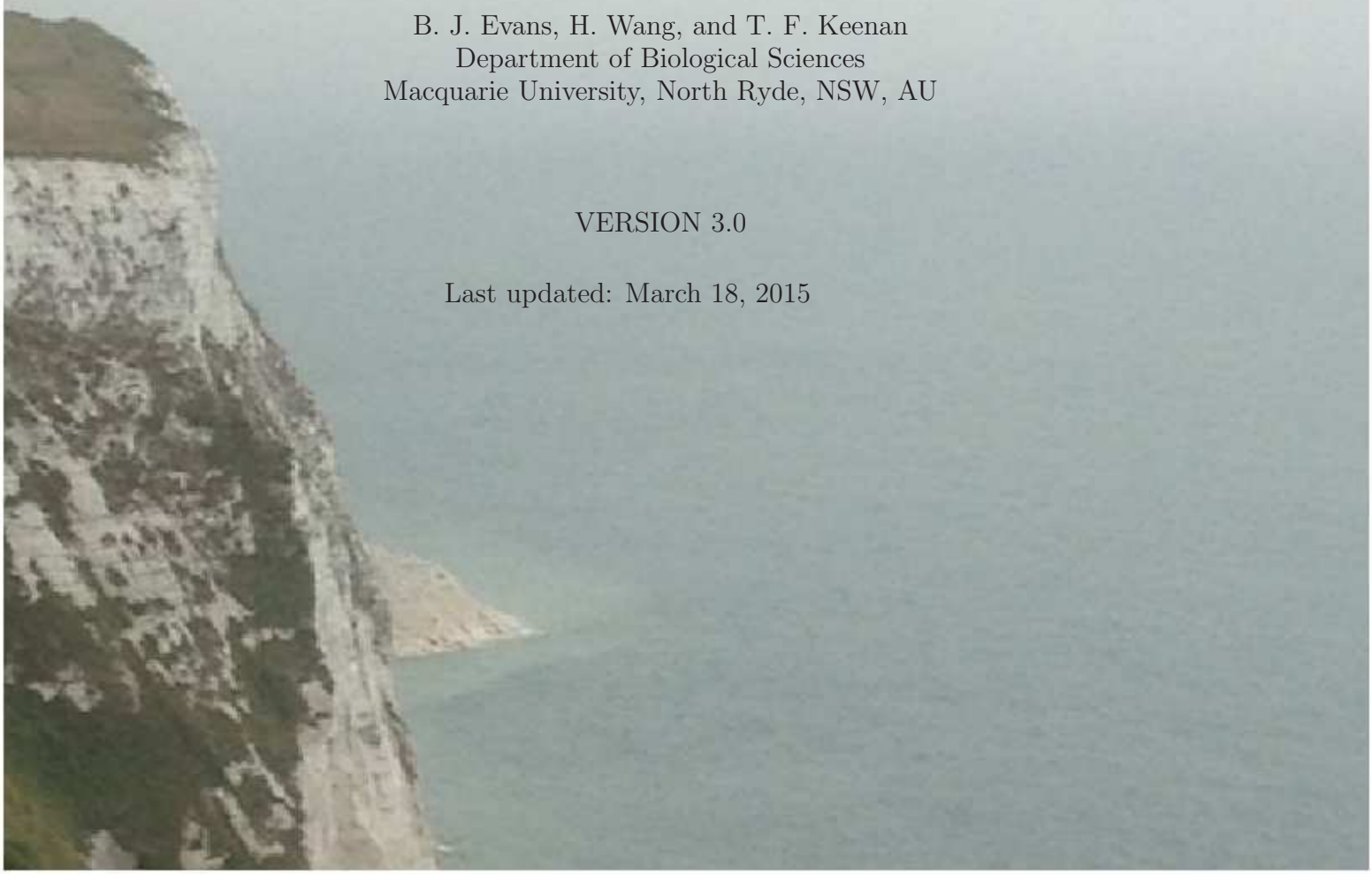
The Global ecosystem Production in Space and Time (GePiSaT) Model of the Terrestrial Biosphere

I. C. Prentice
AXA Chair Programme in Biosphere and Climate Impacts
Department of Life Sciences and Grantham Institute for Climate Change
Imperial College London, London, UK

T. W. Davis and X. M. P. Gilbert
Department of Life Sciences
Imperial College London, London, UK

B. J. Evans, H. Wang, and T. F. Keenan
Department of Biological Sciences
Macquarie University, North Ryde, NSW, AU



VERSION 3.0

Last updated: March 18, 2015

Contents

I Project Documentation **5**

1 Introduction **6**

2 The Modeling System **7**
2.1 Running the Python code 7
2.1.1 table_maker.py 7
2.1.2 db_setup.py 9
2.1.3 model.py 9
2.2 Running the R code 13
2.2.1 plot_partitioning.R 13
2.2.2 plot_outliers.R 14
2.2.3 summary_stats.R 14
2.2.4 plot_gpp.R 14
2.2.5 plot_lue.R 15

3 Observation Data **16**
3.1 NEE & PPFD 16
3.2 SWdown 18
3.3 fPAR 19
3.4 VPD 21
3.5 Tc, Pre and Elv 21
3.6 CO2 22
3.7 Alpha 22

4 Future Work **24**

II Database Documentation **25**

5 Introduction **26**

6 Installation **27**
6.1 Virtual machine setup 27
6.2 Native Linux setup 28
6.3 Resizing Linux for postgreSQL database 29

7 Setup **33**

8 Structure **36**
8.1 Meta-data table 36
8.1.1 Field descriptions 36

8.1.2 Meta-data SQL table creation command 38
8.1.3 Acquiring data 39
8.2 Variable list table 40
8.2.1 Description 40
8.2.2 Variable list SQL table creation command 41
8.3 Data set table 42
8.3.1 Description 42
8.3.2 Data set SQL table creation command 42

III Model Documentation 43

9 Environment Setup 44
9.1 Python environment 44
9.2 R environment 44

10 Next-Generation Model Development 46

A Python Code Snippets: 47
A.1 peirce_dev.py 47
A.2 outlier.py 49
A.3 netcdf.py 51
A.4 hdf.py 52

B Resampling MODIS Data: 53
B.1 The QGIS method 53
B.2 The Python method 55

C Some Useful SQL Commands 57
C.1 List of variables 57
C.2 Find msvidx 57
C.3 Delete single variable data 57

List of Notations

EVI enhanced vegetation index 19

fPAR Absorbed fraction of PAR [unitless] 20

VPD Vapor pressure deficit [kPa] 21

e_s Saturation vapor pressure [kPa] 21

Part I
Project Documentation

1 Introduction

This project is aimed to develop a modeling system for global hindcasting and analysis of spatial and temporal patterns in terrestrial gross primary production (GPP). This system takes a simplistic approach which makes the best use of observational data (from flux towers, meteorological stations, and remote-sensing satellites) while defensibly representing the principal eco-physiological processes that govern GPP.

The modeling work is divided into stages to assess the next-generation GPP model from the ground up. The first stage is the acquisition and decomposition of high-resolution CO_2 eddy flux, F (i.e., net ecosystem exchange), and photosynthetic photon flux density, Q, into ecosystem respiration and GPP. The high-resolution data was acquired from eddy covariance flux towers (e.g., public FLUXNET data) across the world. The regression of Q versus F into flux partitioning parameters is based on the work of Ruimy et al. 1995. The second stage is to integrate the high-resolution Q and estimated GPP into monthly totals. The integration requires gap-filling of the monthly time series of Q, due to missing or errant observations. The third stage is the analysis and fitting of the light-use efficiency quasi-theoretical model to the observed monthly GPP.

The basis of the modeling system is an efficient database structure designed to hold the variety of observational data necessary to complete each stage of the modeling. This modeling work strives for clarity and uniformity so that it may be used by researchers across disciplines. The use of open-source software (i.e., PostgreSQL) and programming languages (e.g., Python and R) allows for portability and transparency. The model will invite a range of applications to the analysis of climate and CO_2 change impacts on ecosystem processes.

2 The Modeling System

This model has been written to take advantage of three key software packages: Python (version 2.7), PostgreSQL (version 9.1), and R (version 2.14). The version numbers presented alongside these three software packages are meant as a reference as to when this model was created. Limitations may be encountered with model performance or functionality using older/newer versions of these three software packages.

The core of the model is written in and operated under the Python programming language. This includes all the model computations, data input and output. PostgreSQL provides a versatile database environment for storing all the model data (e.g., meta data, observations, etc.). An overview and installation instructions for the database is presented in Sections 5–8. R is implemented to perform analytics and data plots of model results.

2.1 Running the Python code

There are three files which make up the Python portion of this model: `table_maker.py`, `db_setup.py`, and `model.py`. Each file runs independently of the others. The following will present an overview of each file including its functions and how it is operated.

2.1.1 table_maker.py

The Python portion of this model begins with `table_maker.py`. The purpose of this file is to convert the format of any source data to that which conforms to the database design used in this modeling system. The database design (as described in Section 8) consists of three tables: `met_data`, `var_list`, and `data_set`.

This script is responsible for producing outputs conforming with the database structure of the last two tables (i.e., `var_list` and `data_set`). This is accomplished via Python classes and functions.

For creating the necessary variable table data (i.e., `var_list`), the Python class, VAR was created. This class holds the main variables required as output to the `var_list` table, namely: *msvidx*, *stationid*, *varid*, *varname*, *varunit*, *vartype*, and *varcore*.

In the current implementation of the model, only core variables are considered; therefore, all variables have the binary "1" associated with the *varcore* field. All core variables are given in a Python dictionary, coreVars, where each *varname* is presented with its associated *varid*. The *varid* field is simply a numeric that is incremented for each additional variable added to the model. Similarly, a Python dictionary, variableUnits, associates each *varname* with its appropriate measurement units, i.e., the *varunit* field. Table 1 provides an example of the core variables, with their units and iden-

tification number, that are currently used in the GePiSaT database (see §3 for details regarding each variable).

Table 1: Current list of variable names, units, and identification numbers used in the GePiSaT database.

varname	***varunit***	***varid***
NEE_f	μmol CO_2 m^{-2} s^{-1}	1
PPFD_f	μmol photons m^{-2} s^{-1}	15
SWdown	W m^{-2}	18
FAPAR	NA	19
VPD	kPa	20
CO2	ppm	21
Tc	°C	22
Pre	mm	23
Elv	m	25
alpha	NA	26

Due to the difference in how stations are named depending on whether they are flux towers or gridded pixels, the *vartype* must be known for the variable in question (e.g., "flux" for flux towers and "grid" for gridded pixels). Depending on the *vartype*, the *stationid* (and therefore the *msxvidx*) can be determined. If the *vartype* is "flux," then the *stationid* is simply read from the filename (as is the convention with flux tower data to have each file saved with the *stationid*). If the *vartype* is "grid," then the *stationid* is sent to the VAR class as a tuple in the place of the file name.

The rest of this script is dedicated to parsing information for each data type (e.g., HDF4, netCDF, CSV) and data source (e.g., flux towers, MODIS, WATCH, CRU, etc.). For some of the more complicated data sources, a Python class is also created to handle the computations necessary. This is true for the flux data (FLUXDATA class) to assist in processing the QC flags associated the NEE and GPP variables. The WATCH forcing data class (WATCHDATA) is used to assist with the timestamps. For MODIS EVI the MODISDATA class is used to perform the upscaling to 0.5° resolution. The CRUDATA class performs the calculations necessary to derive the vapor pressure deficit (VPD) from the max and min air temperature and actual vapor pressure. The GLASDATA class is similar to the MODISDATA class where it upscales canopy height data to 0.5° resolution.

To make processing easier, Python functions are also created to perform an array of tasks. Simple file creation and headerline writing has been implemented in the writeout() function. Process functions are created for each data source to handle the various file types. The table below lists the process function names, the data source, and the file type.

The remainder of the script is left only to defining the directory where the source data exists, defining any ancillary parameters, and running the

Table 2: Process functions for various data sources and file types used in the var_list table.

Function Name	Data Source	File Type
process_flux()	Flux towers	CSV
process_watch()	WFDEI	netCDF
process_modis()	MODIS	HDF4
process_cru_elv()	CRU TS 3.00	DAT
process_cru_vpd()	CRU TS 3.21	netCDF
process_cru()	CRU TS 3.21	netCDF
process_glas()	GLAS	GeoTIFF
process_alpha()	CRU TS 3.21 / WFDEI	ASCII Raster

process function associated with the source data. Output is saved in the same directory as the source data.

2.1.2 db_setup.py

This Python script interfaces with the postgreSQL database to initialize the database tables and populate them with data. For each of the three tables (i.e., `met_data`, `var_list`, and `data_set`), there are two functions: create table and populate table. The create table function holds the SQL query for the table schema including the table column headers and their associated data types (e.g., integer, string, date, etc.).

To make the creation of tables simpler, two functions were created, resetdb() and cleandb(). For initialization of fresh unpopulated tables, the resetdb() function deletes all tables (if they exist) and creates them anew. If the tables already exist, the information within them can be deleted using the cleandb() function. An associated clean_table() function is also available to delete data from only one table. The size of the database (in bytes) can be queried following each addition to the database by calling on the db_size() function.

The remainder of the program is simply the definition of where the data (for the three tables) is located. Based on the naming convention of table_maker.py, each file is sought within the directory provided and processed with the appropriate pop function.

2.1.3 model.py

This Python script performs the modeling system's main processing that is outlined in §9. Currently, only stages 1 and 2 are implemented. This consists of the flux partitioning and the basic light-use efficiency modeling.

Classes The model contains three Python classes: FLUX_PARTI, SOLAR, and LUE.

The first class (i.e., FLUX_PARTI) contains all the variables and methods necessary to carry out the monthly flux partitioning. This includes the original flux tower observation data (NEE and PPFD), the statistics associated with these observations, the dynamically calculated flux partitioning parameters, the optimized flux partitioning parameters (and their associated errors and significance), the model fits, the observation data sets with outliers removed based on both the linear and hyperbolic fits, the statistics of the observation data with outliers removed, the outlier-removed dynamically calculated flux partitioning parameters, the outlier-removed optimized flux partitioning parameters (and their associated errors and significance), and the outlier-removed model fits. All of this information is output into the summary_statistics.txt file. There is also a function for calculating GPP (which depends on the flux partitioning parameters).

For the gap-filling of flux-tower observed PPFD (i.e., §**??**), the calculation of extraterrestrial solar radiation (i.e., Eq. **??**) is handled in the SOLAR class, including the conversion of solar energy to units of photon flux and daily integration. This class also calculates the daylight hours.

The LUE class is used for the light-use efficiency modeling. Monthly datasets (e.g., GPP, PPFD, fPAR, VPD, Tc, etc.) are stored for each flux tower. The basic light-use efficiency curve fitting can also be performed.

Main program The main program begins with a get_stations() command that reads the postgreSQL database for stations where the dimension is "0" (i.e., point-based tower measurements) and the geometry is "point" (to distinguish flux towers from CO_2 station measurements) in the meta data and returns a sorted list of station names (i.e., *stationid* fields). To avoid processing all the flux towers, this function can be replaced with a list of specific stations of interest.

Once a list of station names is retrieved, the summary statistics file is created and initialized (using the summary_file_init() function) with appropriate headers. The output file is hard-coded to be saved in a subdirectory named "out." This directory needs to be created before running the model. Lines in the summary statistics file (which are based on the output from the FLUX_PARTI class) are for each month used in the flux partitioning for each flux tower processed. Columns are associated with the statistics defined in the class.

After a new summary file has been created, a blank LUE class is created (called my_lue). The output file for the station specific modeled LUE is also defined. The LUE class stores all the monthly variables required for LUE model and performs the regression on the monthly variables for the basic LUE model for each flux tower. The output to the LUE_All-Stations.txt is

the LUE for each station, the standard error, and the coefficient of determination.

The model goes on to iterate through each flux tower in the station list. A station-specific light-use file is defined to hold the station's monthly LUE variables (e.g., GPP, PPFD, VPD, Tc, etc.).

The starting and ending time points for a flux tower's observation data are queried from the database using the get_dates() function. This function searches the data_set table in the GePiSaT database for the first and last dates where the station has observations of either NEE or PPFD. The starting date (sd) is reset to the first day of the month returned.

Before iterating through the flux tower's data, the associated 0.5° resolution grid cell is determined using the flux_to_grid() function. This function first finds the flux tower's longitude and latitude from the met_data table in the GePiSaT database using the get_lon_lat() function. Based on the flux tower's coordinates, the nearest 0.5° pixel centroid is found using the grid_centroid() function. The grid_centroid() function uses the nearest linear distance between a pair of regularly spaced pixel centroid coordinate to determine which pixel the flux tower is located within. Should the unlikely event occur where a flux tower is positioned at an equal distance between two pixel centroids, the default is to select the northern/eastern pixel. After determining the pixel centroid, the met_data table in the GePiSaT database is queried for the grid *stationid* field with the associated centroid coordinates. The grid *stationid* is then returned to the main program.

With the associated grid station found for the current flux tower, the gridded variable IDs (i.e., the *msvidx* field) needed for the LUE model can be found using the get_msvidx() function.

Once the starting and ending dates are found, the grid containing the flux tower identified, and the gridded data variable IDs saved, the flux tower's data is then stepped through one month at a time.

The first step in the flux partitioning is querying for the monthly NEE and PPFD observation data using the monthly_ppfd_nee() function. This function first finds the station specific variable IDs (i.e., the *msvidx* field) for flux tower's NEE and PPFD observations. Next, to find the end date for the querying period, it is calculated using the add_one_month() function to the current month (i.e., the starting date). These parameters are sent to a pivot table query (postgreSQL's crosstab[1] operator). The pivot table searches three fields in the data_set table: *datetime*, *msvidx*, and *data*. The *datetime* field serves as the row name while the *msvidx* fields serve as the two categories and the *data* field serves as the value. By using both the source SQL and category SQL queries in the crosstab function, the table of results will be paired NEE and PPFD observations (i.e., value columns based on the two categories) for each *datetime* field. If there is a missing PPFD or

[1]tablefunc module: `http://www.postgresql.org/docs/9.1/static/tablefunc.html`

NEE observation at any given *datetime*, the missing data is padded with a blank value. For the flux partitioning, only matched pairs of NEE and PPFD can be used; therefore, the final step is to filter out any rows that do not contain both NEE and PPFD data.

Once the arrays of PPFD and NEE observations are returned to the main program, their content is checked to make certain enough data is present for flux partitioning (i.e., length of arrays must be greater than three). The flux partitioning is performed using the partition() function. The partition() function allows the user to define the parameter to_write as either "0" or "1." If to_write is defined as "1," then observation files (for each month) are written. If the parameter rm_out is also set to "1," then outlier removal will be performed and monthly outlier-free observation files will also be written.

The partition() function begins by creating a FLUX_PARTI class with the flux tower's NEE and PPFD observations. Inside the class, the statistics for both observation arrays are calculated and the initial model parameters are estimated. Both partitioning models (i.e., the linear and hyperbolic) are then fit to the observations. The optimized model parameters, associated errors, and statistical significance of the fits are saved. If the to_write flag is set to true, then these model fits are written to a file. If the rm_out flag is set to true, the class performs outlier removal using Peirce's criterion (see §**??**). After outliers are identified and removed, the observation statistics are recalculated and model fitting parameters re-estimated. All of this is accomplished using the class methods remove_mh_outliers() and remove_ml_outliers() for the hyperbolic and linear models respectively. Both partitioning models are fit to the outlier-free observations and the optimized parameters, their associated errors, and their significance levels are saved. If the to_write flag is set to true, the outlier-free data are written to file. The last operation performed is to select the model that best represents the data using the class method model_selection(). The model selection is based on the optimized parameters meeting certain requirements (range of validity and significance tests) and the level of model fitness (coefficient of determination greater than 0.5). If none of the models are adequate is representing the data, model selection is set to "0" and processing of this month stops.

Back in the main program, the FLUX_PARTI class used in the flux partitioning is returned by the partition() function (called monthly_parti). If the class was successful at identifying a best representative model (i.e., model selection not equal to "0"), then the second stage of the model is commenced: the light-use efficiency.

In stage 2, the first step is to gap-fill the PPFD observations over the entire month. PPFD observations are measured half-hourly and the gap-filling is performed one day at a time. The gap-filling procedure (see §**??**) is performed by the gapfill_ppfd() function.

The monthly gap-filled PPFD data is then converted to GPP using the calc_gpp() function in the FLUX_PARTI class. The arithmetic error propa-

gation is also calculated. The integration to monthly totals is handled using the Simpson method. Following the integration, the units are updated from μmol·m^{-2} to mol·m^{-2}. The gridded fPAR and ancillary variables (e.g., VPD and canopy height) are queried from the database by first finding the appropriate gridded station (based on the location of the flux tower), then finding the msvidx associated with the gridded station and variable of interest, and finally queried with the get_data_point() function. The monthly GPP, PPFD, fPAR, and ancillary variables are saved to the LUE class and the month's summary statistics are written to file.

After all the months are processed for a flux tower, the LUE class is called to write out the monthly variables and perform the regression for the basic (or next-generation) LUE model. After all stations are processed, the LUE class is called to write out the LUE model results.

2.2 Running the R code

The R scripts are created for performing statistical analysis on the model output.

2.2.1 plot_partitioning.R

This script is for plotting the flux partitioning results (output from the partition() function in model.py). Output files have either the extension ".txt" or "_ro.txt" depending on whether the data was stripped of outliers. This script assumes that these files have been placed in their own separate directories, one for the original observations (i.e., ".txt") and one for observations with outliers removed (i.e., "_ro.txt").

This script will create three output figures for each station. One for the monthly partitioning of the linear and hyperbolic fits to the observation data. One for the hyperbolic flux partitioning of observations with outliers removed. One for the linear flux partitioning of observations with outliers removed. The actual plotting of the flux partitioning for each month is carried out by one of three functions in the script: plot_obs, plot_ro_h, and plot_ro_l.

The plotting functions are sent the filename and directory of the partitioning text files. The partitioning text files are read by a function optim_params, which reads the meta data from the file headers and returns them as a data frame. The plotting functions then create a linear and/or hyperbolic regression line based on the header and content data, plot the NEE versus PPFD, add a legend to the plot, and add the regression lines to the plot. The main loop captures the monthly plots for a flux station and saves them as a postscript image.

2.2.2 plot_outliers.R

This script, similar to the ones described in §2.2.1, plots the flux partitioning based on the output files from model.py. In this case, both the original and outlier-free datasets are required. Once again, these files are assumed to be in their own directories. It is further assumed that these files are further separated into individual directories based on the station ID of the file. This orchestration of the files can be easily accomplished via the file_handler-osx.pl or file_handler-win.pl script.

Each station ID is iterated through and the associated files for the original observation and outlier-free are read from the station-named subdirectories. R handles file reading in a directory organized by file name; therefore, monthly file pairs are assumed.

File pairs are iterated over in the station-named subdirectories, the meta data is read from the file headers, and the data is read from the file contents. Either the linear or the hyperbolic outliers can be plotted (based on separate function calls, plot_outlier_modelL() or plot_outlier_modelH(). Both scripts plot the original observation data in red and the outlier-free data in grey. The result is a plot that highlights the data points that were excluded based on the outlier identification and remove scheme (see §**??**).

2.2.3 summary_stats.R

This script reads the summary_statistics.txt output file for the purposes of updating the dynamic parameter estimation (see §**??**).

For each of the three flux partitioning parameters for the hyperbolic model and the two parameters for the linear model, this script performs regressions on the optimization parameters against statistical properties of the data. For the latest model fits, see Table **??**.

2.2.4 plot_gpp.R

This script reads the LUE text files for each individual station output by model.py. These files are designated by the file extension "_LUE.txt" and are assumed to be located in their own directory. Flux tower station IDs are listed under R-objects, designating them into categories based on the vegetation type (e.g., evergreen needleleaf forest, crops, and grasslands) and climate (e.g., temperate, boreal, and tropical) found in the flux meta data. Each object can then be sent (including an object for all stations) to the function process_gpp which calculates the total annual GPP for each station in the object's list as well as separate the monthly GPP for the purposes of producing object-based monthly statistics of GPP. The annual GPP data is written to file (e.g., "Annual_GPP-All_Stations.txt"). The monthly GPP (which is saved to each R-object) can be viewed as a box plot by calling the box_and_whisker() function.

2.2.5 plot_lue.R

This script creates plots of the basic LUE model from the station LUE output files (i.e., files designated by the file extension "_LUE.txt") by fitting a linear regression through the monthly GPP versus the product of monthly fPAR and monthly gap-filled PPFD. The regression coefficient along with goodness of fit diagnostics (i.e., R^2 and p values) can be added to plot via a legend. The plots are saved to a postscript image.

3 Observation Data

This section reviews the various observation data used in the GePiSaT model, as shown in the following table. Note that not all of the observation data listed in Table 3 is stored in the GePiSaT database (see Table 1).

Table 3: Observation variables used in the GePiSaT model.

Observation	Description	Source
NEE	Net Ecosystem Exchange (CO_2 flux)	fluxdata.org
PPFD	Photosynthetic photon flux density	fluxdata.org
EVI	Enhanced Vegetation Index	lpdaac.usgs.gov
SW_{down}	Shortwave solar radiation	eu-watch.org
vap	Actual vapor pressure	badc.nerc.ac.uk
tmp	Mean air temperature	badc.nerc.ac.uk
pre	Precipitation	badc.nerc.ac.uk
elv	Land surface elevation	badc.nerc.ac.uk
c_a	Atmospheric CO_2 concentration	esrl.noaa.gov

3.1 NEE & PPFD

The high temporal resolution eddy covariance CO_2 flux data is available via flux tower organizations networked around the world. FLUXNET[2] is a universal network for the numerous regional networks, such as:

- AmeriFlux (`http://ameriflux.lbl.gov/`)
- AsiaFlux (`http://www.asiaflux.net/`)
- CarboEurope IP (`http://www.carboeurope.org`)
- Fluxnet-Canada (`http://fluxnet.ccrp.ec.gc.ca`)
- LBA (`http://daac.ornl.gov/LBA/lba.shtml`)
- OzFlux (`http://www.ozflux.org.au`)
- TCOS-Siberia (`www.bgc.mpg.de/public/carboeur/web_TCOS/`)

Figure 1 shows the locations of the more than 500 flux towers distributed over the world, color-coded by their associated regional networks. Despite the numerous flux tower networks in the FLUXNET archives, not all tower data is openly accessible to researchers.

For the purposes of the GePiSaT model, flux tower data (i.e., NEE and PPFD in Table 3) were acquired from the FLUXNET Synthesis Dataset[3]

[2] `http://fluxnet.ornl.gov`
[3] `http://www.fluxdata.org`

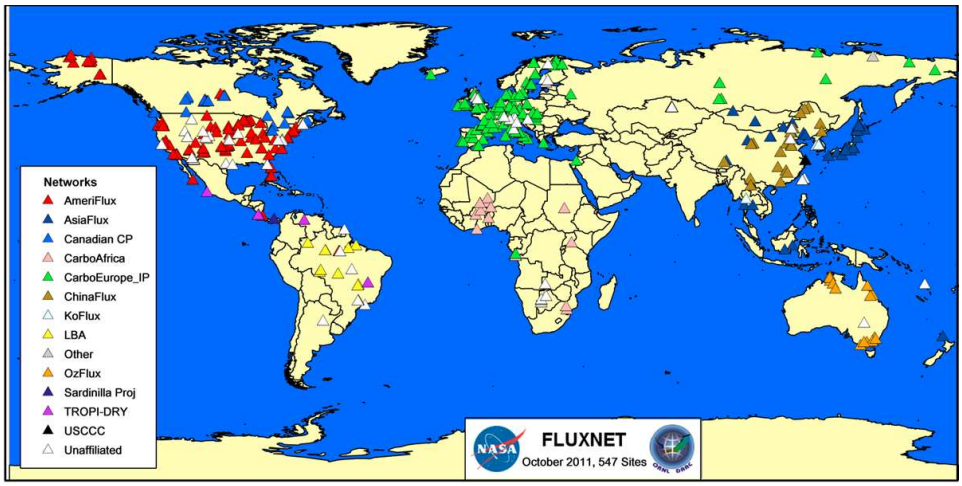


Figure 1: Map of FLUXNET station locations and their associated networks. Source: `http://fluxnet.ornl.gov/introduction`

listed under the Free Fair-Use data policy. In order to obtain this data, an account was created for the Free Fair-Use data[4]. As a registered user, access to the Free Fair-Use flux tower data becomes available. There are 153 flux towers with at least one annual data file available between 1991 and 2007 under the Free Fair-Use data policy. Note that some flux towers have multiple years of data available. Table 4 shows the distribution of flux towers available for different countries (currently dominated by towers located in Europe and North America).

The data are stored in comma-separated file format (CSV) for each year of available data by individual stations. For completeness, the half-hourly synthetic all-variable data files (i.e., "synth.hourly.allvars.csv" file extension) were downloaded. For the 153 flux towers with Free Fair-Use data, 663 files were downloaded.

The synthesis data for each flux tower provide a complete half-hourly time series of all core and ancillary variables. Gap-filling products are used in places where observations are missing or poor. The use of quality flags for each observation of each variable provides a convenient way of filtering out all non-observed data. For example, accompanying each NEE value is a quality flag that is given a value of '-9999' if the NEE value is missing, '1' or '2' if the NEE value is an observation, and '3' through '5' if the NEE value is gap-filled. Similar quality flags are given to the PPFD values. Missing and gap-filled values are filtered from the synthesis datasets leaving only the observation data. It is these observations that are stored in the GePiSaT database (see NEE_f and PPFD_f in Table 1).

[4] `https://roz.lbl.gov:31633/default.aspx`

Table 4: The number of Free Fair-Use flux towers by location.

Country	**#**	**Country**	**#**
Africa	**4**	*Europe*	**83**
Botswana	3	Austria	1
South Africa	1	Belgium	4
		Czech Republic	2
Asia	**8**	Denmark	4
Indonesia	1	Finland	3
Israel	1	France	7
Russia	6	Germany	8
		Hungary	2
Australia	**4**	Iceland	1
		Ireland	2
North America	**53**	Italy	17
Canada	14	Netherlands	7
United States	39	Poland	1
		Portugal	3
South America	**1**	Slovak Republic	1
Brazil	1	Spain	4
		Sweden	6
		Switzerland	2
		United Kingdom	8

3.2 SWdown

The gap-filling of the high-resolution PPFD (see §**??** for methodology) requires high resolution solar radiation data. Daily downwelling shortwave solar radiation data (global coverage at $0.5° \times 0.5°$ resolution, W m^{-2}) can be acquired from the Integrated Project for Water and Global Change (WATCH[5]), which may be acquired through the WATCH project's FTP server hosted at the International Institute for Applied Systems Analysis (IIASA) in Austria[6]. The most recent data files are provided under the WATCH Forcing Data for the ERA Interim (WFDEI) (Weedon et al. 2012). The ERA Interim has recently been extended to include the time period between 1 January 1979 through 31 December 2012. See Figure 2 for an example of daily SW_{down} for 1 July 2002. The daily SW_{down} is stored in the GePiSaT database (see Table 1).

The data is organized in monthly netCDF files. The data within the netCDF files can be accessed via Python's `netcdf` method in the `scipy.io` module. An example of reading WATCH netCDF files using Python is given in Appendix A.3.

[5] `http://www.eu-watch.org/data_availability`

[6] `ftp://rfdata:forceDATA@ftp.iiasa.ac.at/WFDEI/SWdown_daily_WFDEI/`

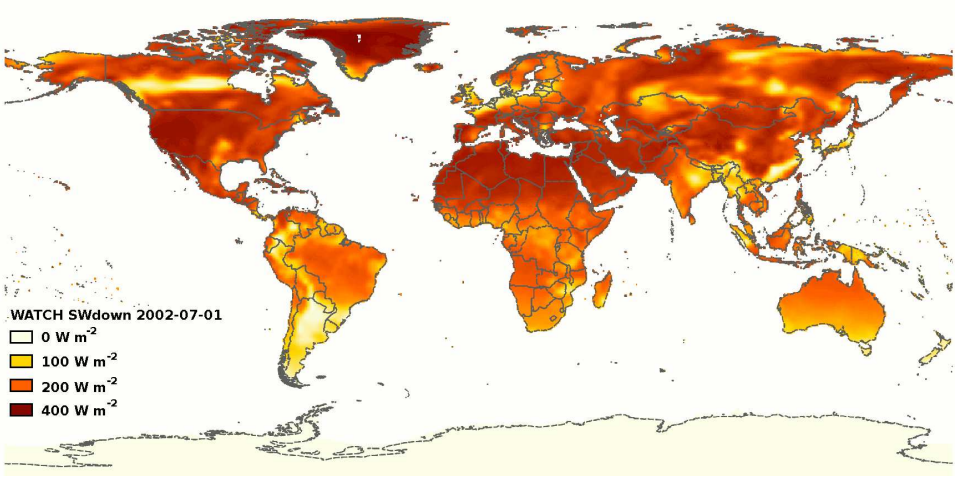


Figure 2: WATCH SW_{down} at 0.5° resolution for 1 July 2002.

3.3 fPAR

During photosynthesis, pigments within plant leaves (e.g., chlorophyll and carotenoids) absorb solar radiation particularly well the wavelengths within the blue and red visible spectrum (i.e., 0.40–0.51 μm and 0.61–0.70 μm, respectively). Light energy within this range (i.e., 0.40–0.70 μm) has come to be known as photosynthetically active radiation (PAR) (Oke 1987). Not all of the energy available within the PAR waveband is absorbed by the vegetation canopy. Therefore, it is often considered more important to investigate the fraction of absorbed PAR (fPAR). Process-based methods of estimating fPAR often rely on some type of vegetation index (e.g., leaf area or satellite-derived index).

The GePiSaT model implements a diagnostic approach to vegetation greenness by relying on remotely-sensed green vegetation as a model input (as opposed to attempting its own vegetation simulation). There are a variety of remotely-sensed vegetation greenness indexes based on the moderate-resolution imaging spectrometer (MODIS) spectral bands. For our purposes, the enhanced vegetation index (EVI) is used to estimate vegetation greenness.

EVI uses near-infrared (NIR) and visible red and blue MODIS spectral bands. The calculation is as follows (Huete et al. 2002, Eq. 2):

$$\text{EVI} = G \frac{\rho_{nir} - \rho_{red}}{\rho_{nir} + C_1\ \rho_{red} - C_2\ \rho_{blue} + L_{bg}} \tag{1}$$

where:

G = gain factor (i.e., 2.5)
L_{bg} = canopy background adjustment factor (i.e., 1.0)
C_1 = aerosol resistance coefficient (i.e., 6.0)

C_2 = aerosol resistance coefficient (i.e., 7.5)
ρ = corrected (or partially corrected) surface reflectances

The calculation to convert EVI to fPAR is expressed as (Xiao et al. 2005, Eq. 11):

$$\text{fPAR} = a\,\text{EVI} \tag{2}$$

where:
$a = 1.0$
fPAR = absorbed fraction of PAR
EVI = MODIS-derived enhanced vegetation index

The fPAR data calculated from the upscaled MODIS-based EVI, by Eq. 2, is stored in the GePiSaT database (see FAPAR in Table 1).

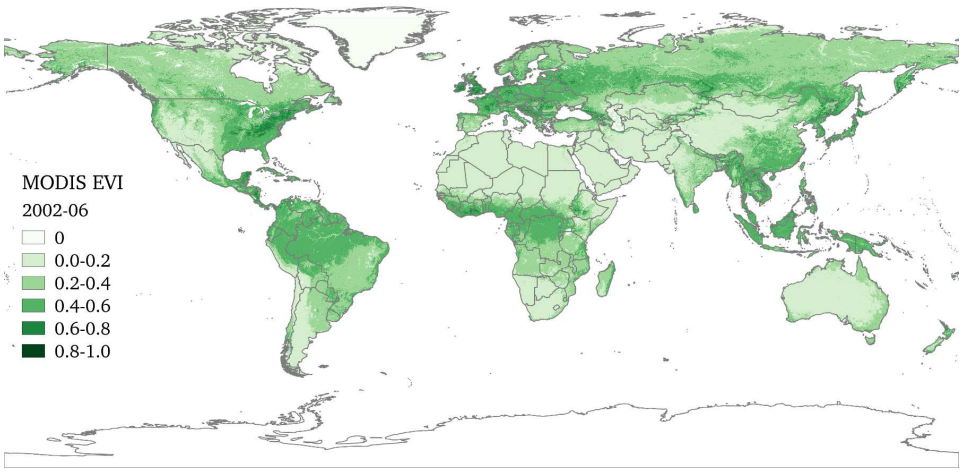


Figure 3: MODIS EVI at 0.5° resolution for the month of June 2002.

MODIS is a satellite-borne sensor that captures 36 spectral bands (ρ) of reflected/emitted energy from the Earth at various spatial resolutions. The MODIS sensor-array is onboard two National Aeronautics and Space Administration (NASA) Earth observation satellites: Aqua and Terra. These satellites have slightly different paths over the planet; however, the same spectral bands are collected by both satellites. Data is available from the Terra satellite starting in February 2000, while data from the Aqua satellite are available starting in July 2002. The moderate-resolution of this sensor is 0.05° (i.e., $1/10^{\text{th}}$ the 0.5° GePiSaT model resolution). Because MODIS data is at $1/10^{\text{th}}$ the model resolution, MODIS EVI products are resampled to 0.5° resolution to match the gridded data used in GePiSaT, see Appendix B for the resampling methodology. Figure 3 shows an example of monthly EVI at 0.5° resolution for June 2002.

3.4 VPD

It has long been known that evapotranspiration is affected by atmospheric vapor pressure, or the amount of moisture in the air. Vapor pressure deficit (VPD) is a common metric used to describe the driving force for evapotranspiration. VPD is defined as the difference between the saturation vapor pressure, e_s, which varies with ambient temperature, and the actual vapor pressure, e_d:

$$\mathrm{VPD} = e_s - e_d \tag{3}$$

In order to estimate VPD, the saturation vapor pressure can be calculated from the air temperature (Abtew and Melesse 2013, Eq. 5.1):

$$e_s = 0.611 \exp\left(\frac{17.27\, T_{air}}{T_{air} + 237.3}\right) \tag{4}$$

where:
e_s = saturation vapor pressure [kPa]
T_{air} = air temperature [°C]

The air temperature in Eq. 4, T_{air}, may reflect a variety of quantities, such as the 24-hour mean air temperature, the daily maximum air temperature, the daily minimum air temperature, or the average air temperature. The latter quantity is used in the calculation of VPD by means of CRU TS 3.21 monthly average daily temperature, tmp, which is the also the average of the monthly average maximum and minimum daily temperatures (tmx and tmn, respectively). The CRU TS 3.21 dataset also includes a measure of monthly vapor pressure, vap, in units of hectopascals (hPa). The full expression for calculating VPD based on CRU TS 3.21 datasets is then:

$$\mathrm{VPD} = 0.611 \exp\left(\frac{17.27\ \mathrm{tmp}}{0.5\ \mathrm{tmp} + 237.3}\right) - 0.10\ \mathrm{vap} \tag{5}$$

where:
VPD = monthly average vapor pressure deficit [kPa]
tmp = monthly average daily mean temperature [°C]
vap = monthly average vapor pressure [hPa]

The monthly VPD data calculated from Eq. 5 is stored in the GePiSaT database (see VPD in Table 1).

3.5 Tc, Pre and Elv

The CRU TS3.10 (currently at version 3.22) monthly 0.5°×0.5° resolution netCDF climatic data sets provide the mean daily air temperature and precipitation data (Harris et al. 2014), which is saved to the GePiSaT database (Tc and Pre in Table 1, respectively).

The $0.5° \times 0.5°$ ground surface elevation data is from the CRU TS3.0 data archives and saved in the GePiSaT database (see Elv in Table 1).

3.6 CO2

Mean annual atmospheric CO_2 concentrations are available from the Global Monitoring Division of the National Oceanic and Atmospheric Association (NOAA) / Earth System Research Laboratory (ESRL)[7].

The mean annual global data provides a representative quantification for the various flux towers located around the world without superposing the issue of dealing with seasonality in the data. The same annual value is therefore used for each month of a particular year.

The data are based on averages taken over marine surface sites and are given in units of parts-per-million (ppm). The annual atmospheric CO_2 concentration data are saved in the GePiSaT database (see CO2 in Table 1).

3.7 Alpha

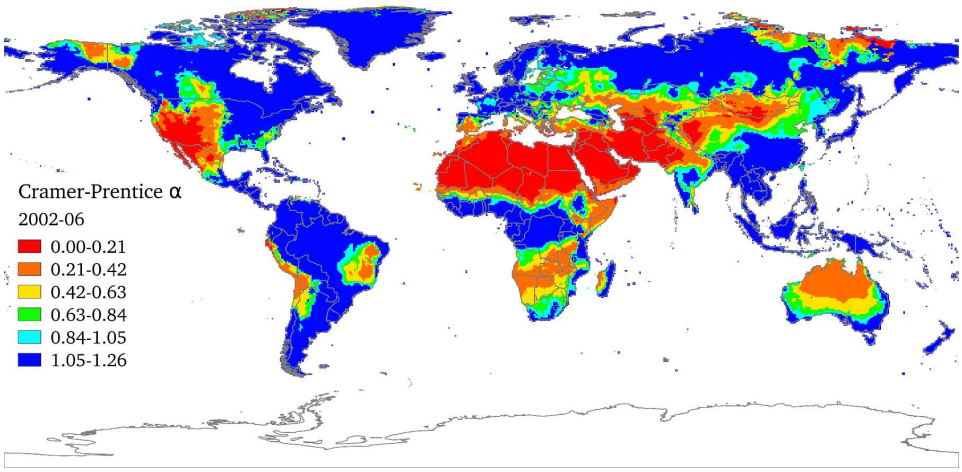


Figure 4: Cramer-Prentice $\alpha^\star$ at 0.5° resolution for June 2002.

In order to account for soil moisture affects on GPP, the Cramer-Prentice bioclimatic moisture index, $\alpha^\star$ (i.e., the ratio of monthly actual to equilibrium evapotranspiration), is incorporated into the GePiSaT model (Cramer and Prentice 1988; Gallego-Sala et al. 2010). To alleviate the burden of modeling soil moisture in GePiSaT, monthly estimates of $\alpha^\star$ are calculated *a priori* based on the STASH (STAtic SHell) model (Sykes and Prentice

[7] `http://www.esrl.noaa.gov/gmd/ccgg/trends/global.html`

1995, 1996; Sykes et al. 1996). Allowing for the fact that the actual evapotranspiration follows the potential values, $\alpha^\star$ is expected to range between 0 and 1.26 (Lhomme 1997; Priestley and Taylor 1972). The monthly $\alpha^\star$ calculated using the STASH code are saved in the GePiSaT database (see alpha in Table 1). Figure 4 shows an example of $\alpha^\star$, calculated from CRU TS climatology, distributed over the terrestrial landscape for June 2002.

4 Future Work

This model is being developed as a data-driven "diagnostic" model of terrestrial GPP; however, the intention is to link it with earth system and/or land-surface models to improve their representation of ecosystem production.

Part II
Database Documentation

5 Introduction

PostgreSQL[8] is a free and open source object-relational database management system available on a variety of operating systems (e.g., Linux, Windows, and Mac). The latest version available (as of the writing of this document) is version 9.2.4. While not currently used for this model, directions for installing the geographic object plugin, PostGIS[9], will be covered in this documentation.

[8] `http://www.postgresql.org`
[9] `http://postgis.net`

6 Installation

PostgreSQL can be installed on a variety of operating systems. A popular alternative to installing PostgreSQL directly onto your computer is to use a virtual computer which operates independent of your main computer's operating system and can be easily ported from one machine to another. BitNami[10] is one such provider of pre-packaged virtual computers with applications already configured for software development.

The database for this modeling system was tested on both a native PostgreSQL installation (on Linux Mint Maya) and on the BitNami Linux Application (LAPP) stack. The LAPP stack provides a complete PostgreSQL development environment which also includes the PostGIS extensions.

In order to run a virtual machine, virtualization software is required. There are numerous virtualization software providers, however, the most common are VirtualBox[11] and VMWare[12].

6.1 Virtual machine setup

The following instructions give a step by step overview on how to install the LAPP stack virtual machine on OSX.

1. Download and install VMware Fusion 6[13]. VMware fusion is not open source and a license purchase is required.

2. Download Bitnami LAPP Stack[14] (Virtual Machine, LAPP Stack on VMware) and extract the **.zip** file to your home directory.

3. Start VMware fusion:

   (a) Case 1, first time using VMware

      i. VMware will offer you to install a virtual machine through several ways. Choose:
      "More options..." → "Import an already existing virtual machine".

      ii. Browse to your home directory, where you have extracted the **.zip** file; the directory should be named:
      bitnami-lappstack-5.4.24-0-ubuntu-12.04

      iii. Select the **.ovf** file; it should be named:
      bitnami-lappstack-5.4.24-0-ubuntu-12.04-VBOX3.ovf

[10] `http://bitnami.com`
[11] `https://www.virtualbox.org`
[12] `http://www.vmware.com`
[13] `https://www.vmware.com/fr/products/fusion/`
[14] `http://bitnami.com/stack/lapp`

iv. Choose a place to install the virtual machine; the default location on OSX is: "Documents/Virtual Machines"

(b) Case 2, add a new virtual machine to an already existing VMware installation

i. In VMware, navigate to `File` → `New...`

ii. At the prompt for the installation method, follow the same procedure as in Case 1 above.

4. If you are using a **wifi connection**:

   (a) Once the BitNami is installed (you should get an installation report with confirmation of success, partition size, etc.), click on "Configuration."

   (b) Select "Network interface controller."

   (c) Tick "wifi" and make sure the "Network interface controller" is activated. You will be prompted for your **OSX** admin password.

   (d) Close the window and start BitNami.
   NOTE: If you forgot to activate the wifi as described above just after the installation process, BitNami configuration will fail on starting up. Run the command: `sudo poweroff` to power down BitNami. Go to `Virtual machine` → Configuration... → "Network interface controller" and repeat the procedure described above. Keyboard shortcut: **cmd + E**.

5. Run BitNami

6. On starting up BitNami, the following will displayed at the top of the screen, including the ip address:
   `*** You can access the application at http://XXX.XXX.X.XX ***`
   This address can later be accessed with the `ifconfig` command.

7. Sign in with the default username (`bitnami`) and password (`bitnami`) at the screen prompts.
   You will be prompted for changing the password following the first login.

6.2 Native Linux setup

The following instructions give a basic overview of installing PostgreSQL with PostGIS on a native Linux system.

1. Download and install the PostgreSQL core distribution for your operating system[15] or use the native application installer (e.g., apt-get) or package manager (e.g., Synaptic Package Manager)

[15] `http://www.postgresql.org/download`

2. Setup the password for the default postgres user.
   Note: for BitNami LAPP stack installations, the postgres user already has a default password (e.g., 'bitnami') and PostgreSQL can be accessed the first time using: `psql -U postgres -d postgres`

   (a) Log into PostgreSQL terminal:
   > `sudo -u postgres psql postgres`

   (b) Create a new password on the PostgreSQL terminal:
   # `\password postgres`

   (c) Enter and confirm the new password

   (d) Quit PostgreSQL:
   # `\q`

6.3 Resizing Linux for postgreSQL database

The database in this model can reach a disk size greater than the default disk size of the BitNami virtual machine (i.e., $\approx$ 17 GB). Additionally, some Linux OS disk partitions are not sized adequately for the database installed in the default location (e.g., /usr/lib/postgresql/9.1/main/data). Therefore, the database data directory needs to be relocated to a partition with adequate space to hold the observations for the model. The following instructions are for moving the default postgreSQL database data directory.

1. Create or identify a partition for holding the database data

   (a) BitNami

      i. While the virtual machine is turned off, expand the hard disk settings to a larger capacity, (e.g. navigate to: VMWare Fusion $\rightarrow$ Virtual Machine $\rightarrow$ Hard Disk (IDE) $\rightarrow$ Hard Disk (IDE) Settings ... and move the slider to allocate additional space)

      ii. Turn on the virtual machine

      iii. List the disk partitions:
      > `sudo fdisk -l`

      The partitioning table will be displayed, providing a breakdown of device names, start and end positions, number of blocks (e.g., size of each device), partition id and associated system name. At the top of the list, the total disk size should be displayed corresponding to the size selected previously (e.g., /dev/sda: 85.9 GB, 167772160 sectors, sector = 512 bytes). Note that the blocks listed in the partition table correspond the the sectors. For example, the current table may

only have two devices: /dev/sda1 and /dev/sda2 corresponding to a Linux and Linux swap system, respectively. The end position of the second device (i.e., /dev/sda2) should be less than the number of sectors listed for the drive (e.g., end = 33998847).

iv. Determine the main partition of your LAPP stack based on the results from the previous step (e.g., /dev/sda)

v. Format the disk, e.g.:
`> sudo fdisk /dev/sda`

vi. Create new partition for holding the database by typing the following keystrokes in `fdisk`:
`p`: prints the partition table
`n`: creates a new partition
`p`: sets new partition type to "primary"
`3`: defines partition number
`<default>`: sets the start position to default
`<default>`: sets the end position to default

Note: the default start and end positions may fall in a small gap located between /dev/sda1 and /dev/sda2; if this is the case, create another primary partition:
`n`: creates new partition
`p`: sets partition type to "primary"
`4`: defines partition number
`<default>`: sets the start position to default
`<default>`: sets the end position to default
`w`: writes partition table

vii. Restart the virtual machine:
`> sudo poweroff`

viii. Make new partition into a file system (Note: in this example /dev/sda4 is the new partition with start position 33998848 and end position 167772159, e.g., 85.9 GB):
`> sudo mkfs -t ext3 /dev/sda4`

ix. Create a mounting point for this partition, e.g.:
`> sudo mkdir /database`

x. Mount new partition to a directory.
Open the fstab document:
`> sudo pico /etc/fstab`

Add the following line to the end of the document and save:
`/dev/sda4 /database ext3 defaults 0 0`

(b) Native Linux

i. Create a directory for holding the database on a partition with adequate space available (e.g., >80 GB). On most Linux machines, the /home directory is on a large partition.
> `mkdir ~/Database`

2. Create a data directory in the database directory created in the previous step (e.g., for BitNami):
> `sudo mkdir /database/data`

3. Change directory owner to user "postgres" (e.g., for BitNami):
> `sudo chown postgres:postgres /database/data`

4. Stop postgresql service (e.g., BitNami):
> `sudo /opt/bitnami/ctrlscript.sh stop`

5. Initialize new data directory (e.g., BitNami):
> `su - postgres -c '/opt/bitnami/postgresql/bin/initdb -D /database/data'`

   Note: a request will be made for postgres user's password, which may not be set yet. If it is not set (i.e., the above command fails for postgres password), first create a password for postgres and then try running the command again:
> `sudo passwd postgres`

6. Edit postgresql start up script (e.g. BitNami):
   Backup the `ctl` script with the following command:
> `sudo cp /opt/bitnami/postgresql/scripts/ctl.sh ... /opt/bitnami/postgresql/scripts/ctl.sh.bak`

   Open the `ctl` script:
> `sudo pico /opt/bitnami/postgresql/scripts/ctl.sh`

   Edit the start-up commands (i.e., the two lines that start with "POSTGRESQL_START" and "POSTGRESQL_STOP") by changing the directory path following the "-D" with the new database data data directory (e.g., "/database/data")

7. Start postgresql service (e.g., BitNami):
> `sudo /opt/bitnami/ctrlscript.sh start`

8. Check that postgresql is running correctly:
> `ps auxw | grep postgresql | grep -- -D`

   Note the directory listed following the -D. If it is the right one (e.g., /database/data), proceed.

9. Create postgres password:
   Log into database:
   > `psql -U postgres -d postgres`

   Set password:
   # `\password postgres`

   Exit psql:
   # `\q`

10. Stop postgresql service (e.g. BitNami):
    > `sudo /opt/bitnami/ctrlscript.sh stop`

11. Edit pg_hba.conf file in the newly established data directory (e.g. BitNami):
    > `su - postgres -c 'pico /database/data/pg_hba.conf'`

    Change all the instances of "trust" to "md5"

12. Follow setup instructions (§7) to create the database

Note that in the old database can still be accessed by stopping the postgresql service, replacing the start-up script (e.g., the "ctl.sh" script located in /opt/bitnami/postgresql/scripts directory) with the original (i.e., "ctl.sh.bak") and restarting the service.

7 Setup

In this section, a new PostgreSQL user and database are created. The following steps are to be performed on either a native installation or within a virtual stack.

Creating a new user allows for individualism in project development as well as added security since the user will have its own password. The new user should be given the privilege for creating databases (i.e., `CREATEDB`) and (optionally) the privilege to create other users (`CREATEUSER`). It is not necessary to create a new user.

The new database is created based on the PostGIS template. BitNami LAPP stack installations require first that the PostGIS template be created. For native PostgreSQL and PostGIS installations, the PostGIS template should already be made available. In addition to the PostGIS extensions, the database utilizes pivot tables as a means of querying and retrieving data. To allow access to pivot tables, the tablefunc extension must be installed on the database. BitNami LAPP stack installations include the `tablefunc` extension. In the event that the `tablefunc` extension is not available, it is included in the additional facilities for postgreSQL package (e.g., `postgresql-contrib-9.1` under the Synaptic Package Manager in Linux). Available PostgreSQL extensions can be viewed by querying the pg_available_extensions table in the database.

1. Create a new user (*optional*)

   (a) Log into the default PostgreSQL database:
   > `psql -h 127.0.0.1 -p 5432 -U postgres -d postgres`

   (b) On the PostgreSQL terminal line, create the new user account:
   # `CREATE USER user PASSWORD '*' CREATEDB CREATEUSER;`

   NOTE: Replace `user` with your name and `'*'` with your own password.

   (c) Quit PostgreSQL:
   # `\q`

   (d) Log into PostgreSQL as the newly created user:
   > `psql -h 127.0.0.1 -p 5432 -U user -d postgres`

2. Create a PostGIS template
   Note: for native PostGIS installations (i.e., not a LAPP stack installation) the template_postgis is already created

   (a) From the PostgreSQL terminal, create a new database:
   # `CREATE DATABASE template_postgis;`

(b) Allow non superusers to create a database from this template:
```
# UPDATE pg_database
SET datistemplate='true'
WHERE datname='template_postgis';
```

Note: a response of "UPDATE 1" indicates success

(c) Load PostGIS SQL routines to the newly created database

i. Log out back into PostgreSQL to the newly created template_postgis database as user postgres:
```
> psql -h 127.0.0.1 -p 5432 -U postgres ...
-d template_postgis
```

ii. Install PostGIS extensions to the template:
```
# CREATE EXTENSION postgis;
# CREATE EXTENSION postgis_topology;
```

iii. Check that the extensions were installed correctly:
```
# SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name LIKE 'postgis%';
```

Note: You should get a result showing postgis v. 2.0.1 and postgis_topology v. 2.0.1 or similar

iv. Enable users to alter the spatial tables:
```
# GRANT ALL ON geometry_columns TO PUBLIC;
# GRANT ALL ON geography_columns TO PUBLIC;
# GRANT ALL ON spatial_ref_sys TO PUBLIC;
```

v. Log out of PostgreSQL:
```
# \q;
```

vi. Check that everything is installed correctly:
```
> sudo -u postgres psql template_postgis -c ...
''SELECT PostGIS_Full_Version();''
```

vii. Hit "q" on your keyboard to exit the results screen

3. Create a new database based on PostGIS template:

(a) Log into the default PostgreSQL database as user postgres

(b) From the PostgreSQL terminal, create a new database:
```
# CREATE DATABASE gepisat
WITH OWNER=user
TEMPLATE=template_postgis;
```

Note: Use your user name for the owner or set owner to 'postgres' is you chose not to create a new user

(c) Log out of PostgreSQL

4. Add extention to database to allow pivot tables:

   (a) Log into the newly created database:
   > `psql -h 127.0.0.1 -p 5432 -d gepisat -U user`

   (b) Add tablefunc extension:
   # `CREATE EXTENSION tablefunc;`

An empty gepisat database is now ready for use. The next step is to create the three database tables and populate them with data.

8 Structure

The database structure is based on three tables. The first table stores the meta data regarding a particular data source. This includes individual flux towers, meteorological stations, and satellite pixels. The meta data includes such information as geolocation, climate, and vegetation details. The second table stores information regarding the data types (e.g., eco-climatological variables) available for each station or grid point. Information regarding the data variables includes the variable name and units of measure. The third and last table stores the actual observation data.

8.1 Meta-data table

8.1.1 Field descriptions

The *mapid* and *map* fields are included in the meta data as a means for quickly referencing all data within a large region. The current convention assigns a continent (North America, South America, Europe, Africa, Asia, Australia, or Antarctica) to each station location. For instances where stations are located on an island far removed from a continent, they are instead assigned to an ocean (North Atlantic, South Atlantic, North Pacific, South Pacific, Indian, Arctic, Southern).

For most stations, the *lon*, *lat*, and *ele* fields are easily taken from the station's website or source. Often the country where the station resides is also given (e.g., the *country* field). However, to get the *countryid* field, ISO alpha-2 references are available online.

Station identifiers (i.e., the *stationid* field) may be given by their source provider (e.g., Fluxdata.org gives each flux station a six-character identifier) or they may be assigned by the user. For gridded data (e.g., satellite observations), each pixel is considered an independent station such that each pixel can have its own set of variables each with its own set of observation data. Due to difficulty in dealing with gridded data of differing resolutions, gridded stations are unique for a single resolution only. The major grid resolution used in this project consists of 0.5° resolution pixels (i.e., 5600 km squares). At this resolution, there are 720 pixels spanning across longitude values -180° to 180° and 360 pixels spanning across latitude values -90° to 90°. Therefore, gridded data at 0.5° resolution consists of 259200 pixels. For the purposes of station numbering, the pixel located at in the bottom left-hand corner (i.e., farthest south-west) at coordinates -180°, -90° is station number 0 and the pixel located at the top right-hand corner (i.e., farthest north-east) at coordinates 180°, 90° is station number 259199. All pixels between these two locations are numbered sequentially in row-major fashion. A prefix of 'HDG' is given to each pixel number to represent the half-degree grid.

Table 5: Description of meta data table postgreSQL database fields.

Name	Description
mapid	Identifier for the part of the world (e.g., continents)
map	Name of continent, ocean, etc.
countryid	ISO alpha-2 country abbreviation
country	Country name
stationid	Unique ID for flux towers, met. stations, etc.
station	Station name
lon	Longitude (DD) of station (or pixel centroid)
lat	Latitude (DD) of station (or pixel centroid)
ele	Approximate ground elevation (m)
classid	IGBP land cover type abbreviation
class	IGBP land cover type name
climateid	Köppen climate classification abbreviation
climate	Köppen climate classification name
data_years	List of years where data is available
years_data	Number of years of data available
network	Data network (e.g., AmeriFlux)
url	Web address for data or tower
created	Date data was created (not implemented)
uploaded	Date data was uploaded to database (not implemented)
geom	Geometry of data (point or grid)
coor	Coordinate projection system (for lon / lat)
dim	Dimension of the data (e.g., gridded data = 2)
res	Resolution of the data, e.g., pixel size

The *station* field is meant to allow for a longer non-unique name to describe each station.

The *classid*, *class*, *climateid*, and *climate* fields are included to allow for quick referencing data that exist within similar environments (i.e., not necessarily within the same geographic region).

Two fields are provided to describe the amount of data and associated time periods available (e.g., *data_years* and *years_data* fields). The *years_data* field is assumed to be an integer; however, it should be noted that data may not be available for the entire year listed.

The *network* and *url* fields are provided to indicate the origins of the data presented in this database. Networks may be institutions, organizations, or other affiliations. To help ensure that data is properly sourced, a website can accompany the network.

Because the data that is used in this study is subject to periodic revisionary updates, two additional fields have been recommended to help maintain fidelity between data revisions (though they are yet to be implemented).

The first field is *created* which indicates the date that the data was created. This is typically published with the data by its source. The second field is *uploaded* which indicates the date that the data was uploaded to the database.

The final fields (i.e., *geom*, *coor*, *dim*, and *res*) provide additional information regarding the shape of the data (e.g., point or gridded observation), the coordinate system (for referencing the *lon* and *lat* fields), the dimension of the data, and the resolution of the data (specifically for defining the pixel size of gridded data).

Table 5 provides a description for each of the column headers used in the meta data for the postgreSQL database.

8.1.2 Meta-data SQL table creation command

To create the meta-data table, the following SQL query may be used. The *stationid*, *lat*, *lon*, *years*, *created*, *uploaded*, and *dim* fields are listed as `NOT NULL` and therefore must be included in the database. The *stationid* is the primary key for this table, meaning it must be a unique value for each entry (i.e., no duplicate stations or pixel names are allowed).

```
CREATE TABLE met_data (
   mapid character(3),
   map text,
   countryid character(2),
   country text,
   stationid varchar(12) NOT NULL,
   station text,
   lat real NOT NULL,
   lon real NOT NULL,
   ele real,
   classid character(3),
   class text,
   climateid varchar(4),
   climate text,
   data_years text,
   years_data integer NOT NULL DEFAULT 0,
   network text,
   url text,
   created timestamp NOT NULL,
   uploaded timestamp NOT NULL,
   geom text,
   coor text,
   dim integer NOT NULL,
   res text,
```

```
   CONSTRAINT set_pk1 PRIMARY KEY (stationid)
)
WITH (
   OIDS=FALSE
);
```

8.1.3 Acquiring data

It is of practical use to know where to obtain the information for the meta data table. This section presents some resources and methods for getting the meta data.

Maps and countries The idea behind this hierarchy of spatial data labelling was to make it intuitive as to where in the world each data in the database represents. The first and top-most hierarchal level is the *mapid* and *map* fields. The *map* field is a broad generalisation of global regions based on the least specific classification: continents and oceans. Unfortunately there is no easy link between local land-based point measurements and the continent; however, there are references linking countries with their respective continent. For example, the Gazetteer of Planetary Nomenclature[16] provides a comprehensive list of countries (including ethnic/cultural groups) by continent with their own nomenclature. For the purposes of this project, the International Organization for Standardization (ISO) country codes (i.e., two-letter ISO-3166[17]) are used to identify the countries of the world.

A Python script (i.e., `map_country.py`) was written to quickly map the *country* and *countryid* fields to return all four meta data fields (i.e., *mapid*, *map*, *countryid*, and *country*).

For the purposes of getting the country names, most data are supplied with coordinates (i.e., longitude and latitude) to accompany the measurements. If not, it is only a matter of assigning an approximate location based on mapping software (e.g., Google Maps[18]). After geolocation information is acquired, country information can be extracted from electronic maps.

Geographic information system (GIS) software is available for mapping and overlaying geographic information. One such open-source software is GRASS GIS[19]. GRASS is available for all major computer operating systems (i.e., Windows, Mac OSX, and Linux) and supports a variety of geographic file formats. For the purposes of data extraction, instructions will assume the use of GRASS GIS.

[16] http://planetarynames.wr.usgs.gov/Abbreviations
[17] http://www.iso.org/iso/country_codes
[18] https://maps.google.co.uk
[19] http://grass.osgeo.org

Shapefiles (or vector-based boundary maps) can be downloaded from a variety of sources which map out various geographic information. For example, open data analysis and maps are available from Geocommons.com for 2002 world country boundaries[20] and world country administrative boundaries[21]. After downloading a shapefile from the internet, extract the compressed information (e.g., unzip) and examine the contents of the folder. There should be a set of files all with the same name but with different extensions (e.g., .dbf, .shp, .shx, etc.). Keep all these files together.

In GRASS GIS, following the instructions online to setup a project mapset[22]. Import the shapefile to the map (`File` → `Import vector data` → `Common import formats`). The point locations need to be in a column format ASCII text file (e.g., .txt or .csv file extension). A simple method for this is to use a spreadsheet software (e.g., MS Office Excel, OpenOffice Calc, etc.) and save the document as a "CSV format." The only information required to add points from a text file to the map are the longitude and latitude values; separate these value pairs into their own columns. Add the points to the map (`File` → `Import vector data` → `ASCII points/GRASS ASCII vector import`). Once the boundary map and point locations are loaded, extract the country information from the boundary file to the points based on the proximity of the points to each country's region. GRASS GIS has the vector function `v.distance` to calculate the distance between vector features (i.e., points, lines, boundaries, areas, etc.). The `v.distance` function can also return the feature from one layer that is closest to a feature in another layer. First open the attribute table of the point data layer (`right-click layer name in GRASS GIS Layer Manager` → `Show attribute data`). Create a new attribute in the point data layer to store the country data to.

8.2 Variable list table

8.2.1 Description

The variables table organizes information regarding each station's observation types. To join the table of station variables with the station's meta data, it is necessary to maintain the *stationid* field. Each eco-climatological variable included in this database (e.g., net ecosystem exchange, shortwave down-dwelling radiation, etc.) is given a unique identification number represented by the *varid* field. Therefore, to create a unique identifier for a specific station's observations, a new field is created, called *msvidx*. The *msvidx* field is a combination of the *stationid* and *varid* fields and is used to match a station with its variable observations.

[20] http://geocommons.com/overlays/5603
[21] http://geocommons.com/overlays/33578
[22] http://grass.osgeo.org/documentation/first-time-users/

Table 6: Description of variable list table postgreSQL database fields.

Name	Description
stationid	Identifier for individual flux towers, met. stations, etc.
varid	Identifier for measurement variable
msvidx	Identifier for a specific station's variable
varname	Variable name or description
varunit	Units of measure
vartype	Flux tower, met. station, or gridded data product
varcore	Boolean for core or non-core variables

The *varname* field is included to provide a description for each *varid*. The *varunit* describes the units of measure that accompany the variable (e.g., days, $m \cdot s^{-1}$, or $W \cdot m^{-2}$). The *vartype* field indicates the general data source (e.g., flux tower, meteorological station, or remote satellite). Lastly, the *varcore* field is included to indicate whether a variable is core (i.e., available across all other stations of the same *vartype*) or secondary (i.e, not necessarily available at other stations).

Table 6 provides a description for each of the column headers used in the variables list for the postgreSQL database.

8.2.2 Variable list SQL table creation command

To create the variables table, the following SQL query may be used. The *msvidx*, *varid*, and *vartype* fields are listed as `NOT NULL`. The *msvidx* field is this table's primary key. The `UNIQUE` command on the field pairs *stationid* and *varid* maintains the unique status of the *msvidx* field (which is the combination of the *stationid* and *varid* fields). Object IDs (i.e., `OIDS`) are not necessary because this table has a primary key. The field *stationid* is referenced to the same field in the meta data table. Therefore, the meta data table must be created and populated before this table is populated.

```
CREATE TABLE var_list (
   msvidx varchar(15) NOT NULL,
   stationid varchar(12) REFERENCES met_data(stationid),
   varid integer NOT NULL,
   varname text,
   varunit text,
   vartype character(4) NOT NULL,
   varcore integer,
   UNIQUE(stationid, varid)
   CONSTRAINT set_pk2 PRIMARY KEY (msvidx)
)
WITH (
```

```
   OIDS = FALSE
);
```

8.3 Data set table

8.3.1 Description

The data table is the simplest table with only four fields. The first two fields are *msvidx* and *stationid* which provides a link connecting the data set to both the variable list and meta data.

The *datetime* field provides a timestamp for each observation. The *data* field provides the associated measurement at each timestamp.

Table 7 provides a description for each of the column headers used in the data set database table.

Table 7: Description of data set table postgreSQL database fields.

Name	Description
msvidx	Identifier for a specific station's variable
datetime	Universal timestamp value (%Y-%m-%d %H:%M:%S)
data	Station's variable observation in time

8.3.2 Data set SQL table creation command

To create the data table, the following SQL command may be used. The *msvidx* field references the same field in the variables table. Therefore, the variables table must be created and populated before this table is populated. The *stationid* field references the same field in the meta data table. There can only be a single observation for each station's variable; hence, there is a `UNIQUE` condition placed on *msvidx* and *datetime* fields. This reduces the risk of having simultaneous measurements of the same observation giving different values.

```
CREATE TABLE data_set (
   msvidx varchar(15) REFERENCES var_list(msvidx),
   stationid varchar(12) REFERENCES met_data(stationid),
   datetime timestamp,
   data float,
   UNIQUE(msvidx, datetime)
)
WITH (
   OIDS = FALSE
);
```

Part III
Model Documentation

9 Environment Setup

The core of the modeling is performed using open-source programming languages (e.g., Python and R). In order to facilitate the model, specific configurations were made to the working environments which these programs operate under. These configurations often accompany having specific library files installed within the working environment that the model is executed under. This section overviews the necessary working environment configurations to operate the model code described in this reference.

9.1 Python environment

The most efficient method of quickly gaining access to all the necessary Python libraries and modules (except psycopg2) is to download an interactive development environment (IDE). Two notable Python IDEs are the Enthought Python Distribution[23] (now called "Canopy") and Google spyder[224] (previously called "Pydee"). Both Canopy and spyder provide a comprehensive Python developing and programming environment for scientific analysis and visualization. There is a free version of Canopy as well as a free one-year academic license for registered users with a school or university e-mail address. Spyder is open-source software available for Windows, Mac OSX, and Linux operating systems; however it has not been used or tested. This text assumes the use of the Enthought Canopy IDE. Table 8 presents a list of Python libraries that are used in this model and the model files in which they are used.

To overcome the lack of native compatibility between Canopy and the psycopg2 module, psycopg2 can be installed as "second party" software by means of pip (Python package manager). In a native Linux system and using the Canopy Python environment as the default Python environment, run: `pip install psycopg2` on the command line. Note that for use of Canopy IDE outside a virtual machine where the GePiSaT database is installed may not be possible (although untried as of this writing). The main advantage of the Canopy IDE is the easy of installing the HDF4 libraries (for reading the MODIS EVI data), which is rather complicated to successfully achieve (in the writer's opinion). For all other circumstances, Canopy may be used separately for processing HDF4 data (via table_maker.py), whilst all other Python programs may be run from a separate Python environment.

9.2 R environment

Similar to the Python, the R programming language benefits from having an open source IDE for programming and development. One such IDE is

[23] https://www.enthought.com/products/canopy/
[24] http://code.google.com/p/spyderlib/

Table 8: List of Python libraries used in this model.

Library	**Description**	**[1]**	**[2]**	**[3]**
datetime	Timestamp handling	x		x
glob	File searching		x	x
numpy	Numerical arrays and operators	x		x
os.path	Directory and file handling	x	x	x
psycopg2	PostgreSQL database management	x	x	
pyhdf	HDF file access (SD)			x
re	Regular expression support			x
scipy.io	netCDF file access			x
scipy.optimize	Non-linear least squares (curve fit)	x		
scipy.special	Error function (erf)	x		
scipy.stats	Student's t-test	x		
sys	Model run termination	x	x	

Included in [1] model.py, [2] db_setup.py, or [3] table_maker.py

RStudio[25].

[25] http://www.rstudio.com/ide/

10 Next-Generation Model Development

At the heart of GePiSaT is a production efficiency or "diagnostic" model for estimating monthly GPP. Similar to most other production efficiency models, GPP is expressed as a function of absorbed light, I_{abs}. The basic equation for monthly GPP takes the form:

$$\text{GPP} = \varepsilon \, I_{abs} \tag{6}$$

where:

GPP = gross primary production [mol C m^{-2} mo^{-1}];
I_{abs} = absorbed photosynthetic light; [mol photons m^{-2} mo^{-1}];
ε = light-use efficiency [mol C mol $photons^{-1}$].

The light-use efficiency, ε, is therefore defined as the ratio of GPP to absorbed light.

The absorbed photosynthetic light is defined as the fraction of absorbed photosynthetically active radiation (PAR) and is calculated as the product of fPAR and PPFD (i.e., PAR in units of quanta).

Despite the simplicity of Eq. 6, this basic model performs rather well at a few of the flux tower sites. However, in the effort of building a universal model, Eq. 6 does not capture the global dynamics of GPP; therefore, a next-generation model is proposed based, in part, on the work of Wang et al. 2012:

$$\text{GPP} = \phi_{\circ} \, \text{f}_{\alpha} \, I_{abs} \, \sqrt{m^2 - c^{2/3} \, m^{4/3}} \tag{7}$$

where:

$\phi_{\circ}$ = intrinsic quantum efficiency [mol C mol $photons^{-1}$];
f_{α} = index of plant available moisture;
I_{abs} = absorbed photosynthetic light [mol photons m^{-2} mo^{-1}];
c = maintenance cost of light-harvesting capacity;
m = substrate limitation term.

A Python Code Snippets:

A.1 peirce_dev.py

```
01  import numpy
02  import scipy.special
03
04  def peirce_crit(N, n, m):
05    N = float(N)
06    n = float(n)
07    m = float(m)
08    if N > 1:
09       Q = (n**(n/N)*(N-n)**((N-n)/N))/N
10       Rnew = 1.0
11       Rold = 0.0
12       while ( abs(Rnew-Rold) > (N*2.0e-16) ):
13          ldiv = Rnew**n
14          if ldiv == 0:
15             ldiv = 1.0e6
16          Lamda = ((Q**N)/(ldiv))**(1.0/(N-n))
17          x2 = 1.0 + (N-m-n)/n * (1.0-Lamda**2.0)
18          if x2 < 0:
19             x2 = 0
20             Rnew = Rold
21          else:
22             Rold = Rnew
23             Rnew = numpy.exp((x2-1)/2.0)
24                * scipy.special.erfc(
25                   numpy.sqrt(x2)/numpy.sqrt(2.0)
26                )
27    else:
28       x2 = 0.0
29    return x2
```

Lines 1–2 import the necessary module libraries for performing the calculations. Lines 4–29 represent the function block for calculating Peirce's threshold error (i.e., x^2). Lines 5–7 cast the float data type to the input data. This is done to allow short hand (i.e., integer) values to be sent to the function. It is necessary for these values to be float-type to avoid integer division (i.e., whole number division). Line 8 is a check to make certain there is enough data for processing. Line 9 is the calculation of Q (step 1). Line 10 is the initial guess for the value of R (step 2). Line 11 initializes the old guess for the value of R and is necessary to prompt the while loop

(i.e., lines 12–26). Line 12 is the declaration of the while-loop criteria (i.e., convergence criteria for R). Line 13 calculates the denominator for a part of the λ calculation. Line 14 checks the denominator to see if it is zero. If the denominator is zero, it is replaced with a 0.000001 to alleviate divide by zero issues (Line 15). Line 16 is the calculation of λ (step 3). Line 17 is the calculation of x^2 (step 4). Line 18 checks to see if the value of x^2 has gone negative. If it has, its value is set to zero and R is updated. Line 22 updates the previous guess for the value of R before it is updated if x^2 is positive. Lines 23–26 calculate the new value of R (step 5). Lines 27 and 28 set the value of x^2 to zero if there is not enough data for processing. When the convergence criteria is met for the while loop (i.e., line 12) the value for x^2 is returned.

A.2 outlier.py

```
01  import numpy
02  from scipy.optimize import curve_fit
03
04  (nee, ppfd) = monthly_pairs()
05  if (len(ppfd) > 3 and len(nee) > 3):
06    (opt, cov) = curve_fit(
07       model_h, ppfd, nee, estimates 08       )
09    nee_fit = model_h(ppfd, opt)
10    se = (nee - nee_fit)**2.0
11    sse = sum(se)
12    mse = float(sse)/(len(nee) - 3.0)
13    x2 = peirce_crit(len(nee), 1, 3)
14    d2 = mse*x2
15    n_index = numpy.where(se > d2)[0]
16    n_found = len(outliers_index)
17    if n_found == 0:
18       x2 = peirce_crit(len(nee), 2, 3)
19       d2 = mse*x2
20       n_index = numpy.where(se > d2)[0]
21       n_found = len(outliers_index)
22    n = 1
23    while (n <= n_found):
24       n += 1
25       x2 = peirce_crit(len(nee), n, 3)
26       d2 = mse*x2
27       n_index = numpy.where(se > d2)[0]
28       n_found = len(outliers_index)
29    ppfd_ro = numpy.delete(ppfd, n_index)
30    nee_ro = numpy.delete(nee, n_index)
```

Lines 1–2 load the necessary Python modules. Line 4 represents a function call to retrieve one month's NEE and PPFD pairs from a particular flux tower. Line 5 checks to make certain that enough data is available to perform the regression. Lines 6–8 calls the SciPy curve_fit function which returns the optimization parameters for fitting the NEE and PPFD observations to model_h (i.e., **??**) given a set of initial parameter estimates. Line 9 takes the optimization parameters found by the curve_fit function to get the model predictions of NEE (i.e., u in equation **??**). Lines 10–12 calculate the squared-error (SE), the sum of the squared-error (SSE), and the mean squared-error (MSE). Line 13 calculates Peirce's deviation (x^2) while Line 14 calculates the threshold squared error (Δ^2). Lines 15–16 identify and count

the instances of where model squared-error exceeds the threshold deviation. Lines 17–21 performs a secondary check for outliers in the case where no are initially found. Lines 22–28 perform an iterative search for additional outliers by incrementing Peirce's n parameter until the number of outliers identified is less than the number of outliers assumed (i.e., Line 23). Lines 29–30 remove the outliers from the original datasets.

A.3 netcdf.py

```
01  import numpy
02  from scipy.io import netcdf
03
04  doc = path + filename
05  (thisyear, thismonth) = get_month(doc)
06  f = netcdf.NetCDFFile(doc, "r")
07  voi = 'SWdown'
08  sh_day, sh_lat, sh_lon = f.variables[voi].shape
09  for y in xrange(sh_lat):
10    pxl_lat = f.variables['lat'].data[y]
11    for x in xrange(sh_lon):
12       stationid = 720*y + x
13       for t in xrange(sh_day):
14          thisday = t+1
15          timestamp = datetime.date(
16             thisyear, thismonth, thisday
17             )
18          pxl_val = f.variables[voi].data[t,y,x]
19          if pxl_val < 1.0e6 and pxl_lat > -60:
20             data = process_watch(voi, pxl_val, timestamp)
21  f.close()
```

Lines 1–2 load the necessary Python modules. Line 4 defines the netCDF file in terms of its path and file name. Line 5 represents an assignment of the current file's associated year and month. In most cases, the year and month can be read directly from the file name; however, there are other alternatives to this. Line 6 opens the netCDFfile for reading. Line 7 defines the variable of interest, in this case it is shortwave downwelling solar radiation (SW_{down}). Line 8 saves the shape of SW_{down} in terms of the number of days and the number of pixels along the latitude and longitude. Line 9 begins iterating through the latitude pixels while Line 10 saves the current latitude value (in decimal degrees). Line 11 begins the iteration through the longitude. Line 12 calculates the *stationid* parameter (as defined in section 8.1.1). Line 13 starts the iteration through the days while Line 14 saves the current month's day. Lines 15–17 create a `datetime.date` object based on the three fields which make up the current day. Line 18 reads the SW_{down} value for the given day, latitude, and longitude. Line 19 filters erroneous values (i.e., $SW_{down} \geq 10^6$) and observations from Antarctica (i.e., $lat \leq -60°$). Line 20 represents the data processing of the valid SW_{down} observations. Line 21 closes the netCDF file.

A.4 hdf.py

```
01  import numpy
02  from pyhdf import SD
03
04  doc = path + filename
05  ts = get_modis_ts(doc)
06  f = SD.SD(doc)
07  voi = ''CMG 0.05 Deg Monthly EVI''
08  f_select = f.select(voi)
09  f_data = f_select.get()
10  f.end()
11  (sh_lat, sh_lon) = data.shape
12  for y in xrange(sh_lat):
13    for x in xrange(sh_lon):
14       zval = data[y][x]
15       evi = zval / 10000.0
```

Lines 1–2 load the necessary Python modules. Note that `pyhdf` is not a simple installation for native Python environments. It is recommended to use a third-party Python developing and programming environment (see Section 9.1). Line 4 defines the HDF file in terms of its path and filename. Line 5 represents the assignment of a timestamp (i.e., a `datetime` object) based on the HDF filename. Line 6 opens the HDF file for reading. Line 7 defines the variable of interest, in this case it is "CMG 0.5 Deg Monthly EVI" corresponding to MODIS 0.5° degree resolution EVI monthly product. Line 8 selects the variable of interest from the HDF file. Line 9 retrieves the associated data (as a numpy array). Line 10 closes the HDF file. Line 11 saves the array shape of the data (in terms of the number of pixels along latitude and longitude). Lines 12–13 iterate through each pixel (in terms of x and y coordinates). Line 14 reads the pixel value associated with the x-y coordinate (EVI × 10,000). Line 15 converts the pixel value to EVI (ranges from -0.2–1.0).

B Resampling MODIS Data:

To upscale MODIS 0.05° resolution data to model-defined 0.5° resolution, there are two methods which can be employed. The first method is considered the long-hand method where each MODIS raster image is resampled through the Quantum GIS (QGIS) software[26]. The second method, considered the quicker automated method, utilizes Python to resample the MODIS data directly from the HDF files.

B.1 The QGIS method

The first step in processing MODIS data in QGIS is exporting the variable of interest from the HDF file to ASCII raster file format. This is accomplished in Python.

1. Import ASCII raster file in QGIS
2. Create 0.5° fishnet:
   (a) Vector → Research Tools → Vector Grid
   (b) Set extents for longitude, xmin: -180.0; xmax: 179.5
   (c) Set extents for latitude, ymin: -89.5; ymax: 90.0
   (d) Set grid cell size, x: 0.5; y: 0.5
   (e) Save the output grid as polygons
   (f) Add new fields in shapefile attribute table:
      i. Output field: "LAT"
      Type: Real (width: 6; precision: 2)
      Expression: `0.5 * (''YMAX'' + ''YMIN'')`
      ii. Output field: "LON"
      Type: Real (width: 7; precision: 2)
      Expression: `0.5 * (''XMAX'' + ''XMIN'')`
      iii. Output field: "STATION"[27]
      Type: Whole number (width: 8)
      Expression: `720.0 * (359.0 - ((90.0 - ''LAT'')/0.5 - 0.5)) + ((''LON'' + 180)/0.5 - 0.5)`
3. Add fishnet to map
4. Polygonize MODIS raster:
   (a) Vector → Conversion → Polygonize

[26] `http://qgis.org`

[27] The "STATION" field is calculated based on the conversion of the numbering index of the MODIS 0.05° to that of the WATCH WFDEI 0.5° grid. See equations 8–10.

(b) Input: MODIS raster layer

(c) Output: Polygon shapefile layer

5. Intersect fishnet and polygonized MODIS layers:

   (a) Vector → Geoprocessing Tools → Intersect

   (b) Input: Fishnet layer

   (c) Intersect: Polygonized MODIS layer

   (d) Output: Intersected polygon shapefile layer

6. Calculate stats:

   (a) Load plugin: Group stats

   (b) Plugins → Group Stats → Group Stats

   (c) Vector: Intersected layer

   (d) Classification: "STATION" field

   (e) Value: "DN" field

   (f) Calculate statistics and save to CSV file

7. Load statistics CSV file as a table (via Vector import)

8. Join table attributes to fishnet layer

   (a) Join: "STATION" field

   (b) Target: "STATION" field

9. Save fishnet layer as a new shapefile

10. Create a new field in attribute table:

    (a) Output field: "EVIZ"
    Type: Whole number (width: 5)
    Expression: `CASE WHEN ‘‘Average’’ IS NULL THEN -3000 ELSE toint(‘‘Average’’) END`

11. Rasterize polygon layer (based on "EVIZ" field)

It should be noted that the MODIS pixels are indexed starting at zero at the top left corner (north-west most pixel) and are row-major ordered ending with the last pixel in the bottom-right corner (south-east most pixel). On the contrary, the WATCH pixels are indexed starting at zero in the bottom-left corner (south-west most pixel) and is row-major ordered ending at the top-right corner (north-west most pixel). The STATION expression shown above starts by first converting the longitude (i.e., "LON" field) and latitude (i.e., "LAT" field) to equivalent MODIS 0.05° x_a and y_a indices:

$$x_a = \frac{\text{“LON”} + 180}{0.05} - 0.5 \tag{8}$$

$$y_a = \frac{90 - \text{“LAT”}}{0.05} - 0.5 \tag{9}$$

To address the different origins between the two numbering schemes, the MODIS row number, y_a, is subtracted from the largest row number to get the WATCH row index, (i.e., $y_b = 359 - y_a$). The column numbering is the same for both schemes (i.e., $x_b = x_a$). Finally the WATCH station numbering scheme is applied:

$$\text{“STATION”} = 720 \cdot y_b + x_b \tag{10}$$

B.2 The Python method

The Python method, instead of exporting raster images, processes the data directly from the HDF file. The methodology is similar to the QGIS method in that MODIS 0.05° resolution pixels are averaged to 0.5° resolution. To circumvent the long method presented previously, Python takes advantage of the pixel numbering scheme as explained in equations 8–10. The Python procedure begins by iterating through each of the 0.5° pixels (i.e., 360 × 720 pixel grid). For each 0.5° pixel, the 100 0.05° pixels are identified that exist within it. This is accomplished by iterating over the boundary box calculated based on the centroid coordinates of the 0.5° pixel. Each of the 100 0.05° pixels are iterated over and valid EVI values are saved to an array. The valid EVI values in the array are averaged and the average EVI is assigned to the 0.5° pixel. This is repeated for all 0.5° pixels.

Figure 5 shows two raster images of MODIS EVI at the original 0.05° resolution (Figure 5a) and at the upscaled 0.5° resolution by the Python method (Figure 5b). While the upscaling removes the finer detail from the image, the general trends in spatial EVI are well maintained. The resampling also reduces the amount of data storage required by a factor of ≈100 (e.g., 148 MB → 1.5 MB).

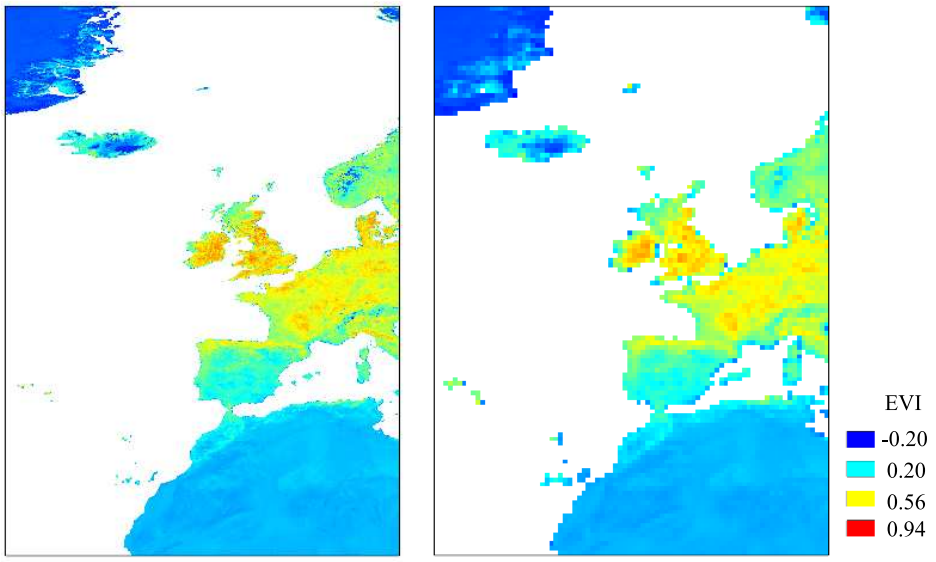


Figure 5: Pseudo-colored raster images of monthly EVI over western Europe, (i.e., longitude: -30°–13°, latitude: 80°–20°), for June 2002 at (a) 0.05° resolution and (b) 0.5° resolution (via the Python method).

C Some Useful SQL Commands

While working with the GePiSaT database, circumstances may arise that force the user to work directly with the database from the postgreSQL command prompt. This appendix highlights some of these circumstances. The syntax of each query is reviewed and explained such that it may assist in creating other queries that may be necessary.

C.1 List of variables

```
SELECT DISTINCT varname FROM var_list;
```

This command queries the field *varname* from the table `var_list` and returns a list of all the variable names currently in the database. The `DISTINCT` keyword limits the query results to only unique values of *varname* (i.e., removes all duplicate results).

C.2 Find msvidx

```
SELECT varname, varid, msvidx FROM var_list WHERE varname = 'VPD'
LIMIT 1;
```

This query returns the fields *varname*, *varid*, and *msvidx* from the table `var_list` for a specified variable name. In this example the specified variable is "VPD." Note that in the query syntax the variable name is placed between a pair of single quotes. The variable name can be any of the observation data in the database (see Table 3 for observation variables; see also Appendix C.1 for querying variable names). The `LIMIT 1` at the end of the query suppresses the number of rows returned by the query to one (since all rows will have the same *varname* and *varid*). The station that is associated with the *msvidx* will be chosen by postgreSQL by random. The purpose of this command is to associate a variable name with its identifier.

C.3 Delete single variable data

```
DELETE FROM data_set WHERE msvidx LIKE '%20';
```

This query performs the action of deleting data (i.e., rows) from the table `data_set` where the *msvidx* field meets a specific criteria. As defined in §8.2.1, the *msvidx* field is comprised of both the *stationid* and *varid* fields. By definition the last two characters in the *msvidx* field are associated with the *varid*. This query takes advantage of this knowledge and the knowledge of the *varid* for the variable of interest (see Appendix C.2). The `LIKE` keyword prompts postgreSQL for a "regular expression" search string. In postgreSQL, the escape character for "match anything" is the percent sign

(%). In this example, the *varid* is 20, which is associated with the "VPD" variable name. The '%20' (note the single quotes) will therefore match all *stationid* parts in the *msvidx* where the *varid* part matches '20.'

Following the successful query execution, postgreSQL will return the number of fields that were removed from the database.

References

Abtew, W. and A. Melesse (2013). "Evaporation and Evapotranspiration: Measurements and Estimations". In: New York: Springer. Chap. Vapor Pressure Calculation Methods, pp. 53–54 (cit. on p. 21).

Cramer, W. and I. C. Prentice (1988). "Simulation of regional soil moisture deficits on a European scale". In: *Norsk Geografisk Tidsskrift - Norwegian Journal of Geography* 42.2–3, pp. 149–151 (cit. on p. 22).

Gallego-Sala, A. V., J. M. Clark, J. I. House, H. G. Orr, I. C. Prentice, P. Smith, T. Farewell, and S. J. Chapman (2010). "Bioclimatic envelope model of climate change impacts on blanket peatland distribution in Great Britain". In: *Climate Research* 45, pp. 151–162 (cit. on p. 22).

Harris, I., P. D. Jones, T. J. Osborn, and D. H. Lister (2014). "Updated high-resolution grids of monthly climatic observations - the CRU TS3.10 Dataset". In: *International Journal of Climatology* 34, pp. 623–642 (cit. on p. 21).

Huete, A., K. Didan, T. Miura, E. P. Rodriguez, X. Gao, and L. G. Ferreira (2002). "Overview of the radiometric and biophysical performance of the MODIS vegetation indices". In: *Remote Sensing of Environment* 83, pp. 195–213 (cit. on p. 19).

Lhomme, J.-P. (1997). "A theoretical basis for the Priestley-Taylor coefficient". In: *Boundary Layer Meteorology* 82, pp. 179–191 (cit. on p. 23).

Oke, T. R. (1987). "Boundary Layer Climates". In: 2nd ed. London: Methuen and Co. Chap. Climates of Vegetated Surfaces, pp. 117–118 (cit. on p. 19).

Priestley, C. H. B. and R. J. Taylor (1972). "On the assessment of surface heat flux and evaporation using large-scale parameters". In: *Monthly Weather Review* 100.2, pp. 81–92 (cit. on p. 23).

Ruimy, A., P. G. Jarvis, D. D. Baldocchi, and B. Saugier (1995). "CO_2 fluxes over plant canopies and solar radiation: a review". In: *Advances in Ecological Research* 26, pp. 1–68 (cit. on p. 6).

Sykes, M. T. and I. C. Prentice (1995). "Boreal forest futures: modelling the controls on tree species range limits and transient responses to climate change". In: *Water, Air, and Soil Pollution* 82.1–2, pp. 415–428 (cit. on p. 22).

Sykes, M. T. and I. C. Prentice (1996). "Climate change, tree species distributions and forest dynamics: a case study in the mixed conifer/northern hardwoods zone in Northern Europe". In: *Climate Change* 34.2, pp. 161–177 (cit. on p. 23).

Sykes, M. T., I. C. Prentice, and W. Cramer (1996). "A bioclimatic model for the potential distributions of north European tree species under present and future climates". In: *Journal of Biogeography* 23, pp. 203–233 (cit. on p. 23).

Wang, H., I. C. Prentice, and J. Ni (2012). "Primary production in forests and grasslands in China: contrasting environmental responses of light- and water-use efficiency models". In: *Biogeosciences* 9, pp. 4689–4705.

Weedon, G. P., S. Gomes, G. Balsamo, M. J. Best, N. Bellouin, and P. Viterbo (2012). *WATCH forcing databased on ERA-INTERIM*. Online. URL: `ftp://rfdata:forceDATA@ftp.iiasa.ac.at` (cit. on p. 18).

Xiao, X., Q. Zhang, D. Hollinger, J. Aber, and B. III Moore (2005). "Modeling gross primary production of an evergreen needleleaf forest using MODIS and climate data". In: *Ecological Applications* 15.3, pp. 954–969 (cit. on p. 20).