ssh/sshd_config`

   ```
   PasswordAuthentication yes
   ```

   from `yes` to `no`. If there is a `#` sign at the front of the line, remove it.)

4. To set up sftp, find the line in `/etc/ssh/sshd_config` containing:

   ```
   Subsystem sftp
   ```

   Change it and add lines following it to match:

   ```
   Subsystem sftp /usr/lib/openssh/sftp-server
   Match group ftpaccess
   ChrootDirectory %h
   X11Forwarding no
   AllowTcpForwarding no
   ForceCommand internal-sftp -d /www
   ```

   This will require the user to be in the group `ftpaccess` (next step), lock it to its home directory, and automatically start logged into the `www` directory in its home folder.

5. Add the `nasa` user to the ftpaccess group

   ```
   sudo groupadd ftpaccess
   sudo usermod -a -G ftpaccess nasa
   ```

   Also create a `www` directory in the `nasa` user's home folder if it does not already exist.

This should be sufficient. You may test this by adding your own public key to the `nasa` user's authorized keys file and attempting to sftp in:

```
sftp nasa@128.32.208.11
```

If successful, you should be in a usual ftp prompt in the www directory.

B.2 MODIS

Sources:

- http://modaps.nascom.nasa.gov/services/faq/

MODIS requires no special setup to establish a subscription, as it has an open ftp server. We access it using a combination of `wget` and SOAPpy.

B.3 General setup

All the necessary scripts are written in Bash or Python and are contained in the BEHR repo's Downloading folder. These can be automated through the use of crontab. Most can be run from your user, however some need to be run as root (to allow access to the nasa user).

First, the root crontab, edit with `sudo crontab -e`:

```
# m h  dom mon dow   command
30 3 * * 6 /home/josh/Documents/MATLAB/BEHR/Downloading/order_omi.sh OMNO2
30 3 * * 5 /home/josh/Documents/MATLAB/BEHR/Downloading/order_omi.sh OMPIXCOR
```

And the regular user's crontab, edit with `crontab -e`:

```
# m h  dom mon dow   command
0 1 * * 6 /home/josh/SatDL/get_modis.sh
0 4 * * 7 /home/josh/SatDL/get_omi.sh OMNO2
0 4 * * 5 /home/josh/SatDL/get_omi.sh OMPIXCOR
0 1 * * 1 /home/josh/SatDL/run_read_main.sh
0 1 * * 2 /home/josh/SatDL/run_behr_main.sh
0 1 * * 3 /home/josh/SatDL/run_publishing.sh
```

Note: be VERY CAREFUL not to type ***`crontab -r`*** *as this erases the existing crontab without any prompt to ask if you are sure!!!*

This reads as, e.g. "run `order_omi.sh OMNO2` at 4:00 AM on Saturday morning." The only two key points for the order are:

1. `order_omi.sh` must run before `get_omi.sh`.
2. `get_omi.sh` and `get_modis` should run before any of the `run_xxx.sh` files.

These scripts rely on the following environmental variables and aliases being set. Add the following to your `~/.bashrc` file:

```
# Define the environmental variables and aliases first,
# this way they'll be defined before the check if the shell
# is interactive

# User defined environmental variables
export OMNO2DIR="/mnt/sat/SAT/OMI/OMNO2/version_3_3_0/download_staging"
export OMPIXCORDIR="/mnt/sat/SAT/OMI/OMPIXCOR/version_003/download_staging"
export MODDIR="/mnt/sat/SAT/MODIS"
export SPDIR="/mnt/sat/SAT/BEHR/SP_Files_2014"
export BEHRDIR="/mnt/sat/SAT/BEHR/BEHR_Files_2014"
export AMFTOOLSDIR="${HOME}/Documents/MATLAB/BEHR/AMF_tools"
export NO2PROFDIR="/mnt/sat/SAT/BEHR/Monthly_NO2_Profiles"
export HDFSAVEDIR="/mnt/sat/SAT/BEHR/WEBSITE/webData"
export MATRUNDIR="$HOME/Documents/MATLAB/Run"

# Aliases
```

```
alias startmatlab="export MATLAB_DISPLAY=0; /usr/local/MATLAB/R2014b/bin/matlab -nodisplay -nosplash"

# Add to PATH
export PATH="/usr/local/bin:$PATH"
```

To the root `.bashrc` in `/root/.bashrc` (you'll need to use `sudo` to edit it) add the `OMNO2DIR` variable:

```
# User defined environmental variables
export OMNO2DIR="/mnt/sat/SAT/OMI/OMNO2/version_3_3_0/download_staging"
export OMPIXCORDIR="/mnt/sat/SAT/OMI/OMPIXCOR/version_003/download_staging"
```

just before the following lines:

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

Save this but DO NOT exit before testing that you can still make yourself root with `sudo su`.

To allow automatic emailing of success or failure, link or copy the `automessage.sh` script to `/usr/local/bin` (making sure it's executable, `chmod u+x automessage.sh`) and install the sendEmail package (`sudo apt-get install sendEmail`) if it is not already. Finally, place the following information in the file `.gmailcredentials` in both your home folder and `/root`:

```
behrautodownload@gmail.com
!%e^XN!8Lk-j=HYRM-2S&wfvky&T5U+&
```