One Layer Land Atmosphere Model Notes

Adam Bauer*

March 12, 2021

Contents

1 Code Infrastructure

In this section, I will briefly outline the motivation for writing each of the programs in the subsection headers. I will then discuss their current functionality, i.e., what is computed by them and how it factors into the One Layer Land Atmosphere (OLLA) model. Lastly, I will touch on outstanding improvements that need to be made to each program, as well as how extensively each module within the program has been tested.

Note the naming convention that we use is `get ...` is a function that "gets" whatever the ... is. Classes (if we had any) would just be capitalized, i.e., `Class`. The master file (i.e., the thing that gets run at the end of the day) has Master in the program name.

1.1 MasterOLLA

`MasterOLLA` is the master file of the simulation. It calls functions within each of the following programs, as well as handling the plotting of the output. The basic structure is to

*Department of Physics, University of Illinois Urbana-Champaign, 1110 W Green St Loomis Laboratory, Urbana, IL 61801

1. Gather precipitation, radiation, temperature and humidity data;
2. Send these data to `getStats`, where the relevant statistics are computed, and an array of relevant statistical quantities is returned;
3. Feed these statistical quantities to `getForcingNetCDF` where a synthetic forcing NetCDF is created that is tuned to the observations found in the previous step;
4. Use the NetCDF generated in the step prior to force the simulation in `getOLLA`;
5. Plot the time series of temperature.

Current Functionality

The entirety of `MasterOLLA` has been written and runs well. Plots of moisture content and temperature are produced for as many summers as one desires.

Improvements Needed

No obvious improvements need to be made to `MasterOLLA`, other than making the figures look a bit more professional, which can be saved for later.

1.2 MasterDataStatistics

`MasterDataStatistics` is a notebook that was developed to investigate how well `getForcingFunctions` was creating synthetic noise that matched up to the observational record. The quantities that are investigated are

1. The distribution of daily mean radiation values and the distribution of daily max radiation values are calculated and plotted;
   (a) The standard deviation and skewness of the daily mean radiation distribution are also computed in the notebook.
2. The autocorrelation of the daily mean radiation distributions are computed and plotted;
3. The distribution of mean precipitation event amounts;
4. The average diurnal cycle for radiation and precipitation.

Current Functionality

The code computes the quantities and distributions in the order described above. The notebook relies on `getRemoveDefects`, `getPrecipEventMeansXARRAY_1D`, `getAutocorrelationTrunc`, and `getDailyMeanMaxStdSkewArrays` at various points.

Improvements Needed

The code right now runs as desired. It is most likely worth checking the computation of the average diurnal cycle to make sure this was done correctly (especially for precipitation), but otherwise the code works as expected.

1.3 getStats

`getStats` is a program that calculates the following statistical quantities to be returned to the Master: radiation standard deviation, maximum radiation peak value, radiation average, minimum peak radiation value, temperature average, cumulative amount of precipitation in a summer, precipitation standard deviation, and the individual rain event mean. It also houses three additional functions: `getRemoveDefects`, `getAutocorrelationTrunc` and `getDailyMeanMaxStdSkewArrays`.

Current Functionality

The function that calculates these quantities is aptly named `getStats`, and takes a list of file names as its arguments for the radiation data and precipitation data, as well as the keyword that corresponds to the parameter of interest in each NetCDF file. The function `getStats` returns a seven element long list of statistical quantities needed by other parts of the code. `getStats` has several dependencies found in the same program.

First, there is `getMeanStdMaxMin`. As arguments, `getMeanStdMaxMin` takes a list of file names (that were fed into `getStats`) and the keyword for the data parameter one wants the mean, standard deviation, maximum and minimum of. The returns of this function are exactly those values, as well as the average cumulative amount of a parameter in each file (i.e., each summer).

The second dependency of `getStats` is `getPrecipEventMeansXARRAY_1D`. The arguments of this function are the NetCDF files to be examined, and the threshold for the amount of precipitation needed to be considered an "event." The function returns the mean of individual precipitation events rather than the mean monthly precipitation, as `getMeanStdMaxMin` does. To do this, the code scans each file by the minute, and returns when the precipitation value is above the threshold to be considered an "event." The code then monitors how long this event persists; whenever the precipitation levels dip beneath the threshold again, the mean of the event is taken, and the process continues again. This function returns the mean of the individual precipitation event means.

The third dependency of `getStats` (and many other bits of code) is `getRemoveDefects`. This program takes as arguments the array of values that need to be corrected and the data keyword in the NetCDF that corresponds to the data in the given array. This code was made as a corrective measure for inappropriate data values, i.e., values of -9999 for precipitation and radiation. To remove the defect, the code finds the data points that have troublesome values, and linearly interpolate over them. For radiation, the code flags all values where the amount of radiation is less than or equal to zero as corrupted data points. For precipitation, zero is an allowable value, but any precipitation values that are less than zero will be interpolated over. `getRemoveDefects` was extensively tested on the observational data.

Independent from the `getStats` routine, there is also `getAutocorrelationTrunc` and `getDailyMeanMaxStdSkewArrays`. The former simply takes a single array as its argument, and computes the autocorrelation of that array and itself. However, it uses the `numpy` function `numpy.correlate`, which returns the autocorrelation for all $t \in \mathbb{R}$. However, we're only interested in the autocorrelation for $t \in \mathbb{R}^+$, so this function just truncates half of the array for which $t < 0$, for convenience.

`getDailyMeanMaxStdSkewArrays` takes as arguments the file names one wants the daily quantities computed for and the data keyword corresponding to the values of interest. This function only differs from `getMeanStdMaxMin` in that the mean, max, standard deviation and skewness arrays are returned for *daily* time series, not for entire summers, as is done in `getMeanStdMaxMin`. The data fed into `getDailyMeanMaxStdSkewArrays` is cleaned using `getRemoveDefects` and an array of daily means, etc. are returned.[1]

Improvements Needed

Each function named above has been extensively tested and works as intended. `getRemoveDefects` is by far the biggest time sink in the code, but this is likely not something that can be fixed anytime soon. The time investment comes from a `numpy` function `numpy.where`, where the code searches where in an array a condition is satisfied and returns an array of index values where the condition is true. Therefore, for each data file, all 132480 values must be checked, which just takes a long time.

1.4 getForcingFunctions

`getForcingFunctions` is a program that generates a NetCDF file of synthetic forcing for a specified number of years (or summers, more precisely). The forcing generated is red in its spectrum, and is tuned to observational data, the statistics of which were calculated by `getStats`. Note that the forcing is resolved by the minute.

An important note about this program is that it is almost entirely based off of a previous code written by Lucas Zeppetello (University of Washington) for his Simple Land Atmosphere Model (SLAM).

Current Functionality

The main function of `getForcingFunctions` is `getForcingNetCDF`. The arguments of this function are the number of summers that need forcing generated, the statistics of the observational data, and the file name (and, by extension, the path) that the newly generated forcing NetCDF will be named (and saved to). This function is plagiarized directly from Zeppetello's work. It has one dependency, `getRedNoise`.

`getRedNoise` is a function that generates the actual forcing data that gets put in the NetCDF created by `getForcingNetCDF`. The function takes the statistics of observational data as its only argument. The program first generates some blank daily and monthly cycles. Then, these cycles are populated

[1] In the most recent version of this function, only mean and maximum arrays are returned, as it was determined that the standard deviation and skewness arrays were not relevant.

by interpolated data that is self correlated and interpolated. Lastly, the radiation forcing is calibrated to the cloud fraction values in the data, and precipitation events are set to only occur on sufficiently cloudy days. This function has two dependencies. It uses `getInterpolate` to interpolate data trends to the desired time, whereas `getPrecip` generates the precipitation forcing. The code also has built in the ability to only have precipitation events happen on cloudy days, for consistency.

The first dependency of `getRedNoise` is `getInterpolate` is a function that interpolates data from its current length to a new length. The arguments of this function are the array to be interpolated and the desired array length. It returns the new array interpolated using `numpy`'s `interp` function.

The second dependency is `getPrecip`. The argument of this function is simply the statistics of the observational data. This program was plagiarized from Zeppetello's work as well. It returns an array of precipitation events that have been tuned to the observational data.

Improvements Needed

The program currently runs well. However, the results are unsatisfactory, and do not accurately reproduce data that could pass as an SGP data set. The precipitation diurnal cycle is too long, and the radiation mean is too low. In addition, the diurnal cycle lasts for longer than the observational diurnal cycle lasts for. Much work is needed in tuning this program to accurately reproduce observational data, and will be the topic of future work.

1.5 getOLLA

`getOLLA` is where the actual model is forced using the synthetic forcing created in `getForcingFunctions`. The end result is the time series for surface temperature and soil moisture anomalies.

Current Functionality

The main function for this program is `getOLLA`. This function takes the filename of the synthetic forcing as its only argument. It then expands this NetCDF into multiple numpy arrays that are fed into `getIntegration` to be officially run through the model.

`getIntegration` numerically integrates the model equations using Newton's method. As such, the arguments for `getIntegration` are the forcing arrays that were taken from the forcing NetCDF. For reference, the model equations are,

$$C\frac{dT}{dt} = \mathcal{F}(t) - \alpha\left(T - T_0\right) - Lvm\left(q_s(T) - q(t)\right), \tag{1.1}$$

$$\mu\frac{dm}{dt} = \mathcal{P}(t) - vm\left(q_s(T) - q(t)\right), \tag{1.2}$$

where C is the heat capacity of water, μ is something like the moisture capacity of the soil, T is temperature, $\mathcal{F}(t)$ is the radiative forcing, T_0 is the freezing temperature of water, α is the radiative feedback, L is the latent heat of water, v is the surface resistance of the soil, m is the soil moisture content, $\mathcal{P}(t)$ is the precipitation forcing term, q is the specific humidity of the atmosphere, $q_s(T)$ is

the specific saturation humidity, given by

$$q_s(T) = \frac{0.622}{P_{surf}} \exp\left(\frac{L}{R_w}\left(T_0^{-1} - T^{-1}\right)\right),$$

where R_w is the specific gas constant of water and P_{surf} is the surface pressure of the Earth. In their discretized forms, (1.1) and (1.2) take the form

$$T_i = T_{i-1} + \underbrace{\frac{1}{C}\left(\mathcal{F}_{i-1} - \alpha\left(T_{i-1} - T_0\right) - Lvm_{i-1}\left(q_s(T_{i-1}) - q_{i-1}\right)\right)}_{=:\Xi_{i-1}},$$

$$m_i = m_{i-1} + \underbrace{\frac{1}{\mu}\left(\mathcal{P}_{i-1} - vm_{i-1}\left(q_s(T_{i-1}) - q_{i-1}\right)\right)}_{=:\Sigma_{i-1}},$$

so we have

$$T_i = T_{i-1} + \Xi_{i-1}, \tag{1.3}$$

$$m_i = m_{i-1} + \Sigma_{i-1}, \tag{1.4}$$

where Ξ_{i-1} and Σ_{i-1} are the temperature and moisture fluxes, respectively. The initial conditions for our simulation are zero, which is simply assuming the system is initially in equilibrium, prior to forcing

The only dependencies of `getIntegration` are `getTempFlux` and `getMoisFlux`, which calculate Ξ_{i-1} and Σ_{i-1}, respectively. They take the relevant terms as their arguments.

Improvements Needed

This program works more or less how it should right now. However, the integration results are *ultra* sensitive to evaporative cooling; in fact, as far as we can tell, the main issue with the integration is that moisture isn't evaporating fast enough. The could be for numerous reasons; for one, the synthetic forcing was shown using `MasterDataStatistics` to have far too little (almost by hafl!) radiative forcing from the Sun compared to the observational record. Also, the precipitation events seem to be far more common in the synthetic data than in the observational data.

1.6 Things to do

- Tune `getRedNoise` to actually reproduce the observational data like we want it to.
- Tune `getOLLA` to get moisture to evaporate faster?