SMRF Documentation

Scott Havens

Jun 10, 2020

CONTENTS

Spatial Modeling for Resources Framework (SMRF) was developed by Dr. Scott Havens at the USDA Agricultural Research Service (ARS) in Boise, ID. SMRF was designed to increase the flexibility of taking measured weather data and distributing the point measurements across a watershed. SMRF was developed to be used as an operational or research framework, where ease of use, efficiency, and ability to run in near real time are high priorities.

CHAPTER

ONE

FEATURES

SMRF was developed as a modular framework to enable new modules to be easily integrated and utilized.

- Load data into SMRF from MySQL database, CSV files, or gridded climate models (i.e. WRF)
- **Variables currently implemented:**
    - Air temperature
    - Vapor pressure
    - Precipitation mass, phase, density, and percent snow
    - Wind speed and direction
    - Solar radiation
    - Thermal radiation
- Output variables to NetCDF files
- Data queue for multithreaded application
- Computation tasks implemented in C

1.1 Getting started

1.1.1 Installation

To install SMRF locally on Linux of MacOSX, first clone the repository and build into a virtual environment. The general steps are as follows and will test the SMRF installation by running the tests.

Clone from the repository

```
git clone https://github.com/USDA-ARS-NWRC/smrf.git
```

And install the requirements, SMRF and run the tests.

```
python3 -m pip install -r requirements_dev.txt
python3 setup.py install
python3 -m unittest -v
```

For in-depth instructions and specific requirements for SMRF, check out the the *installation page*

For Windows, the install method is using Docker.

Installation

SMRF relies on the Image Processing Workbench (IPW) so it must be installed first. IPW currently has not been tested to run natively on Windows and must use Docker. Check the *Windows* section for how to run. Please go through and install the dependencies for your system prior to installing IPW and SMRF.

Note: SMRF is only maintained for Python 3 and using Python 2 may not work.

Note: SMRF uses the OpenMP specification v4.X and will not work with GCC >= 9.0.

Ubuntu

SMRF is actively developed on Ubuntu and requires gcc greater than 4.8 and less than 9.0. Install the dependencies by updating, install build-essentials and installing python3-dev:

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install python3-dev
```

Mac OSX

Mac OSX greater than 10.8 is required to run SMRF. Mac OSX comes standard with Python installed with the default compiler clang. To utilize multi-threading and parallel processing, gcc must be installed with Python compiled with that gcc version.

Install the system dependencies using MacPorts or homebrew:

a. MacPorts install system dependencies

```
port install gcc5
port install python3
```

b. Homebrew install system dependencies

```
brew tap homebrew/versions
brew install gcc5
brew install python3
```

Note: Ensure that the correct gcc and Python are activated, use `gcc --version` and `python3 --version`. If they are not set, use Homebrew or MacPorts activate features.

Windows

Since IPW has not been tested to run in Window, Docker will have to be used to run SMRF. The docker image for SMRF can be found on docker hub here. The docker image is already setup to run SMRF so the following steps do not apply for running out of a docker.

Installing IPW

Clone IPW using the command below and follow the instructions in the Install text file. If you would prefer to read the file in your browser click here.

```
git clone https://github.com/USDA-ARS-NWRC/ipw.git
```

Double check that the following environment variables are set and readable by Python

- $IPW, and $IPW/bin environment variable is set.
- WORKDIR, the location where temporary files are created and modified which is not default on Linux. Use ~/tmp for example.
- PATH, is set and readable by Python (mainly if running inside an IDE environment).

Installing SMRF

Once the dependencies have been installed for your respective system, the following will install smrf. It is preferable to use a Python virtual environment to reduce the possibility of a dependency issue.

1. Create a virtualenv and activate it.

   ```
   python3 -m virtualenv .venv
   source .venv/bin/activate
   ```

2. Clone SMRF source code from the ARS-NWRC github.

   ```
   git clone https://github.com/USDA-ARS-NWRC/smrf.git
   ```

3. Change directories into the SMRF directory. Install the python requirements. After the requirements are done, install SMRF.

   ```
   cd smrf
   python3 -m pip install -r requirements_dev.txt
   python3 setup.py install
   ```

4. (Optional) Generate a local copy of the documentation.

   ```
   make docs
   ```

   To view the documentation use the preferred browser to open up the files. This can be done from the browser by opening the index.rst file directly or by the commandline like the following:

   ```
   google-chrome _build/html/index.html
   ```

5. Test the installation by running the test suite.

   ```
   python3 -m unittest -v
   ```

If all tests passed, SMRF is installed. See examples for specific types of runs. Happy SMRF-ing!

Create a Topo

The topo provides SMRF with the following static layers:

1. Digital elevation model
2. Vegetation type
3. Vegetation height
4. Vegetation extinction coefficient
5. Vegetation optical transmissivity
6. Basin mask (optional)

All these layers are stored in a netCDF file, typically referred to the `topo.nc` file.

Note: The `topo.nc` *must* have projection information. It's just good practice.

Generating the topo

While the `topo.nc` file can be generated manually, a great option is to use basin_setup which creates a topo file that is compatible with SMRF and AWSM. To get a minimal topo file generated, the following are necessary:

1. Pour point file in `bna` format
2. Docker

`basin_setup` will perform auto basin delineation for the watershed and will output shapefiles for the basin and sub basins. Next, `basin_setup` will generate the `topo.nc` file with all the necessary variables for SMRF.

See the `basin_setup` documentation for more details.

Vegetation

The vegetation data comes from the LandFire dataset and contains the vegetation type and height at 30 meters. The vegetation is important in the following locations within SMRF

1. Adds sheltering in the wind distribution in the *`Winstral wind model`*
2. WindNinja log law roughness *`scaling`*
3. Precipitation redistribution interference in the *`Winstral precipitation`* rescaling model
4. Albedo *`decay date method`*
5. Vegetation correction to *`solar radiation`*
6. Vegetation correction to *`thermal radiation`*

Vegetation type is configured in SMRF as `veg_<type>`. For example, to add sheltering for vegetation type 3011, the configuration option `veg_3011` will be set to the value needed, say `10.0`. SMRF will apply the value `10.0` to any cells with vegetation type 3011.

Maxus file

If running SMRF with the Winstral wind model, a maxus (maximum upwind slope) file must be generated. The file is a calculation of the *maximum upwind slope* for many possible wind directions. To generate the maxus file, use the `gen_maxus` script.

```
gen_maxus --out_maxus=maxus.nc --sv_global 300 --sv_local=60 topo.nc
```

This will generate a `maxus.nc` which are the raw calulations for each direction. `gen_maxus` will also generate a `maxus_100window.nc` that averages the maxus values over a window, this averaged file is what typically is used for SMRF.

If using the Winstral precipitation rescaling method, the `tbreak.nc` is also required. Using the same command as above, adding the `--make_tbreak` will make this file.

```
gen_maxus --out_maxus=maxus.nc --sv_global 300 --sv_local=60 --make_tbreak --out_
↪tbreak=tbreak.nc topo.nc
```

Create a config file

After the topo file has been created, build the SMRF configuration file. For in depth documentation see how to *use a configuration file* and the *core configuration* for all SMRF options.

Note: Configuration file paths are relative to the configuration file location.

At a minimum to get started, the following configuration file will apply all the defaults. The required changes are specifying the path to the `topo.nc` file, dates to run the model and the location of the csv input data.

```
################################################################################
# Files for DEM and vegetation
################################################################################
[topo]
filename:                       ./topo/topo.nc

################################################################################
# Dates to run model
################################################################################
[time]
time_step:                      60
start_date:                     1998-01-14 15:00:00
end_date:                       1998-01-14 19:00:00
time_zone:                      utc

################################################################################
# CSV section configurations
################################################################################
[csv]
wind_speed:                     ./station_data/wind_speed.csv
air_temp:                       ./station_data/air_temp.csv
cloud_factor:                   ./station_data/cloud_factor.csv
wind_direction:                 ./station_data/wind_direction.csv
precip:                         ./station_data/precip.csv
vapor_pressure:                 ./station_data/vapor_pressure.csv
metadata:                       ./station_data/metadata.csv
```

(continues on next page)

(continued from previous page)

```
################################################################################
# Air temperature distribution
################################################################################
[air_temp]

################################################################################
# Vapor pressure distribution
################################################################################
[vapor_pressure]

################################################################################
# Wind speed and wind direction distribution
################################################################################
[wind]
maxus_netcdf:                   ./topo/maxus.nc

################################################################################
# Precipitation distribution
################################################################################
[precip]

################################################################################
# Albedo distribution
################################################################################
[albedo]

################################################################################
# Cloud Factor - Fraction used to limit solar radiation Cloudy (0) - Sunny (1)
################################################################################
[cloud_factor]

################################################################################
# Solar radiation
################################################################################
[solar]

################################################################################
# Incoming thermal radiation
################################################################################
[thermal]

################################################################################
# Soil temperature
################################################################################
[soil_temp]

################################################################################
# Output variables
################################################################################
[output]
out_location:                   ./output

################################################################################
# System variables and Logging
################################################################################
[system]
```

Run SMRF

After installing SMRF, generating a topo and creating a configuration file, SMRF can be ran. There are two ways to run SMRF, first is through the `run_smrf` command or through the SMRF API. If SMRF is being used as input to a snow or hydrology model, we recommend to use `run_smrf` as it will generate all the input required.

`run_smrf` command

To run a full simulation simply run (barring any errors):

```
run_smrf <config_file_path>
```

SMRF API

The `smrf` package can also be used as an API, typically to focus on a single variable. There are steps that SMRF uses to load the data then distribute and usage of the API should follow the same pattern. For example, below is the function *run_smrf*.

```
with SMRF(config) as s:
    # load topo data
    s.loadTopo()

    # initialize the distribution
    s.initializeDistribution()

    # initialize the outputs if desired
    s.initializeOutput()

    # load weather data  and station metadata
    s.loadData()

    # distribute
    s.distributeData()
```

The next example below builds on above and will distribute air temperature and vapor pressure. They can be used to get the distributed dew point or web bulb temperature.

```
configFile = 'config.ini'

with smrf.framework.SMRF(configFile) as s:

    # ===================================================================
    # Model setup and initialize
    # ===================================================================

    # These are steps that will load the necessary data and initialize
    # the framework. Once loaded, this shouldn't need to be re-ran except
    # if something major changes

    # load topo data
    s.loadTopo()

    # Create the distribution class
    s.distribute['air_temp'] = smrf.distribute.air_temp.ta(
```

(continues on next page)

(continued from previous page)

```
        s.config['air_temp'])
    s.distribute['vapor_pressure'] = smrf.distribute.vapor_pressure.vp(
        s.config['vapor_pressure'])

    # load weather data  and station metadata
    s.loadData()

    # Initialize the distribution
    for v in s.distribute:
        s.distribute[v].initialize(s.topo, s.data)

    # initialize the outputs if desired
    s.initializeOutput()

    # Distribute the data and output
    for output_count, t in enumerate(s.date_time):

        s.distribute['air_temp'].distribute(s.data.air_temp.ix[t])
        s.distribute['vapor_pressure'].distribute(
            s.data.vapor_pressure.ix[t],
            s.distribute['air_temp'].air_temp)

        # output at the frequency and the last time step
        if (output_count % s.config['output']['frequency'] == 0) or \
                (output_count == len(s.date_time)):
            s.output(t)
```

SMRF and Docker

SMRF is also built into a docker image to make it easy to install on any operating system. The docker images are built automatically from the Github repository and include the latest code base or stable release images.

The SMRF docker image has a folder meant to mount data inside the docker image at `/data`.

```
docker run -v <path to data>:/data usdaarsnwrc/smrf run_smrf <path to config>
```

The `<path to data>` should be the path to where the configuration file, data and topo are on the host machine. This will also be the location to where the SMRF output will go.

Note: The paths in the configuration file must be adjusted for being inside the docker image. For example, in the command above the path to the config will be inside the docker image. This would be `/data/config.ini` and not the path on the host machine.

In a way that ARS uses this, we keep the config, topo and data on one location as the files are fairly small. The output then is put in another location as it file size can be much larger. To facilitate this, mount the input and output data separately and modify the configuration paths.

```
docker run -v <input>:/data/input -v <output>:/data/output usdaarsnwrc/smrf run_smrf
↪<path to config>
```

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/USDA-ARS-NWRC/smrf/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it. If the added feature expands the options available in the config flie, please make them available by adding to the CoreConfig.ini in ./smrf/framework/CoreConfig.ini. For more information on syntax for this, please reference the configuration section.

Write Documentation

SMRF could always use more documentation, whether as part of the official SMRF docs, in docstrings, or even on the web in blog posts, articles, and such.

Versioning

SMRF uses bumpversion to version control. More about bumpversion can be found at https://pypi.python.org/pypi/bumpversion. This can easily be used with the command:

```
$ bumpbversion patch --tag
```

Don't forget to push your tags afterwards with:

```
$ git push origin --tags
```

The development team of SMRF attempted to adhere to semantic versioning. Here is the basics taken from the semantic versioning website.

- Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

- Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented
- Major version X (X.y.z | X > 0) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY include minor and patch level changes. Patch and minor version MUST be reset to 0 when major version is incremented.

For more info on versions see http://semver.org

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/USDA-ARS-NWRC/smrf/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *smrf* for local development.

1. Fork the *smrf* repo on GitHub.
2. Clone your fork locally:

```
$ git clone https://github.com/your_name_here/smrf
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv smrf
$ cd smrf/
$ pip install -r requirements.txt
$ pip install -e .
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 smrf
$ python setup.py test
```

   To get flake8, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4+, and for PyPy. Check https://travis-ci.com/USDA-ARA-NWRC/smrf/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python3 -m unittest discover -v
```

To check the coverage of the tests:

```
$ coverage run --source smrf setup.py test
$ coverage html
$ xdg-open htmlcov/index.html
```

History

0.10.0 (TBD)

- Cloud factor was removed from solar and made its own module
- IPW Topo has been fully dperecated and removed
- Dozens config file options were renamed in favor of verbosity
- New documentation

0.9.0 (2019-12-05)

- First formal release under new branching model
- Updated Weather forecast retrieval in dockerfile
- Fixed a bug with tbreak in the wind calc
- Fixed Cloud Factor typo
- Reduced designated solar hours to limit dusk and dawn effects
- Expanded tests to HRRR input data

- Performance improvements to the gridded input data calculations

0.8.0 (2019-02-06)

- Added local gradient interpolation option for use with gridded data
- Removed ipw package to installed spatialnc dependency
- Added projection info to output files

0.7.0 (2018-11-28)

- New cloud factor method for HRRR data
- Added use of WindNinja outputs from Katana package using HRRR data
- Added unit testing as well as Travis CI and Coveralls
- Added PyKrig
- Various bug fixes

0.6.0 (2018-07-13)

- Added a new feature allowing wet bulb to be used to determine the phase of the precip.
- Added a new feature to redistribute precip due to wind.
- Added in kriging as a new distribution option all distributable variables.

0.5.0 (2018-04-18)

- Removed inicheck to make its own package.
- Added in HRRR input data for new gridded type
- Fixed various bugs associated with precip
- Modularized some functions for easiuer use scripting
- Added netcdf functionality to gen_maxus
- Added first integration test

0.4.0 (2017-11-14)

- Small improvements to our config file code including: types checking, relative paths to config, auto documentation
- Fixed bugs related to precip undercatch
- Improvements to ti station data backup
- Various adjustments for better collaboration with AWSM
- Moved to a new station database format

0.3.0 (2017-09-08)

- New feature for backing up the input data for a run in csv.
- Major update to config file, enabling checking and default adding
- Updated C file prototypes.

0.2.0 (2017-05-09)

- SMRF can run with Python 3
- Fixed indexing issue in wind
- Minor Config file improvements.

1.2 User Guide

The User Guide covers concepts within SMRF. This starts with the configuration file where a user can specify parameters for almost any part of SMRF. The *CoreConfig* has detailed information on all of the SMRF configuration options.

1.2.1 Using Configuration Files

SMRF simulation details are managed using configuration files. The python package inicheck is used to manage and interpret the configuration files. Each configuration file is broken down into sections containing items and each item is assigned a value.

A brief description of the syntax is:

- Sections are noted by being in a line by themselves and are bracketed.
- Items are denoted by colon (**:**).
- Values are simply written in, and values that are lists are comma separated.
- Comments are preceeded by a **#**

For more information regarding inicheck syntax and utilities refer to the inicheck documentation.

Understanding Configuration Files

The easiest way to get started is to look at one of the config files in the repo already. A simple case to use is the Lakes Basin test which can be view easily here.

Take a look at the "topo" section from the config file show below

```
################################################################################
# Files for DEM and vegetation
################################################################################

[topo]
filename:                    ./topo/topo.nc
```

This section describes all the topographic information required for SMRF to run. At the top of the section there is comment that describes the section. The section name "topo" is bracketed to show it is a section and the items underneath are assigned values by using the colon.

Editing/Checking Configuration Files

Use any text editor to make changes to a config file. Some editors have the ability to read and edit .ini syntax.

If you are unsure of what to use various entries in your config file refer to the *CoreConfig* or use the inicheck command for command line help. Below is an example of how to use the inicheck details option to figure out what options are available for the topo section type item.

```
inicheck --details topo <filename> -m smrf
```

The output is:

```
Providing details for section topo and item filename...

Section          Item             Default          Options                    Description
==========================================================================================================
topo             filename         None             []                         A netCDF␣
↪file containing all veg info and dem.
```

Creating Configuration Files

Not all items and options need to be assigned, if an item is left blank it will be assigned a default. If it is a required parameter, SMRF will throw an error until it is assigned.

To make an up to date config file use the following command to generate a fully populated list of options.

```
inicheck -f config.ini -m smrf -w
```

This will create a config file using the same name but call "config_full.ini" at the end.

Core Configuration File

Each configuration file is checked against the core configuration file stored *./smrf/framework/CoreConfig.ini* and various scenarios are guided by the a recipes file that is stored in *./smrf/framework/recipes.ini*. These files work together to guide the outcomes of the configuration file.

To learn more about syntax and how to contribute to a Core or Master configuration file see Master Configuration Files in inicheck.

1.2.2 Configuration File Reference

The SMRF configuration file is described in detail below. This information is all based on the CoreConfig file stored under framework.

For configuration file syntax information please visit http://inicheck.readthedocs.io/en/latest/

topo

filename

A netCDF file containing all veg info and dem.

Default: None

Type: criticalfilename

gradient_method

Method to use for calculating the slope and aspect. gradient_d8 uses 3 by 3 finite difference window and gradient_d4 uses a two cell finite difference for x and y which mimics the IPW gradient function

Default: gradient_d8

Type: string

Options: gradient_d8 gradient_d4

northern_hemisphere

Boolean describing whether the model domain is in the northern hemisphere or not

Default: True

Type: bool

time

end_date

Date and time to end the data distribution that can be parsed by pandas.to_datetime

Default: None

Type: datetimeorderedpair

start_date

Date and time to start the data distribution that can be parsed by pandas.to_datetime

Default: None

Type: datetimeorderedpair

time_step

Time interval that SMRF distributes data at in minutes

Default: 60

Type: int

time_zone

Time zone for all times provided and how the model will be run see pytz docs for information on what is accepted

Default: UTC

Type: string

csv

air_temp

Path to CSV containing the station measured air temperature

Default: None

Type: criticalfilename

cloud_factor

Path to CSV containing the station measured cloud factor

Default: None

Type: criticalfilename

metadata

Path to CSV containing the station metadata

Default: None

Type: criticalfilename

precip

Path to CSV containing the station measured precipitation

Default: None

Type: criticalfilename

stations

List of station IDs to use for distributing any of the variables

Default: None

Type: station

vapor_pressure

Path to CSV containing the station measured vapor pressure

Default: None

Type: criticalfilename

wind_direction

Path to CSV containing the station measured wind direction

Default: None

Type: criticalfilename

wind_speed

Path to CSV containing the station measured wind speed

Default: None

Type: criticalfilename

mysql

air_temp

name of the table column containing station air temperature

Default: air_temp

Type: string

cloud_factor

name of the table column containing station cloud factor

Default: cloud_factor

Type: string

data_table

name of the database table containing station data

Default: tbl_level2

Type: string

database

name of the database containing station data

Default: weather_db

Type: string

host

IP address to server.

Default: None

Type: string

metadata
name of the database table containing station metadata
Default: tbl_metadata
Type: string

password
password used for database login.
Default: None
Type: password

port
Port for MySQL database.
Default: 3606
Type: int

precip
name of the table column containing station precipitation
Default: precip_accum
Type: string

solar
name of the table column containing station solar radiation
Default: solar_radiation
Type: string

station_table
name of the database table containing client and source
Default: tbl_stations
Type: string

stations
List of station IDs to use for distributing any of the variables
Default: None
Type: station

user

username for database login.

Default: None

Type: string

vapor_pressure

name of the table column containing station vapor pressure

Default: vapor_pressure

Type: string

wind_direction

name of the table column containing station wind direction

Default: wind_direction

Type: string

wind_speed

name of the table column containing station wind speed

Default: wind_speed

Type: string

gridded

data_type

Type of gridded input data

Default: hrrr_netcdf

Type: string

Options: wrf hrrr_grib netcdf hrrr_netcdf

hrrr_directory

Path to the top level directory where multiple HRRR gridded dataset are located

Default: None

Type: criticaldirectory

hrrr_forecast_flag

True if the HRRR data is a forecast

Default: False

Type: bool

netcdf_file

Path to the netCDF file containing weather data

Default: None

Type: criticalfilename

wrf_file

Path to the netCDF file containing WRF data

Default: None

Type: criticalfilename

air_temp

The air_temp section controls all the available parameters that effect the distribution of the air_temp module, espcially the associated models. For more detailed information please see `smrf.distribute.air_temp`

detrend

Whether to elevationally detrend prior to distributing

Default: true

Type: bool

detrend_slope

If detrend is true constrain the detrend_slope to positive (1) or negative (-1) or no constraint (0)

Default: -1

Type: int

Options: -1 0 1

distribution

Distribution method to use for <this variable>. Stations use dk idw or kriging. Gridded data use grid. Stations use dk idw or kriging. Gridded data use grid.

Default: idw

Type: string

Options: dk idw grid kriging

dk_ncores

Number of threads or processors to use in the dk calculation

Default: 1

Type: int

grid_local

Use local elevation gradients in gridded interpolation

Default: False

Type: bool

grid_local_n

number of closest grid cells to use for calculating elevation gradient

Default: 25

Type: int

grid_mask

Mask the distribution calculations

Default: True

Type: bool

grid_method

Gridded interpolation method to use for air temperature

Default: cubic

Type: string

Options: nearest linear cubic

idw_power

Power for decay of a stations influence in inverse distance weighting.

Default: 2.0

Type: float

krig_anisotropy_angle

CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy.

Default: 0.0

Type: float

krig_anisotropy_scaling

Scalar stretching value for kriging to take into account anisotropy.

Default: 1.0

Type: float

krig_coordinates_type
Determines if the x and y coordinates are interpreted as on a plane (euclidean) or as coordinates on a sphere (geographic).
Default: euclidean
Type: string
Options: euclidean geographic

krig_nlags
Number of averaging bins for the kriging semivariogram
Default: 6
Type: int

krig_variogram_model
Specifies which kriging variogram model to use
Default: linear
Type: string
Options: linear power gaussian spherical exponential hole-effect

krig_weight
Flag that specifies if the kriging semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model.
Default: False
Type: bool

max
Maximum possible value for air temperature in Celsius
Default: 47.0
Type: float

min
Minimum possible value for air temperature in Celsius
Default: -73.0
Type: float

stations
Stations to use for distributing air temperature
Default: None
Type: station

vapor_pressure

The vapor_pressure section controls all the available parameters that effect the distribution of the vapor_pressure module, espcially the associated models. For more detailed information please see `smrf.distribute.vapor_pressure`

detrend
Whether to elevationally detrend prior to distributing

Default: true

Type: bool

detrend_slope
If detrend is true constrain the slope to positive (1) or negative (-1) or no constraint (0)

Default: -1

Type: int

Options: -1 0 1

dew_point_nthreads
Number of threads to use in the dew point calculation

Default: 2

Type: int

dew_point_tolerance
Solving criteria for the dew point calculation

Default: 0.01

Type: float

distribution
Distribution method to use for vapor pressure. Stations use dk idw or kriging. Gridded data use grid.

Default: idw

Type: string

Options: dk idw grid kriging

dk_ncores
Number of threads to use in the dk calculation

Default: 1

Type: int

grid_local
Use local elevation gradients in gridded interpolation
Default: False
Type: bool

grid_local_n
number of closest grid cells to use for calculating elevation gradient
Default: 25
Type: int

grid_mask
Mask the distribution calculations
Default: True
Type: bool

grid_method
interpolation method to use for this variable
Default: cubic
Type: string
Options: nearest linear cubic

idw_power
Power for decay of a stations influence in inverse distance weighting
Default: 2.0
Type: float

krig_anisotropy_angle
CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy.
Default: 0.0
Type: float

krig_anisotropy_scaling
Scalar stretching value for kriging to take into account anisotropy.
Default: 1.0

Type: float

krig_coordinates_type

Determines if the x and y coordinates are interpreted as on a plane (euclidean) or as coordinates on a sphere (geographic).

Default: euclidean

Type: string

Options: euclidean geographic

krig_nlags

Number of averaging bins for the kriging semivariogram

Default: 6

Type: int

krig_variogram_model

Specifies which kriging variogram model to use

Default: linear

Type: string

Options: linear power gaussian spherical exponential hole-effect

krig_weight

Flag that specifies if the kriging semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model.

Default: False

Type: bool

max

Maximum possible vapor pressure in Pascals

Default: 5000.0

Type: float

min

Minimum possible vapor pressure in Pascals

Default: 20.0

Type: float

stations

Stations to use for distributing vapor pressure in Pascals

Default: None

Type: station

wind

The wind section controls all the available parameters that effect the distribution of the wind module, espcially the associated models. For more detailed information please see `smrf.distribute.wind`

detrend

Whether to elevationally detrend prior to distributing

Default: False

Type: bool

detrend_slope

if detrend is true constrain the detrend_slope to positive (1) or negative (-1) or no constraint (0)

Default: 1

Type: int

Options: -1 0 1

distribution

Distribution method to use for wind. Stations use dk idw or kriging. Gridded data use grid.

Default: idw

Type: string

Options: dk idw grid kriging

dk_ncores

Number of threads to use in the dk calculation

Default: 2

Type: int

grid_local

Use local elevation gradients in gridded interpolation

Default: False

Type: bool

grid_local_n
Number of closest grid cells to use for calculating elevation gradient
Default: 25
Type: int

grid_mask
Mask the distribution calculations
Default: True
Type: bool

grid_method
interpolation method to use for wind
Default: linear
Type: string
Options: nearest linear cubic

idw_power
Power for decay of a stations influence in inverse distance weighting
Default: 2.0
Type: float

krig_anisotropy_angle
CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy.
Default: 0.0
Type: float

krig_anisotropy_scaling
Scalar stretching value for kriging to take into account anisotropy.
Default: 1.0
Type: float

krig_coordinates_type
Determines if the x and y coordinates are interpreted as on a plane (euclidean) or as coordinates on a sphere (geographic).
Default: euclidean
Type: string
Options: euclidean geographic

krig_nlags

Number of averaging bins for the kriging semivariogram

Default: 6

Type: int

krig_variogram_model

Specifies which kriging variogram model to use

Default: linear

Type: string

Options: linear power gaussian spherical exponential hole-effect

krig_weight

Flag that specifies if the kriging semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model.

Default: False

Type: bool

max

Maximum possible wind in M/s

Default: 35.0

Type: float

maxus_netcdf

NetCDF file containing the maxus values for wind

Default: None

Type: criticalfilename

min

Minimum possible for wind in M/s

Default: 0.447

Type: float

reduction_factor

If wind speeds are still off here is a scaling factor

Default: 1.0

Type: float

station_default
Account for sheltered station wind measurements for example 11.4 equates to a small forest opening and 0 equates to unsheltered measurements.
Default: 11.4
Type: float

station_peak
Name of stations that lie on a peak or a high point
Default: None
Type: station

stations
Stations to use for distributing wind in M/s
Default: None
Type: station

veg_3011
Applies the value where vegetation equals 3011(Rocky Mountain aspen)
Default: 3.3
Type: float

veg_3061
Applies the value where vegetation equals 3061(mixed aspen)
Default: 3.3
Type: float

veg_41
Applies the value where vegetation type equals NLCD class 41
Default: 3.3
Type: float

veg_42
Applies the value where vegetation type equals NLCD class 42
Default: 3.3

Type: float

veg_43

Applies the value where vegetation type equals NLCD class 43

Default: 11.4

Type: float

veg_default

Applies the value to all vegetation not specified

Default: 0.0

Type: float

wind_model

Wind model to interpolate wind measurements to the model domain

Default: winstral

Type: string

Options: winstral wind_ninja interp

wind_ninja_dir

Location in which the ascii files are output from the WindNinja simulation. This serves as a trigger for checking for WindNinja files.

Default: None

Type: criticaldirectory

wind_ninja_dxdy

grid spacing at which the WindNinja ascii files are output.

Default: 100

Type: int

wind_ninja_height

The output height of wind fields from WindNinja in meters.

Default: 5.0

Type: string

wind_ninja_pref

Prefix of all outputs from WindNinja that matches the topo input to WindNinja.

Default: None
Type: string

wind_ninja_roughness
The surface roughness used in WindNinja generally grass.
Default: 0.01
Type: string

wind_ninja_tz
Time zone that from the WindNinja config.
Default: UTC
Type: string

precip

The precipitation section controls all the available parameters that effect the distribution of the precipitation module, espcially the associated models. For more detailed information please see `smrf.distribute.precipitation`

detrend
Whether to elevationally detrend prior to distributing
Default: true
Type: bool

detrend_slope
if detrend is true constrain the detrend_slope to positive (1) or negative (-1) or no constraint (0)
Default: 1
Type: int
Options: -1 0 1

distribution
Distribution method to use for precipitation. Stations use dk idw or kriging. Gridded data use grid.
Default: dk
Type: string
Options: dk idw grid kriging

dk_ncores
Number of threads to use in the dk calculation

Default: 2
Type: int

grid_local
Use local elevation gradients in gridded interpolation
Default: False
Type: bool

grid_local_n
number of closest grid cells to use for calculating elevation gradient
Default: 25
Type: int

grid_mask
Mask the distribution calculations
Default: True
Type: bool

grid_method
interpolation method to use for precipitation
Default: cubic
Type: string
Options: nearest linear cubic

idw_power
Power for decay of a stations influence in inverse distance weighting
Default: 2.0
Type: float

krig_anisotropy_angle
CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy.
Default: 0.0
Type: float

krig_anisotropy_scaling
Scalar stretching value for kriging to take into account anisotropy.

Default: 1.0
Type: float

krig_coordinates_type
Determines if the x and y coordinates are interpreted as on a plane (euclidean) or as coordinates on a sphere (geographic).
Default: euclidean
Type: string
Options: euclidean geographic

krig_nlags
Number of averaging bins for the kriging semivariogram
Default: 6
Type: int

krig_variogram_model
Specifies which kriging variogram model to use
Default: linear
Type: string
Options: linear power gaussian spherical exponential hole-effect

krig_weight
Flag that specifies if the kriging semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model.
Default: False
Type: bool

marks2017_timesteps_to_end_storms
number of timesteps to elapse with precip under start criteria before ending a storm.
Default: 6
Type: int

max
Maximum possible precipitation in millimeters
Default: None
Type: float

min

Minimum possible for precipitation in millimeters

Default: 0.0

Type: float

new_snow_density_model

Method to use for calculating the new snow density

Default: susong1999

Type: string

Options: marks2017 susong1999 piecewise_susong1999

precip_rescaling_model

Method to use for redistributing precipitation. Winstrals method focuses forming drifts from wind

Default: None

Type: string

Options: winstral

precip_temp_method

which variable to use for precip temperature

Default: dew_point

Type: string

Options: dew_point wet_bulb

station_adjust_for_undercatch

Apply undercatch relationships to precip gauges

Default: true

Type: bool

station_undercatch_model_default

WMO model used to adjust for undercatch of precipitaiton

Default: us_nws_8_shielded

Type: string

Options: us_nws_8_shielded us_nws_8_unshielded

stations

Stations to use for distributing this precipitation

Default: None

Type: station

storm_days_restart

Path to netcdf representing the last storm days so a run can continue in between stops

Default: None

Type: discretionarycriticalfilename

storm_mass_threshold

Start criteria for a storm in mm of measured precipitation in millimeters in any pixel over the domain.

Default: 1.0

Type: float

susong1999_timesteps_to_end_storms

number of timesteps to elapse with precip under start criteria before ending a storm.

Default: 6

Type: int

winstral_max_drift

max multiplier for precip redistribution in a drift cell

Default: 3.5

Type: float

winstral_max_scour

max multiplier for precip redistribution to account for wind scour.

Default: 1.0

Type: float

winstral_min_drift

min multiplier for precip redistribution in a drift cell

Default: 1.0

Type: float

winstral_min_scour

minimum multiplier for precip redistribution to account for wind scour.

Default: 0.55

Type: float

winstral_tbreak_netcdf
NetCDF file containing the tbreak values for wind
Default: None
Type: filename

winstral_tbreak_threshold
Threshold for drift cells measured in degrees from tbreak file.
Default: 7.0
Type: float

winstral_veg_3011
Interference inverse factor for precip redistribution where vegetation equals 3011(Rocky Mountain Aspen).
Default: 0.7
Type: float

winstral_veg_3061
Interference inverse factor for precip redistribution where vegetation equals 3061(Mixed Aspen).
Default: 0.7
Type: float

winstral_veg_41
Interference inverse factor for precip redistribution where vegetation equals 41.
Default: 0.7
Type: float

winstral_veg_42
Interference inverse factor for precip redistribution where vegetation equals 42.
Default: 0.7
Type: float

winstral_veg_43
Interference inverse factor for precip redistribution where vegetation equals 43.
Default: 0.7
Type: float

winstral_veg_default
Applies the value to all vegetation not specified
Default: 1.0
Type: float

albedo

The albedo section controls all the available parameters that effect the distribution of the albedo module, espcially the associated models. For more detailed information please see `smrf.distribute.albedo`

date_method_decay_power
Exponent value of the decay rate equation prescribed by the method.
Default: 0.714
Type: float

date_method_end_decay
Starting date for applying the decay method described by date_method
Default: None
Type: datetimeorderedpair

date_method_start_decay
Starting date for applying the decay method described by date_method
Default: None
Type: datetimeorderedpair

date_method_veg_41
Applies the value where vegetation equals 41
Default: 0.36
Type: float

date_method_veg_42
Applies the value where vegetation equals 42
Default: 0.36
Type: float

date_method_veg_43
Applies the value where vegetation equals 43

Default: 0.25
Type: float

date_method_veg_default
Applies the value to all vegetation not specified
Default: 0.25
Type: float

decay_method
Describe how the albedo decays in the late season
Default: None
Type: string
Options: * hardy2000 date_method none*

dirt
Effective contamination for adjustment to visible albedo (usually between 1.5-3.0)
Default: 2.0
Type: float

grain_size
Effective optical grain radius of snow after last storm in micro-meters
Default: 100.0
Type: float

grid_mask
Mask the distribution calculations
Default: True
Type: bool

hardy2000_litter_albedo
Albedo of the litter on the snow using the hard method
Default: 0.2
Type: float

hardy2000_litter_default
Litter rate for places where vegetation not specified for Hardy et al. 2000 decay method

Default: 0.003
Type: float

hardy2000_litter_veg_41
Litter rate for places where vegetation not specified for Hardy et al. 2000 decay method for vegetation classes NLCD 41
Default: 0.006
Type: float

hardy2000_litter_veg_42
Litter rate for places where vegetation not specified for Hardy et al. 2000 decay method for vegetation classes NLCD 42
Default: 0.006
Type: float

hardy2000_litter_veg_43
Litter rate for places where vegetation not specified for Hardy et al. 2000 decay method for vegetation classes NLCD 43
Default: 0.003
Type: float

max
Maximum possible for albedo
Default: 1.0
Type: float

max_grain
Max optical grain radius of snow possible in micro-meters
Default: 700.0
Type: float

min
Minimum possible for albedo
Default: 0.0
Type: float

cloud_factor

The cloud_factor section controls all the available parameters that effect the distribution of the cloud_factor module, espcially the associated models. For more detailed information please see `smrf.distribute.cloud_factor`

detrend
Whether to elevationally detrend prior to distributing
Default: false
Type: bool

detrend_slope
If detrend is true constrain the detrend_slope to positive (1) or negative (-1) or no constraint (0)
Default: 0
Type: int
Options: -1 0 1

distribution
Distribution method to use for cloud factor. Stations use dk idw or kriging. Gridded data use grid. Stations use dk idw or kriging. Gridded data use grid.
Default: idw
Type: string
Options: dk idw grid kriging

dk_ncores
Number of threads or processors to use in the dk calculation
Default: 1
Type: int

grid_local
Use local elevation gradients in gridded interpolation
Default: False
Type: bool

grid_local_n
number of closest grid cells to use for calculating elevation gradient
Default: 25
Type: int

grid_mask
Mask the distribution calculations
Default: True
Type: bool

grid_method
Gridded interpolation method to use for cloud factor
Default: cubic
Type: string
Options: nearest linear cubic

idw_power
Power for decay of a stations influence in inverse distance weighting.
Default: 2.0
Type: float

krig_anisotropy_angle
CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy.
Default: 0.0
Type: float

krig_anisotropy_scaling
Scalar stretching value for kriging to take into account anisotropy.
Default: 1.0
Type: float

krig_coordinates_type
Determines if the x and y coordinates are interpreted as on a plane (euclidean) or as coordinates on a sphere (geographic).
Default: euclidean
Type: string
Options: euclidean geographic

krig_nlags
Number of averaging bins for the kriging semivariogram
Default: 6
Type: int

krig_variogram_model

Specifies which kriging variogram model to use

Default: linear

Type: string

Options: linear power gaussian spherical exponential hole-effect

krig_weight

Flag that specifies if the kriging semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model.

Default: False

Type: bool

max

Max prossible cloud factor as a decimal representing full clouds (0) to full sun (1).

Default: 1.0

Type: float

min

Minimum possible cloud factor as a decimal representing full clouds (0) to full sun (1).

Default: 0.0

Type: float

stations

Stations to use for distributing cloud factor as a decimal representing full clouds (0) to full sun (1).

Default: None

Type: station

solar

The solar section controls all the available parameters that effect the distribution of the solar module, espcially the associated models. For more detailed information please see `smrf.distribute.solar`

clear_gamma

Scattering asymmetry parameter

Default: 0.3

Type: float

clear_omega
: Single-scattering albedo

Default: 0.85
Type: float

clear_opt_depth
: Elevation of optical depth measurement

Default: 100.0
Type: float

clear_tau
: Optical depth at z

Default: 0.2
Type: float

correct_albedo
: Multiply the solar radiation by 1-snow_albedo.

Default: true
Type: bool

correct_cloud
: Multiply the solar radiation by the cloud factor derived by station data.

Default: true
Type: bool

correct_veg
: Apply solar radiation corrections according to veg_type

Default: true
Type: bool

max
: Maximum possible solar radiation in W/m^2

Default: 800.0
Type: float

min

Minimum possible solar radiation in W/m^2

Default: 0.0

Type: float

thermal

The thermal section controls all the available parameters that effect the distribution of the thermal module, espcially the associated models. For more detailed information please see `smrf.distribute.thermal`

clear_sky_method

Method for calculating the clear sky thermal radiation

Default: marks1979

Type: string

Options: marks1979 dilley1998 prata1996 angstrom1918

cloud_method

Method for adjusting thermal radiation due to cloud effects

Default: garen2005

Type: string

Options: garen2005 unsworth1975 kimball1982 crawford1999

correct_cloud

Specify whether to use the cloud adjustments in thermal calculation

Default: true

Type: bool

correct_terrain

Specify whether to account for vegetation in the thermal calculations

Default: true

Type: bool

correct_veg

Specify whether to account for vegetation in the thermal calculations

Default: true

Type: bool

detrend
: Whether to elevationally the detrend prior to distributing

  Default: False
  Type: bool

detrend_slope
: if detrend is true constrain the detrend_slope to positive (1) or negative (-1) or no constraint (0)

  Default: 0
  Type: int
  Options: -1 0 1

distribution
: Distribution method to use for incoming thermal when using HRRR input data.

  Default: grid
  Type: string
  Options: grid

grid_local
: Use local elevation gradients in gridded interpolation

  Default: False
  Type: bool

grid_local_n
: number of closest grid cells to use for calculating elevation gradient

  Default: 25
  Type: int

grid_mask
: Mask the thermal radiation calculations

  Default: True
  Type: bool

grid_method
: interpolation method to use for this variable

  Default: cubic
  Type: string
  Options: nearest linear cubic

marks1979_nthreads

Number of threads to use thermal radiation calcs when using Marks1979

Default: 2

Type: int

max

Maximum possible incoming thermal radiation in W/m^2

Default: 600.0

Type: float

min

Minimum possible incoming thermal radiation in W/m^2

Default: 0.0

Type: float

soil_temp

The soil_temp section controls all the available parameters that effect the distribution of the soil_temp module, especially the associated models. For more detailed information please see *smrf.distribute.soil_temp*

temp

Constant value to use for the soil temperature.

Default: -2.5

Type: float

output

file_type

Format to use for outputting data.

Default: netcdf

Type: string

Options: netcdf

frequency

Number of timesteps between output values. 1 is every timestep.

Default: 1

Type: int

input_backup
Specify whether to backup the input data and create config file to run the smrf run from that backup
Default: true
Type: bool

mask_output
Mask the final NetCDF output.
Default: False
Type: bool

out_location
Directory to output results
Default: None
Type: directory

variables
Variables to output after being calculated.
Default: thermal air_temp vapor_pressure wind_speed wind_direction net_solar precip percent_snow snow_density precip_temp
Type: string
Options: all air_temp albedo_vis albedo_ir precip percent_snow snow_density storm_days precip_temp clear_ir_beam clear_ir_diffuse clear_vis_beam clear_vis_diffuse cloud_factor cloud_ir_beam cloud_ir_diffuse cloud_vis_beam cloud_vis_diffuse net_solar veg_ir_beam veg_ir_diffuse veg_vis_beam veg_vis_diffuse thermal vapor_pressure dew_point flatwind wind_speed wind_direction thermal_clear thermal_veg thermal_cloud

system

log_file
File path to a txt file for the log info to be outputted
Default: None
Type: filename

log_level
level of information to be logged
Default: debug

Type: string
Options: debug info error

qotw

Default: false
Type: bool

queue_max_values
How many timesteps that a calculation can get ahead while threading if it is independent of other variables.
Default: 2
Type: int

threading
Specify whether to use python threading in calculations.
Default: true
Type: bool

time_out
Amount of time to wait for a thread before timing out
Default: None
Type: float

1.2.3 Input Data

To generate all the input forcing data required to run iSnobal, the following measured or derived variables are needed

- Air temperature
- Vapor pressure
- Precipitation
- Wind speed and direction
- Cloud factor

This page documents a more detailed description of each of the input variables, the types of input data that can be used for SMRF, and the data format for passing the data to SRMF.

Variable Descriptions

Air temperature [Celcius] Measured or modeled air temperature at the surface

Vapor pressure [Pascals] Derived from the air temperature and measured relative humidity. Can be calculated using the IPW utility `sat2vp` or the SMRF function `rh2vp`.

Precipitation [mm] Instantaneous precipitation with no negative values. If using a weighing precipitation gauge that outputs accumulated precipitation, the value must be converted.

Wind speed [meters per second] The measured wind speed at the surface. Typically an average value over the measurement interval.

Wind direction [degrees] The measured wind direction at the surface. Typically an average value over the measurement interval.

Cloud factor [None] The percentage between 0 and 1 of the incoming solar radiation that is obstructed by clouds. 0 equates to no light and 1 equates to no clouds. The cloud factor is derived from the measured solar radiation and the modeled clear sky solar raditation. The modeled clear sky solar radiation can be calculated using the IPW utility `twostream` or the SMRF function `model_solar`.

Input Data Types

All types of input data to SMRF are assumed to be point measurements. Therefore, each measurement location must have a X, Y, and elevation associated with it.

Weather Stations

Generally, SMRF will be run using measured variables from weather stations in and around the area of interest. Below are some potential websites for finding data for weather stations:

- Mesowest
- NRCS SNOTEL
- California Data Exchange Center

Gridded Model Output

Gridded datasets can be used as input data for SMRF. The typical use will be for downscaling gridded weather model forecasts to the snow model domain in order to produce a short term snowpack forecast. In theory, any resolution can be utilized, but the methods have been tested and developed using Weather Reserach and Forecasting (WRF) at a 1 and 3 km resolution. Each grid point will be used as if it were a weather stations, with it's own X, Y, and elevation. Therefore, the coarse resolution model terrain can be taken into account when downscaling to a higher resolution DEM.

See Havens et al. (2017) for more details and further discussion on using WRF for forcing iSnobal.

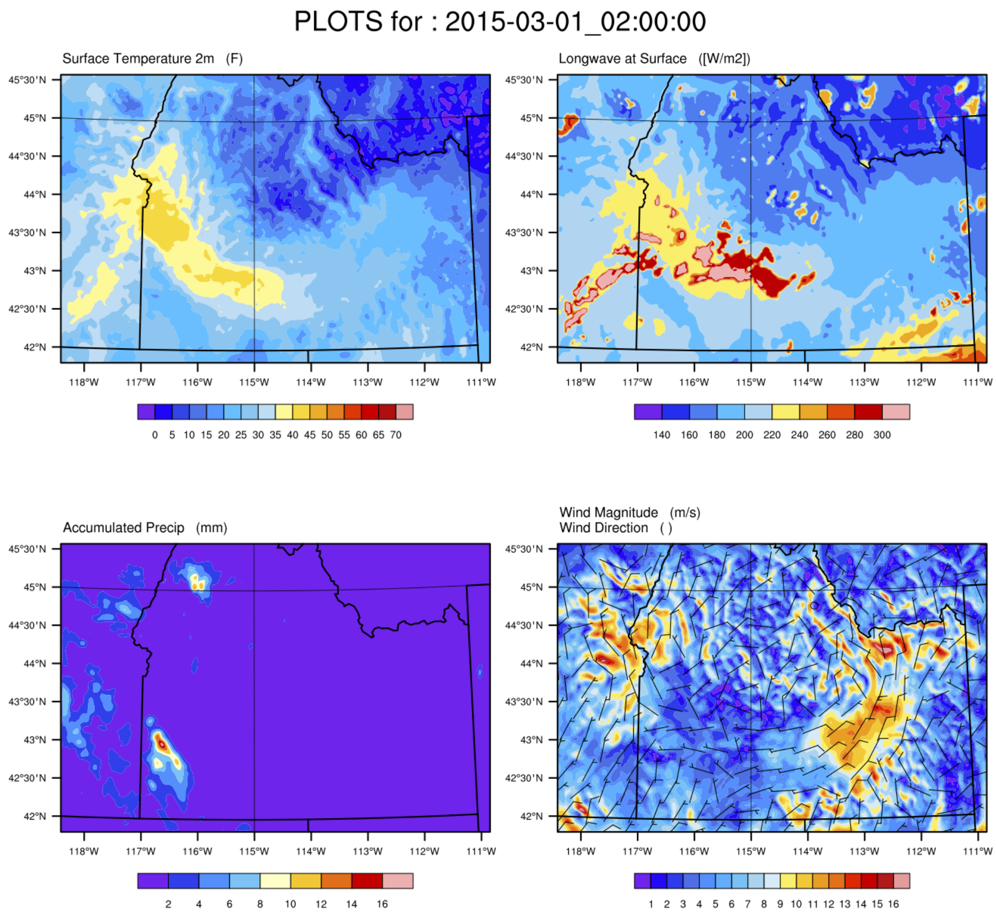


Fig. 1.1: Using WRF as a gridded dataset for SMRF.

Data Formats

CSV Files

Each variable requires its own CSV file plus a metadata file. See smrf.data.csv_data for more information. The variable files must be structured as:

date_time	ID_1	ID_2	…	ID_N
10/01/2008 00:00	5.2	13.2	…	-1.3
10/01/2008 01:00	6.3	NAN	…	-2.5
…	…	…	…	…
09/30/2009 00:00	10.3	21.9	…	0.9

date_time must be chronolgical and in any format that pandas.to_datetime() can parse. Errors will occur on import when pandas cannot parse the string. The best format to use is MM-DD-YYYY HH:mm.

The column headers are the station ID numbers, which uniquely identify each station. The station ID is used throughout SMRF to filter and specify stations, as well as the metadata.

The data for each station is in the column under the station ID. Missing values can be included as either NAN or blank, which will be converted to NaN in SMRF. Missing data values will not be included in the distribution calculations.

The metadata CSV file tells SMRF important information about the location for each stations. At a minimum the metadata file must have a primary_id, X, Y, and elevation. The locations must be in UTM and the elevation is in same units as the DEM (typically meters).

primary_id	X	Y	elevation
ID_1	625406	4801625	1183
ID_2	586856	4827316	998
…	…	…	…
ID_N	641751	4846381	2310

Example data files can be found in the tests directory for RME.

MySQL Database

The MySQL database is more flexible than CSV files but requires more effort to setup. However, SMRF will only import the data and stations that were requested without loading in additional data that isn't required. See *smrf.data.mysql_data* for more information.

The data table contains all the measurement data with a single row representing a measurement time for a station. The date column (i.e. date_time) must be a DATETIME data type with a unique constraint on the date_time column and primary_id column.

date_time	primary_id	var1	var2	…	varN
10/01/2008 00:00	ID_1	5.2	13.2	…	-1.3
10/01/2008 00:00	ID_2	1.1	0	…	-10.3
10/01/2008 01:00	ID_1	6.3	NAN	…	-2.5
10/01/2008 01:00	ID_2	0.3	7.1	…	9.4

The metadata table is the same format as the CSV files, with a primary_id, X, Y, and elevation column. A benefit to using MySQL is that we can use a client as a way to group multiple stations to be used for a given model run. For example, we can have a client named BRB, which will have all the station ID's for the stations that would be used to run SMRF. Then we can specify the client in the configuration file instead of listing out all the station ID's. To use this feature, a table must be created to hold this information. Then the station ID's matching the client will only be imported. The following is how the table should be setup. Source is used to track where the data is coming from.

station_id	client	source
ID_1	BRB	Mesowest
ID_2	BRB	Mesowest
ID_3	TUOL	CDEC
…	…	…
ID_N	BRB	Mesowest

Visit the Weather Database GitHub page if you'd like to use a MySQL database.

Weather Research and Forecasting (WRF)

Gridded datasets can come in many forms and the `smrf.data.loadGrid` module is meant to import gridded datasets. Currently, SMRF can ingest WRF output in the standard wrf_out NetCDF files. SMRF looks for specific variables with the WRF output file and converts them to the related SMRF values. The grid cells are imported as if they are a single measurement station with it's own X, Y, and elevation. The minimum variables required are:

Times The date time for each timestep

XLAT Latitude of each grid cell

XLONG Longitude of each grid cell

HGT Elevation of each grid cell

T2 Air temperature at 2 meters above the surface

DWPT Dew point temperature at 2 meters above the surface, which will be used to calculate vapor pressure

GLW Incoming thermal radiation at the surface

RAINNC Accumulated precipitation

CLDFRA Cloud fraction for all atmospheric layers, the average will be used at the SMRF cloud factor

UGRD Wind vector, u component

VGRD Wind vector, v component

High Resolution Rapid Refresh (HRRR)

The High Resolution Rapid Refersh (HRRR) is a real time 3-km, hourly atmospheric model with forecasts ran by NOAA. The data is focused on recent water years (>WY2017). Loading the HRRR data into SMRF is performed by weather_forecast_retrieval based on a rigid directory structure used by the NOMADS archive. Because HHHR has a minimum of an 18 hour forecast every hour, if a data file is not found or is incomplete, `weather_forecast_retrieval` will search the previous forecasts for a good image for that specific time.

The variables used from HRRR are:

- Air temperature at 2 meters
- Relative humidity at 2 meters
- Wind u/v components at 10 meters
- Total precipitation for that hour

- Short wave radiation at the surface to calculated cloud factor
- Elevation of the terrain

Generic netCDF files

SMRF also has the ability to load generic netCDF files that may come from a variety of sources. At a minimum, the netCDF file requires at a minimum the following fields:

- `lat` for the grid cell latitude
- `lon` for the grid cell longitude
- `elev` for the grid cell elevation
- `time` CF compliant time

Each variable name is specified in the configuration file and maps from the file variable to the SMRF variable.

1.2.4 Distribution Methods

Detrending Measurement Data

Most meterological variables used in SMRF have an underlying elevational gradient. Therefore, all of the distribution methods can estimate the gradient from the measurement data and apply the elevational gradient to the DEM during distribution. Here, the theory of how the elevational gradient is calculated, removed from the data, and reapplied after distribution is explained. All the distribution methods follow this pattern and detrending can be ignored by setting `detrend: False` in the configuration.

Calculating the Elevational Trend

The elevational trend for meterological stations is calculated using all available stations in the modeling domain. A line is fit to the measurement data with the slope as the elevational gradient (Fig. 1.2a, Fig. 1.3a, and Fig. 1.4a). The slope can be constrained as positive, negative, or no constraint.

Gridded datasets have significantly more information than point measurements. Therefore, the approach is slightly different for calculating the elevational trend line. To limit the number of grid cells that contribute to the elevational trend, only those grid cells within the mask are used. This ensures that only the grid cells within the basin boundary contribute to the estimation of the elevational trend line.

Distributing the Residuals

The point measurements minus the elevational trend at the stations (or grid cell's) elevation is the measurement residual. The residuals are then distributed using the desired distribution method (Fig. 1.2b, Fig. 1.3b, and Fig. 1.4b) and show the deviance from the estimated elevational trend.

Retrending the Distributed Residuals

The distributed residuals are added to the elevational trend line evaluated at each of the DEM grid points (Fig. 1.2c, Fig. 1.3c, and Fig. 1.4c). This produces a distributed value that has the underlying elevational trend in the measurement data but also takes into account local changes in that value.

Note: Constraints can be placed on the elevational trend to be either positive, negative, or no constraint. However, if a constraint is applied and the measurement data does not fit the constraint (for example negavite trend for air temp but there is a positive trend during an inversion or night time), then the slope of the trend line will be set to zero. This will distribute the data based on the underlying method and not apply any trends.

Methods

The methods outlined below will distribute the measurement data or distribute the residuals if detrending is applied. Once the values are distributed, the values can be used as is or retrended.

Inverse Distance Weighting

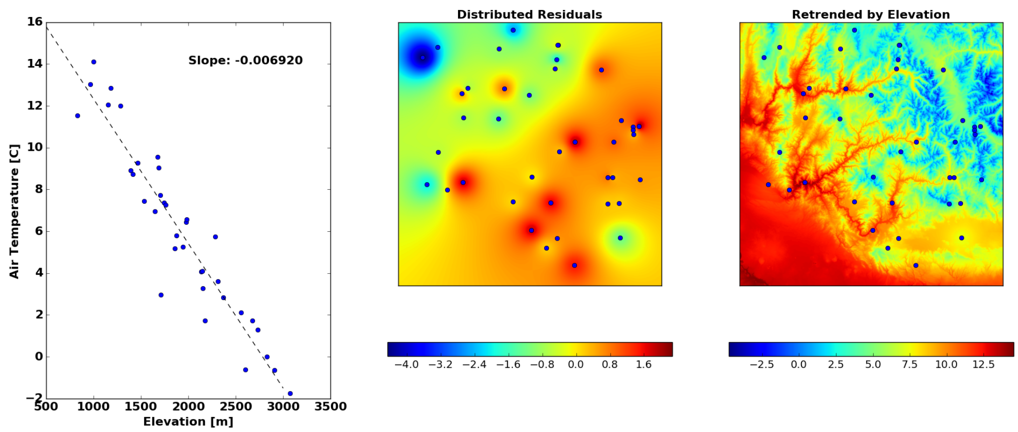


Fig. 1.2: Distribution of air temperature using inverse distance weighting. a) Air temperature as a function of elevation. b) Inverse distance weighting of the residuals. c) Retrending the residuals to the DEM elevation.

Inverse distance weighting takes the weighted average of the measurment data based on the inverse of the distance between the measurement location and the modeling grid [22]. For N set of measurement locations, the value at any x, y location can be calculated:

$$u(x,y) = \frac{\sum_{i=1}^{N} w_i(x,y)\, u_i}{\sum_{i=1}^{N} w(x,y)}$$

where

$$w_i(x,y) = \frac{1}{d_i(x,y)^p}$$

and $d_i(x,y)$ is the distance between the model grid cell and the measurement location raised to a power of p (typcially defaults to 2). The results of the inverse distance weighting, $u(x,y)$, is shown in Figure 1.2b.

Detrended Kriging

Detrended kriging is based on the work developed by Garen et al. (1994) [23].

Detrended kriging uses a model semivariogram based on the station locations to distribute the measurement data to the model domain. Before kriging can begin, a model semivariogram is developed from the measurement data that provides structure for the distribution. Given measurement data Z for N measurement points, the semivariogram $\hat{\gamma}$ is defined as:

$$\hat{\gamma}(\mathbf{h}) = \frac{1}{2m} \sum_{i=1}^{m} [z(\mathbf{x}_i) - z(\mathbf{x}_i + \mathbf{h})]^2$$

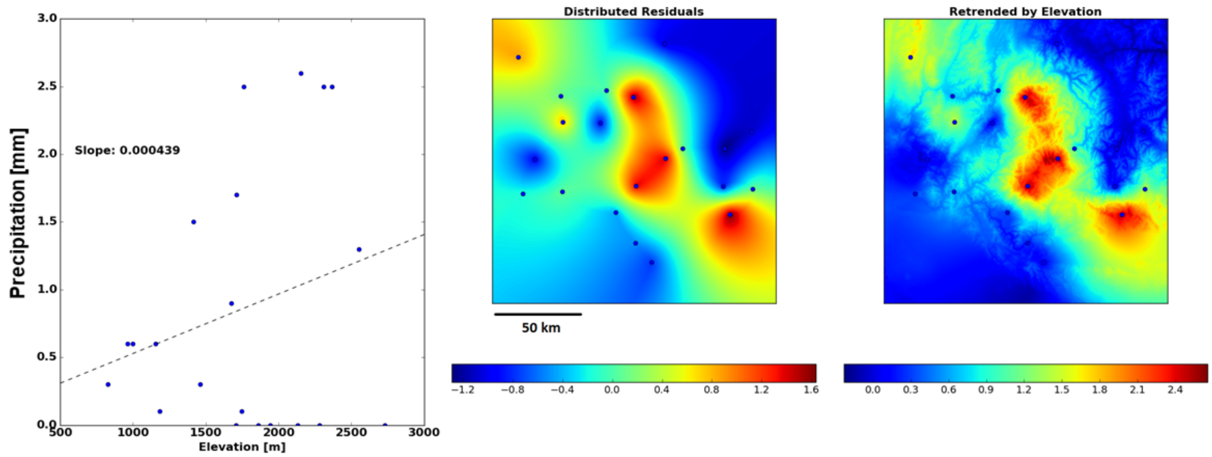


Fig. 1.3: Distribution of precipitation using detrended kriging. a) Precipitation as a function of elevation. b) Kriging of the residuals. c) Retrending the residuals to the DEM elevation.

where $\mathbf{h}$ is the seperation vector between measurement points, m is the number of points at lag $\mathbf{h}$, and $z(\mathbf{x})$ and $z(\mathbf{x}+\mathbf{h})$ represent the measurement values at locations seperated by $\mathbf{h}$. For the purposes of the detrended kriging within SMRF, m will be one as all locations will have their unique lag distance $\mathbf{h}$.

The kriging calculations require a semivariogram model to interpolate the measurement data. Detrended kriging uses a linear semivariogram $\tau(\mathbf{h}) = \tau_n + bh$ where τ_n is the nugget and b is the slope of the line. A linear semivariogram model means that on average, Z becomes increasing dissimilar at larger lag distances. With the linear semivariogram model, ordinary kriging methods are used to calculate the weights at each point through solving of a system of linear equations with the constraint of the weights summing to 1. See Garen et al. (1994) [23] or [24] for a review of oridinary kriging methods.

In this implementation of detrended kriging, simplifications are made based on the use of the linear semivariogram. With a linear semivariogram, the kriging weights are independent of the slope and nugget of the model, as the semivariogram is a function of only the lag distance. Therefore, this assumption simplifies the kriging weight calculations as $\hat{\gamma}(\mathbf{h}) = h$. There the weights only need to be calculated once when the current set of measurement locations change. The kriging weights are futher constrained to only use stations that are within close proximity to the estimation point.

Ordinary Kriging

Detrended kriging above is a specific application of ordinary kriging for distributing meterological data. A more generic kriging approach is to use PyKrige that supports 2D ordinary and universal kriging. See PyKrige documentation for more information and the *configuration file reference* for specific application within SMRF.

Gridded Interpolation

Gridded interpolation was developed for gridded datasets that have orders of magnitude more data than station measurements (i.e. 3000 grid points for a gridded forecast). This ensures that the computions required for inverse distance weighting or detrended kriging are not performed to save memory and computational time. The interpolation uses `scipy.interpolate.griddata` (documentation here) to interpolate the values to the model domain. Four different interpolation methods can be used:

- linear (default)
- nearest neighbor

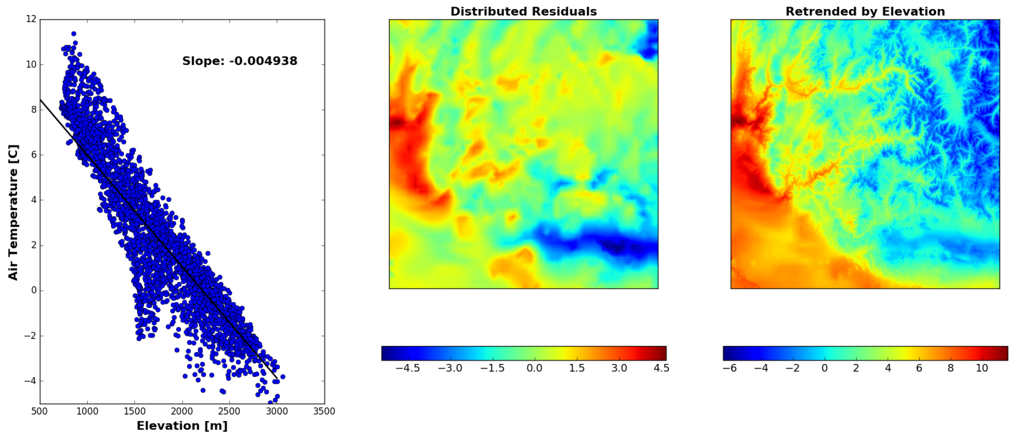


Fig. 1.4: Distribution of air temperature using gridded interpolation. a) Air temperature as a function of elevation. b) Linear interpolation of the residuals. c) Retrending the residuals to the DEM elevation.

- cubic 1-D
- cubic 2-D

1.2.5 Wind Models

Three wind distribution methods are available for the Wind class:

1. Winstral and Marks 2002 method for maximum upwind slope (maxus)
2. Import WindNinja simulations
3. Standard interpolation

The type of wind model can be specfied in the `[wind]` section of the configuration file. The options are `winstral`, `wind_ninja`, and `interp`.

Wind Measurement Height

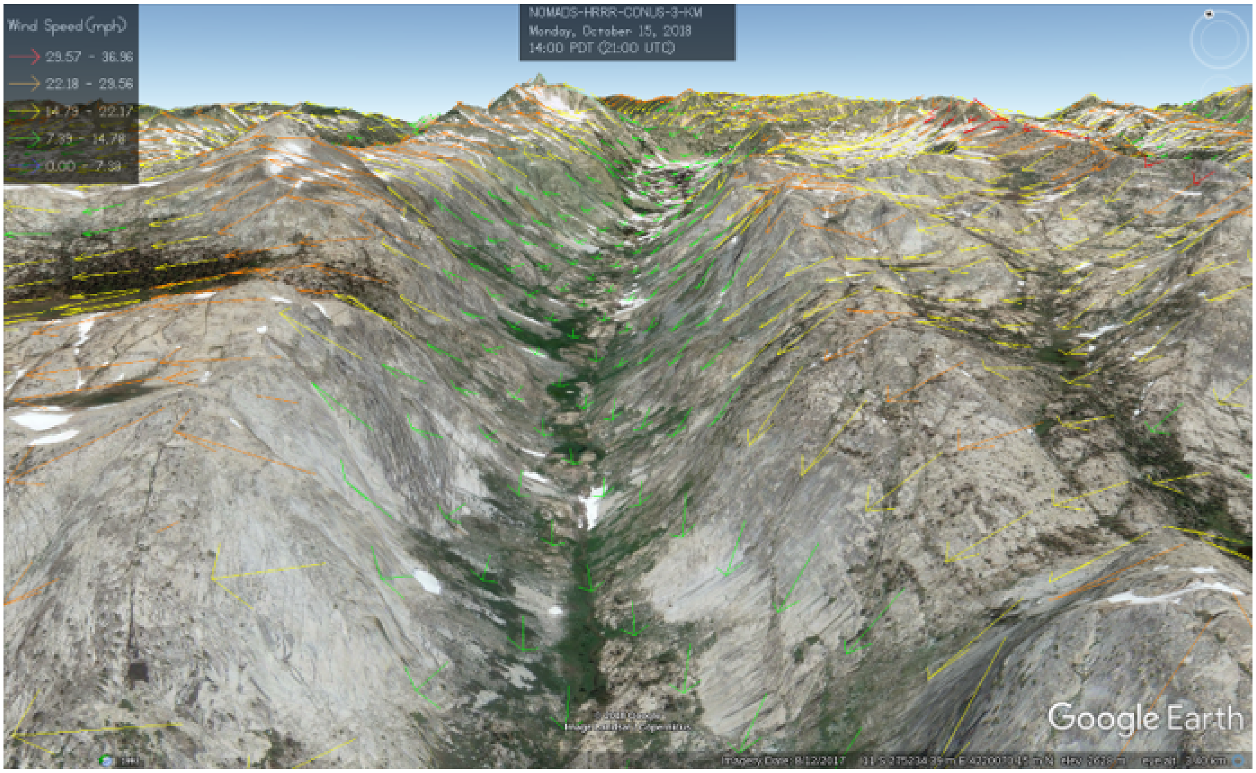
This paragraph aims to be a cautionary note. Wind is an important driver for the turbulent fluxes in a snow or hydrological model. However, atmospheric models and wind measurements can be taken at a multitude of different heights. For example, with WindNinja, HRRR outputs at 10 meters above the ground surface. WindNinja can then scale that to a different height (default is 5 meters in `katana`). Therefore, the wind speed that SMRF interpolates will be at 5 meters and that should match the measurement height of wind in the snow or hydrology model.

The same is true with wind speed measurements as not all sensors are placed at the same height. Take care to review the weather station metadata and convert the wind speed to fixed height above the ground before using SMRF.

Note: Check the wind measurement and output heights for consistency.

WindNinja

WindNinja simulates wind over complex terrain using a computational fluid dynamics approach and was originally developed for wildland fire forecasting. WindNinja includes a conservation of mass and a conservation of mass and momentum solver, implemented using OpenFOAM.



WindNinja has been built into a Docker image in katana that performs the WindNinja simulations. The function of Katana is to deal with the data editing and data flow required to run WindNinja over large areas and long periods of time, as well as actually running WindNinja. The power of Katana is that it organizes all of the necessary software (WindNinja, GDAL, wgrib2) into an easy to use docker.

The steps that Katana takes are as follows:

1. Create topo ascii for use in WindNinja
2. Crop grib2 files to a small enough domain to actually run WindNinja as we cannot read in the full CONUS domain
3. Extract the necessary variables from the grib2 files
4. Organize the new, smaller grib2 files into daily folders
5. Create WindNinja config
6. Run WindNinja (one run per day)

Note: `katana` must be ran prior to SMRF.

SMRF reads the ASCII WindNinja outputs and interpolates (if needed) to the model domain. See the `katana` README for more information on how to setup the configuration file and what atmospheric models it can run. For SMRF, WindNinja is ran as if the vegetation was grass.

Once WindNinja simulations have been performed, SMRF applies a log law scaling to adjust the simulated wind field for the roughness of the vegetation height. The vegetation roughness in the log law scaling is based on Brutsaert (1974) [25] and Cataldo and Zeballos (2009) [26].

Winstral Wind Model

The methods described here follow the work developed by Winstral and Marks (2002) and Winstral et al. (2009) [19] [20] which parametrizes the terrain based on the upwind direction. The underlying method calculates the maximum upwind slope (maxus) within a search distance to determine if a cell is sheltered or exposed. See `smrf.utils.wind.model` for a more in depth description. A maxus file (library) is used to load the upwind direction and maxus values over the dem. The following steps are performed when estimating the wind speed:

1. Calculate the maxus values for the topo
2. Adjust measured wind speeds at the stations and determine the wind direction components
3. Distribute the flat wind speed and wind direction components
4. Simulate the wind speeds based on the distribute flat wind, wind direction, and maxus values

Note: Winstral wind model works only with station measurements and not atmospheric models.

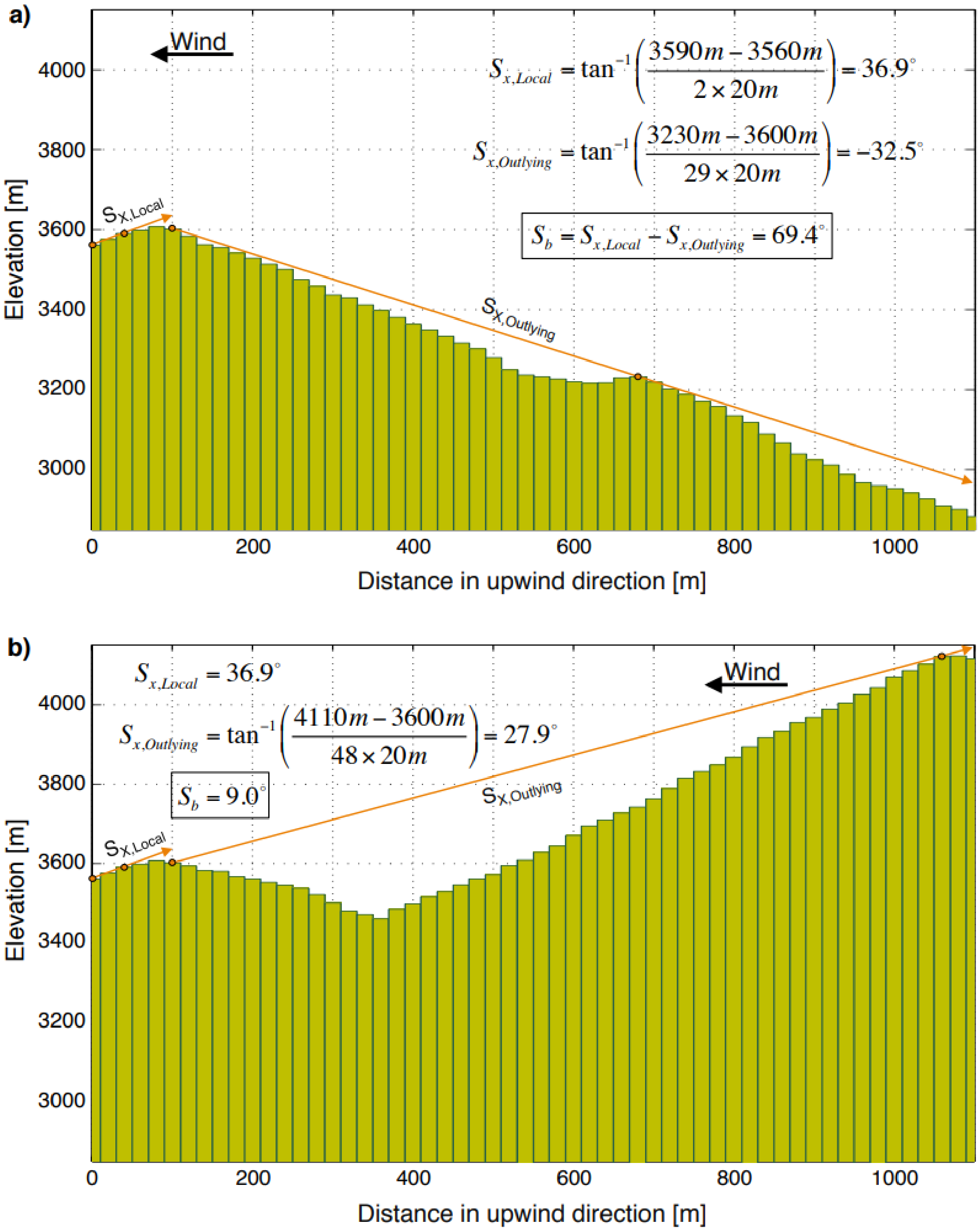
1. Maxus values

The azimuth **A** is the direction of the prevailing wind for which the maxus value will be calculated within a maximum search distance **dmax**. The maxus (**Sx**) parameter can then be estimated as the maximum value of the slope from the cell of interest to all of the grid cells along the search vector. The efficiency in selection of the maximum value can be increased by using the techniques from the horizon function which calculates the horizon for each pixel. Therefore, less calculations can be performed. Negative **Sx** values indicate an exposed pixel location (shelter pixel was lower) and positive **Sx** values indicate a sheltered pixel (shelter pixel was higher).

After all the upwind direction are calculated, the average **Sx** over a window is calculated. The average **Sx** accounts for larger landscape obsticles that may be adjacent to the upwind direction and affect the flow. A window size in degrees takes the average of all **Sx**.

2. Adjust measured wind speeds

After the maxus is calculated for multiple wind directions over the entire DEM, the measured wind speed and direction can be distributed. The first step is to adjust the measured wind speeds to estimate the wind speed if the site were on a flat surface. The adjustment uses the maxus value at the station location and an enhancement factor for the site based on the sheltering of that site to wind. A special consideration is performed when the station is on a peak, as artificially high wind speeds can be calculated. If the station is on a peak, the minimum maxus value is chosen for all wind directions. The wind direction is then broken up into the u,v components.


a)
Wind
$S_{x,Local} = \tan^{-1}\left(\frac{3590m - 3560m}{2 \times 20m}\right) = 36.9^{\circ}$
$S_{x,Outlying} = \tan^{-1}\left(\frac{3230m - 3600m}{29 \times 20m}\right) = -32.5^{\circ}$
$S_b = S_{x,Local} - S_{x,Outlying} = 69.4^{\circ}$
$S_{X,Local}$
$S_{X,Outlying}$
Elevation [m]
4000
3800
3600
3400
3200
3000
0
200
400
600
800
1000
Distance in upwind direction [m]
b)
$S_{x,Local} = 36.9^{\circ}$
$S_{x,Outlying} = \tan^{-1}\left(\frac{4110m - 3600m}{48 \times 20m}\right) = 27.9^{\circ}$
$S_b = 9.0^{\circ}$
Wind
$S_{X,Local}$
$S_{X,Outlying}$
Elevation [m]
4000
3800
3600
3400
3200
3000
0
200
400
600
800
1000
Distance in upwind direction [m]

3. Distribute flat wind speed and direction

Next the flat wind speed, u wind direction component, and v wind direction component are distributed using the underlying SMRF distribution methods.

4. Simulate wind speed with maxus

With the distributed flat wind speed and wind direction, the simulated wind speeds can be estimated. The distributed wind direction is binned into the upwind directions in the maxus library. This determines which maxus value to use for each pixel in the DEM. Each cell's maxus value is further enhanced for vegetation, with larger, more dense vegetation increasing the maxus value (more sheltering) and bare ground not enhancing the maxus value (exposed). With the adjusted maxus values, wind speed is estimated using the relationships in Winstral and Marks (2002) and Winstral et al. (2009) [19] [20] based on the distributed flat wind speed and each cell's maxus value.

Standard interpolation

Standard interpolation using SMRF's *distribution methods*.

1.3 API Documentation

Everything you could ever want to know about SMRF.

1.3.1 smrf.data package

Submodules

smrf.data.loadData module

class `smrf.data.loadData.`**wxdata**(*dataConfig*, *start_date*, *end_date*, *time_zone='UTC'*, *dataType=None*)

Bases: `object`

Class for loading and storing the data, either from - CSV file - MySQL database - Add other sources here

Inputs to data() are: - dataConfig, either the [csv] or [mysql] section - start_date, datetime object - end_date, datetime object - dataType, either 'csv' or 'mysql'

The data will be loaded into a Pandas dataframe

db_config_vars = ['user', 'password', 'host', 'database', 'port', 'metadata', 'data_ta

load_from_csv()

Load the data from a csv file Fields that are operated on - metadata -> dictionary, one for each station, must have at least the following: primary_id, X, Y, elevation - csv data files -> dictionary, one for each time step, must have at least the following columns: date_time, column names matching metadata.primary_id

load_from_mysql()

Load the data from a mysql database

variables = ['air_temp', 'vapor_pressure', 'precip', 'wind_speed', 'wind_direction', '

smrf.data.loadGrid module

smrf.data.loadGrid.**apply_utm**(*s*, *force_zone_number*)

Calculate the utm from lat/lon for a series

Parameters

- **s** – pandas series with fields latitude and longitude
- **force_zone_number** – default None, zone number to force to

Returns pandas series with fields 'X' and 'Y' filled

Return type s

class smrf.data.loadGrid.**grid**(*dataConfig*, *topo*, *start_date*, *end_date*, *time_zone='UTC'*, *dataType='wrf'*, *tempDir=None*, *forecast_flag=False*, *day_hour=0*, *n_forecast_hours=18*)

Bases: object

Class for loading and storing the data, either from a gridded dataset in: - NetCDF format - other format

Inputs to data() are: - dataConfig, from the [gridded] section - start_date, datetime object - end_date, datetime object

get_latlon(*utm_x*, *utm_y*)

Convert UTM coords to Latitude and longitude

Parameters

- **utm_x** – UTM easting in meters in the same zone/letter as the topo
- **utm_y** – UTM Northing in meters in the same zone/letter as the topo

Returns

(lat,lon) latitude and longitude conversion from the UTM coordinates

Return type tuple

load_from_hrrr()

Load the data from the High Resolution Rapid Refresh (HRRR) model The variables returned from the HRRR class in dataframes are

- metadata
- air_temp
- relative_humidity
- precip_int
- cloud_factor
- wind_u
- wind_v

The function will take the keys and load them into the appropriate objects within the *grid* class. The vapor pressure will be calculated from the *air_temp* and *relative_humidity*. The *wind_speed* and *wind_direction* will be calculated from *wind_u* and *wind_v*

load_from_netcdf()

Load the data from a generic netcdf file

Parameters

- **lat** – latitude field in file, 1D array

- **lon** – longitude field in file, 1D array
- **elev** – elevation field in file, 2D array
- **variable** – variable name in file, 3D array

load_from_wrf()

Load the data from a netcdf file. This was setup to work with a WRF output file, i.e. wrf_out so it's going to look for the following variables: - Times - XLAT - XLONG - HGT - T2 - DWPT - GLW - RAINNC - CLDFRA - UGRD - VGRD

Each cell will be identified by grid_IX_IY

model_domain_grid()

Retrieve the bounding box for the gridded data by adding a buffer to the extents of the topo domain.

Returns (dlat, dlon) Domain latitude and longitude extents

Return type tuple

smrf.data.loadTopo module

class smrf.data.loadTopo.**Topo**(*topoConfig*, *calcInput=True*, *tempDir=None*)

Bases: object

Class for topo images and processing those images. Images are: - DEM - Mask - veg type - veg height - veg k - veg tau

Inputs to topo are the topo section of the config file topo will guess the location of the WORKDIR env variable and should work for unix systems.

topoConfig

configuration for topo

tempDir

location of temporary working directory

dem

numpy array for the DEM

mask

numpy array for the mask

veg_type

numpy array for the veg type

veg_height

numpy array for the veg height

veg_k

numpy array for the veg K

veg_tau

numpy array for the veg transmissivity

sky_view

ny

number of columns in DEM

nx

number of rows in DEM

u,v
: location of upper left corner

du, dv
: step size of grid

unit
: geo header units of grid

coord_sys_ID
: coordinate syste,

x,y
: position vectors

X,Y
: position grid

stoporad_in
: numpy array for the sky view factor

IMAGES = ['dem', 'mask', 'veg_type', 'veg_height', 'veg_k', 'veg_tau']

get_center(*ds*, *mask_name=None*)
: Function returns the basin center in the native coordinates of the a netcdf object.

The incoming data set must contain at least and x, y and optionally whatever mask name the user would like to use for calculating . If no mask name is provided then the entire domain is used.

Parameters

- **ds** – netCDF4.Dataset object containing at least x,y, optionally a mask variable name
- **mask_name** – variable name in the dataset that is a mask where 1 is in the mask

Returns x,y of the data center in the datas native coordinates

Return type tuple

gradient(*gfile*)
: Calculate the gradient and aspect

Parameters **gfile** – IPW file to write the results to

readNetCDF()
: Read in the images from the config file where the file listed is in netcdf format

stoporadInput()
: Calculate the necessary input file for stoporad The IPW and WORKDIR environment variables must be set

smrf.data.mysql_data module

Created on Dec 22, 2015

Read in metadata and data from a MySQL database The table columns will most likely be hardcoded for ease of development and users will require the specific table setup

class smrf.data.mysql_data.**database**(*user*, *password*, *host*, *db*, *port*)
: Bases: object

Database class for querying metadata and station data

get_data (*table*, *station_ids*, *start_date*, *end_date*, *variables*, *time_zone='UTC'*)
Get data from the database, either for the specified stations or for the specific group of stations in client

Parameters

- **table** – table to load data from
- **station_ids** – list of station ids to get
- **start_date** – start of time period
- **end_date** – end of time period
- **variable** – string for variable to get
- **time_zone** – String timezone to set the data in

metadata (*table*, *station_ids=None*, *client=None*, *station_table=None*)
Similar to the CorrectWxData database call Get the metadata from the database for either the specified stations or for the specific group of stations in client

Parameters

- **table** – metadata table in the database
- **station_id** – list of stations to read, default None
- **client** – client to read from the station_table, default None
- **station_table** – table name that contains the clients and list of stations, default None

Returns Pandas DataFrame of station information

Return type d

query (*query*, *params*)

smrf.data.mysql_data.**date_range** (*start_date*, *end_date*, *increment*)
Calculate a list between start and end date with an increment

Module contents

1.3.2 smrf.distribute package

Subpackages

smrf.distribute.wind package

Submodules

smrf.distribute.wind.wind module

class smrf.distribute.wind.wind.**Wind** (*config*)
Bases: *smrf.distribute.image_data.image_data*

The *wind* class allows for variable specific distributions that go beyond the base class.

Three distribution methods are available for the Wind class:

1. Winstral and Marks 2002 method for maximum upwind slope (maxus)
2. Import WindNinja simulations

3. Standard interpolation

   Parameters **self.config** – The full SMRF configuration file

config
configuration from [wind] section

wind_speed
numpy matrix of the wind speed

wind_direction
numpy matrix of the wind direction

veg_type
numpy array for the veg type, from *smrf.data.loadTopo.Topo.veg_type*

_maxus_file
the location of the maxus NetCDF file

maxus
the loaded library values from *_maxus_file*

maxus_direction
the directions associated with the *maxus* values

min
minimum value of wind is 0.447

max
maximum value of wind is 35

stations
stations to be used in alphabetical order

VARIABLE = 'wind'

distribute(*data_speed*, *data_direction*, *t*)
Distribute given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute` for the *wind_model* chosen.

Parameters

- **data_speed** – Pandas dataframe for single time step from wind_speed
- **data_direction** – Pandas dataframe for single time step from wind_direction
- **t** – time stamp

distribute_thread(*queue*, *data_speed*, *data_direction*)
Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call `smrf.distribute.wind.wind.distribute` then puts the distributed data into the queue for *wind_speed*.

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

initialize(*topo*, *data*)
Initialize the distribution, calls `smrf.distribute.image_data.image_data._initialize`. Checks for the enhancement factors for the stations and vegetation.

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **data** – data Pandas dataframe containing the station data, from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'flatwind': {'long_name': 'Simulated wind on a flat surface', 's

post_process_variables = {}

smrf.distribute.wind.wind_ninja module

class smrf.distribute.wind.wind_ninja.**WindNinjaModel**(*smrf_config*)
Bases: *smrf.distribute.image_data.image_data*

The *WindNinjaModel* loads data from a WindNinja simulation. The WindNinja is ran externally to SMRF and the configuration points to the location of the output ascii files. SMRF takes the files and interpolates to the model domain.

DATE_FORMAT = '%Y%m%d'

VARIABLE = 'wind'

WN_DATE_FORMAT = '%m-%d-%Y_%H%M'

convert_wind_ninja(*t*)
Convert the WindNinja ascii grids back to the SMRF grids and into the SMRF data streamself.

Parameters t – datetime of timestep

Returns wind speed numpy array wd: wind direction numpy array

Return type ws

distribute(*data_speed*, *data_direction*)
Distribute the wind for the model

Parameters

- **{DataFrame} -- wind speed data frame** (*data_speed*) –
- **{DataFrame} -- wind direction data frame** (*data_direction*) –

fill_data(*g_vel*)
Fill the WindNinja array that has NaN's. This makes an assumption that all the NaN values are along the left and bottom edge. This will be the case in the Northern hemisphere. First fill the Y direction with 1d interpolation exprapolated to the edges, then do the same in the X direction. At the end, it will check to ensure that there are no NaN values left.

Parameters {np.array} -- numpy array to fill (*g_vel*) –

Raises ValueError – If there are still NaN values after filling

Returns np.array – filled numpy array

initialize(*topo*, *data=None*)
Initialize the model with data

Parameters

- **{topo class} -- Topo class** (*topo*) –
- **{None} -- Not used but needs to be there** (*data*) –

initialize_interp(*t*)
Initialize the interpolation weights

Parameters **{datetime} -- initialize with this file** (*t*) –

wind_ninja_path(*dt*, *file_type*)
Generate the path to the wind ninja data and ensure it exists.

Parameters **{str} -- type of file to get** (*file_type*) –

smrf.distribute.wind.wind_ninja.**interpx**(*yi*, *xi*, *x*)
Interpolate in on direction

Parameters

- **{array} -- y data to fit** (*yi*) –
- **{array} -- x data to fit** (*xi*) –
- **{array} -- x data to interpolate over** (*x*) –

Returns array – y values evaluated at x

smrf.distribute.wind.winstral module

class smrf.distribute.wind.winstral.**WinstralWindModel**(*smrf_config*)
Bases: *smrf.distribute.image_data.image_data*

Estimating wind speed and direction is complex terrain can be difficult due to the interaction of the local topography with the wind. The methods described here follow the work developed by Winstral and Marks (2002) and Winstral et al. (2009) [19] [20] which parameterizes the terrain based on the upwind direction. The underlying method calulates the maximum upwind slope (maxus) within a search distance to determine if a cell is sheltered or exposed. See *smrf.utils.wind.model* for a more in depth description. A maxus file (library) is used to load the upwind direction and maxus values over the dem. The following steps are performed when estimating the wind speed:

1. Adjust measured wind speeds at the stations and determine the wind direction componenets
2. Distribute the flat wind speed
3. Distribute the wind direction components
4. Simulate the wind speeds based on the distribute flat wind, wind direction, and maxus values

After the maxus is calculated for multiple wind directions over the entire DEM, the measured wind speed and direction can be distirbuted. The first step is to adjust the measured wind speeds to estimate the wind speed if the site were on a flat surface. The adjustment uses the maxus value at the station location and an enhancement factor for the site based on the sheltering of that site to wind. A special consideration is performed when the station is on a peak, as artificially high wind speeds can be calcualted. Therefore, if the station is on a peak, the minimum maxus value is choosen for all wind directions. The wind direction is also broken up into the u,v componenets.

Next the flat wind speed, u wind direction component, and v wind direction compoenent are distributed using the underlying distribution methods. With the distributed flat wind speed and wind direction, the simulated wind speeds can be estimated. The distributed wind direction is binned into the upwind directions in the maxus library. This determines which maxus value to use for each pixel in the DEM. Each cell's maxus value is further enhanced for vegetation, with larger, more dense vegetation increasing the maxus value (more sheltering) and bare ground not enhancing the maxus value (exposed). With the adjusted maxus values, wind speed is estimated using the relationships in Winstral and Marks (2002) and Winstral et al. (2009) [19] [20] based on the distributed flat wind speed and each cell's maxus value.

VARIABLE = 'wind'

distribute (*data_speed*, *data_direction*)

Distribute the wind for the model

Follows the following steps for station measurements:

1. **Adjust measured wind speeds at the stations and determine the wind** direction componenets
2. Distribute the flat wind speed
3. Distribute the wind direction components
4. **Simulate the wind speeds based on the distribute flat wind, wind** direction, and maxus values

Parameters

- **{DataFrame} -- wind speed data frame** (*data_speed*) –
- **{DataFrame} -- wind direction data frame** (*data_direction*) –

initialize (*topo*, *data*)

Initialize the model with data

Parameters

- **{topo class} -- Topo class** (*topo*) –
- **{data object} -- SMRF data object** (*data*) –

simulateWind (*data_speed*)

Calculate the simulated wind speed at each cell from flatwind and the distributed directions. Each cell's maxus value is pulled from the maxus library based on the distributed wind direction. The cell's maxus is further adjusted based on the vegetation type and the factors provided in the [wind] section of the configuration file.

Parameters **data_speed** – Pandas dataframe for a single time step of wind speed to make the pixel locations same as the measured values

stationMaxus (*data_speed*, *data_direction*)

Determine the maxus value at the station given the wind direction. Can specify the enhancemet for each station or use the default, along with whether or not the station is on a peak which will ensure that the station cannot be sheltered. The station enhancement and peak stations are specified in the [wind] section of the configuration file. Calculates the following for each station:

- `flatwind`
- `u_direction`
- `v_direction`

Parameters

- **data_speed** – wind_speed data frame for single time step
- **data_direction** – wind_direction data frame for single time step

Module contents

Submodules

smrf.distribute.air_temp module

class smrf.distribute.air_temp.**ta**(*taConfig*)
Bases: *smrf.distribute.image_data.image_data*

The *ta* class allows for variable specific distributions that go beyond the base class.

Air temperature is a relatively simple variable to distribute as it does not rely on any other variables, but has many variables that depend on it. Air temperature typically has a negative trend with elevation and performs best when detrended. However, even with a negative trend, it is possible to have instances where the trend does not apply, for example a temperature inversion or cold air pooling. These types of conditions will have unintended consequences on variables that use the distributed air temperature.

Parameters **taConfig** – The [air_temp] section of the configuration file

config
configuration from [air_temp] section

air_temp
numpy array of the air temperature

stations
stations to be used in alphabetical order

distribute(*data*)
Distribute air temperature given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

Parameters **data** – Pandas dataframe for a single time step from air_temp

distribute_thread(*queue*, *data*)
Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call *smrf.distribute.air_temp.ta.distribute* then puts the distributed data into `queue['air_temp']`.

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

initialize(*topo*, *data*)
Initialize the distribution, solely calls `smrf.distribute.image_data.image_data._initialize`.

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **metadata** – metadata Pandas dataframe containing the station metadata from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'air_temp': {'long_name': 'Air temperature', 'standard_name': '

post_process_variables = {}

variable = 'air_temp'

smrf.distribute.albedo module

class smrf.distribute.albedo.**albedo**(*albedoConfig*)

Bases: *smrf.distribute.image_data.image_data*

The *albedo* class allows for variable specific distributions that go beyond the base class.

The visible (280-700nm) and infrared (700-2800nm) albedo follows the relationships described in Marks et al. (1992) [1]. The albedo is a function of the time since last storm, the solar zenith angle, and grain size. The time since last storm is tracked on a pixel by pixel basis and is based on where there is significant accumulated distributed precipitation. This allows for storms to only affect a small part of the basin and have the albedo decay at different rates for each pixel.

Parameters **albedoConfig** – The [albedo] section of the configuration file

albedo_vis

numpy array of the visible albedo

albedo_ir

numpy array of the infrared albedo

config

configuration from [albedo] section

min

minimum value of albedo is 0

max

maximum value of albedo is 1

stations

stations to be used in alphabetical order

distribute(*current_time_step*, *cosz*, *storm_day*)

Distribute air temperature given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

Parameters

- **current_time_step** – Current time step in datetime object
- **cosz** – numpy array of the illumination angle for the current time step
- **storm_day** – numpy array of the decimal days since it last snowed at a grid cell

distribute_thread(*queue*, *date*)

Distribute the data using threading and queue

Parameters

- **queue** – queue dict for all variables
- **date** – dates to loop over

Output:

Changes the queue albedo_vis, albedo_ir for the given date

initialize(*topo*, *data*)

Initialize the distribution, calls image_data.image_data._initialize()

Parameters

- **topo** – smrf.data.loadTopo.Topo instance contain topo data/info

- **data** – data dataframe containing the station data

output_variables = {'albedo_ir': {'long_name': 'Infrared wavelength albedo', 'standa

post_process_variables = {}

variable = 'albedo'

smrf.distribute.cloud_factor module

class smrf.distribute.cloud_factor.**cf**(*config*)

Bases: *smrf.distribute.image_data.image_data*

The *cf* class allows for variable specific distributions that go beyond the base class. Cloud factor is a relatively simple variable to distribute as it does not rely on any other variables.

Cloud factor is calculated as the ratio between measured incoming solar radiation and modeled clear sky radiation. A value of 0 means no incoming solar radiation (or very cloudy) and a value of 1 means sunny.

Parameters **config** – The [cloud_factor] section of the configuration file

config
configuration from [cloud_factor] section

cloud_factor
numpy array of the cloud factor

stations
stations to be used in alphabetical order

distribute(*data*)
Distribute cloud factor given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

Parameters **data** – Pandas dataframe for a single time step from cloud_factor

distribute_thread(*queue, data*)
Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call *smrf.distribute.cloud_factor.cf.distribute* then puts the distributed data into `queue['cloud_factor']`.

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

initialize(*topo, data*)
Initialize the distribution, solely calls `smrf.distribute.image_data.image_data._initialize`.

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **metadata** – metadata Pandas dataframe containing the station metadata from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'cloud_factor': {'long_name': 'cloud factor', 'standard_name':

post_process_variables = {}

variable = 'cloud_factor'

smrf.distribute.image_data module

class smrf.distribute.image_data.**image_data**(*variable*)

Bases: object

A base distribution method in SMRF that will ensure all variables are distributed in the same manner. Other classes will be initialized using this base class.

```
class ta(smrf.distribute.image_data):
    '''
    This is the ta class extending the image_data base class
    '''
```

Parameters variable (*str*) – Variable name for the class

Returns A smrf.distribute.image_data class instance

variable
The name of the variable that this class will become

[variable_name]
The *variable* will have the distributed data

[other_attribute]
The distributed data can also be stored as another attribute specified in _distribute

config
Parsed dictionary from the configuration file for the variable

stations
The stations to be used for the variable, if set, in alphabetical order

metadata
The metadata Pandas dataframe containing the station information from *smrf.data.loadData* or *smrf.data.loadGrid*

idw
Inverse distance weighting instance from *smrf.spatial.idw.IDW*

dk
Detrended kriging instance from *smrf.spatial.dk.dk.DK*

grid
Gridded interpolation instance from *smrf.spatial.grid.GRID*

getConfig(*cfg*)
Check the configuration that was set by the user for the variable that extended this class. Checks for standard distribution parameters that are common across all variables and assigns to the class instance. Sets the *config* and *stations* attributes.

Parameters cfg (*dict*) – dict from the [variable]

getStations(*config*)
Determines the stations from the [variable] section of the configuration file.

Parameters config (*dict*) – dict from the [variable]

post_processor(*output_func*)
Each distributed variable has the oppurtunity to do post processing on a sub variable. This is necessary in cases where the post proecessing might need to be done on a different timescale than that of the main loop.

Should be redefined in the individual variable module.

smrf.distribute.precipitation module

class smrf.distribute.precipitation.**ppt**(*pptConfig*, *start_date*, *time_step=60*)

Bases: *smrf.distribute.image_data.image_data*

The ppt class allows for variable specific distributions that go beyond the base class.

The instantaneous precipitation typically has a positive trend with elevation due to orographic effects. However, the precipitation distribution can be further complicated for storms that have isolated impact at only a few measurement locations, for example thunderstorms or small precipitation events. Some distribution methods may be better suited than others for capturing the trend of these small events with multiple stations that record no precipitation may have a negative impact on the distribution.

The precipitation phase, or the amount of precipitation falling as rain or snow, can significantly alter the energy and mass balance of the snowpack, either leading to snow accumulation or inducing melt [2] [3]. The precipitation phase and initial snow density estimated using a variety of models that can be set in the configuration file.

For more information on the available models, checkout *snow*.

After the precipitation phase is calculated, the storm information can be determined. The spatial resolution for which storm definitions are applied is based on the snow model thats selected.

The time since last storm is based on an accumulated precipitation mass threshold, the time elapsed since it last snowed, and the precipitation phase. These factors determine the start and end time of a storm that has produced enough precipitation as snow to change the surface albedo.

Parameters

- **pptConfig** – The [precip] section of the configuration file
- **time_step** – The time step in minutes of the data, defaults to 60

config
configuration from [precip] section

precip
numpy array of the precipitation

percent_snow
numpy array of the percent of time step that was snow

snow_density
numpy array of the snow density

storm_days
numpy array of the days since last storm

storm_total
numpy array of the precipitation mass for the storm

last_storm_day
numpy array of the day of the last storm (decimal day)

last_storm_day_basin
maximum value of last_storm day within the mask if specified

min
minimum value of precipitation is 0

max
maximum value of precipitation is infinite

stations
stations to be used in alphabetical order

distribute(*data*, *dpt*, *precip_temp*, *ta*, *time*, *wind*, *temp*, *az*, *dir_round_cell*, *wind_speed*, *cell_maxus*, *mask=None*)
Distribute given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

The following steps are taken when distributing precip, if there is precipitation measured:

1. Distribute the instantaneous precipitation from the measurement data
2. **Determine the distributed precipitation phase based on the** precipitation temperature
3. **Calculate the storms based on the accumulated mass, time since last** storm, and precipitation phase threshold

Parameters

- **data** – Pandas dataframe for a single time step from precip
- **dpt** – dew point numpy array that will be used for
- **precip_temp** – numpy array of the precipitation temperature
- **ta** – air temp numpy array
- **time** – pass in the time were are currently on
- **wind** – station wind speed at time step
- **temp** – station air temperature at time step
- **az** – numpy array for simulated wind direction
- **dir_round_cell** – numpy array for wind direction in discreet increments for referencing maxus at a specific direction
- **wind_speed** – numpy array of wind speed
- **cell_maxus** – numpy array for maxus at correct wind directions
- **mask** – basin mask to apply to the storm days for calculating the last storm day for the basin

distribute_for_marks2017(*data*, *precip_temp*, *ta*, *time*, *mask=None*)
Specialized distribute function for working with the new accumulated snow density model Marks2017 requires storm total and a corrected precipitation as to avoid precip between storms.

distribute_for_susong1999(*data*, *ppt_temp*, *time*, *mask=None*)
Susong 1999 estimates percent snow and snow density based on Susong et al, (1999) [4].

Parameters

- **data** (*pd.DataFrame*) – Precipitation mass data
- **ppt_temp** (*pd.DataFrame*) – Precipitation temperature data
- **time** – Unused
- **mask** (*np.array, optional*) – Mask the output. Defaults to None.

distribute_thread(*queue*, *data*, *date*, *mask=None*)
Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call `smrf.distribute.precip.ppt.distribute` then puts the distributed data into the queue for:

- *percent_snow*
- *snow_density*
- *storm_days*
- *last_storm_day_basin*

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

initialize(*topo*, *data*)

output_variables = {'last_storm_day': {'long_name': 'Decimal day of the last storm s

post_process_variables = {}

post_processor(*main_obj*, *threaded=False*)
Each distributed variable has the oppurtunity to do post processing on a sub variable. This is necessary in cases where the post proecessing might need to be done on a different timescale than that of the main loop.

Should be redefined in the individual variable module.

post_processor_threaded(*main_obj*)

variable = 'precip'

smrf.distribute.soil_temp module

class smrf.distribute.soil_temp.**ts**(*soilConfig*, *tempDir=None*)
Bases: *smrf.distribute.image_data.image_data*

The *ts* class allows for variable specific distributions that go beyond the base class.

Soil temperature is simply set to a constant value during initialization. If soil temperature measurements are available, the values can be distributed using the distribution methods.

Parameters

- **soilConfig** – The [soil] section of the configuration file
- **tempDir** – location of temp/working directory (default=None)

config
configuration from [soil] section

soil_temp
numpy array of the soil temperature

stations
stations to be used in alphabetical order

distribute()
No distribution is performed on soil temperature at the moment, method simply passes.

Parameters None –

initialize(*topo*, *data*)
Initialize the distribution and set the soil temperature to a constant value based on the configuration file.

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **metadata** – data Pandas dataframe containing the station data, from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'soil_temp': {'long_name': 'Soil temperature', 'standard_name':

post_process_variables = {}

variable = 'soil_temp'

smrf.distribute.solar module

class smrf.distribute.solar.**solar**(*config*, *stoporad_in*, *tempDir=None*)

Bases: *smrf.distribute.image_data.image_data*

The *solar* class allows for variable specific distributions that go beyond the base class.

Multiple steps are required to estimate solar radiation:

1. Terrain corrected clear sky radiation
2. Adjust solar radiation for vegetation effects
3. Calculate net radiation using the albedo

The Image Processing Workbench (IPW) includes a utility `stoporad` to model terrain corrected clear sky radiation over the DEM. Within `stoporad`, the radiation transfer model `twostream` simulates the clear sky radiation on a flat surface for a range of wavelengths through the atmosphere [5] [6] [7]. Terrain correction using the DEM adjusts for terrain shading and splits the clear sky radiation into beam and diffuse radiation.

The second step requires sites measuring solar radiation. The measured solar radiation is compared to the modeled clear sky radiation from `twostream`. The cloud factor is then the measured incoming solar radiation divided by the modeled radiation. The cloud factor can be computed on an hourly timescale if the measurement locations are of high quality. For stations that are less reliable, we recommend calculating a daily cloud factor which divides the daily integrated measured radiation by the daily integrated modeled radiation. This helps to reduce the problems that may be encountered from instrument shading, instrument calibration, or a time shift in the data. The calculated cloud factor at each station can then be distrubted using any of the method available in *smrf.spatial*. Since the cloud factor is not explicitly controlled by elevation like other variables, the values may be distributed without detrending to elevation. The modeled clear sky radiation (both beam and diffuse) are adjusted for clouds using *smrf.envphys.radiation.cf_cloud*.

The third step adjusts the cloud corrected solar radiation for vegetation affects, following the methods developed by Link and Marks (1999) [8]. The direct beam radiation is corrected by:

$$R_b = S_b * exp(-\mu h / cos\theta)$$

where S_b is the above canopy direct radiation, μ is the extinction coefficient (m^{-1}), h is the canopy height (m), θ is the solar zenith angle, and R_b is the canopy adjusted direct radiation. Adjusting the diffuse radiation is performed by:

$$R_d = \tau * R_d$$

where R_d is the diffuse adjusted radiation, τ is the optical transmissivity of the canopy, and R_d is the above canopy diffuse radiation. Values for μ and τ can be found in Link and Marks (1999) [8], measured at study sites in Saskatchewan and Manitoba.

The final step for calculating the net solar radiation requires the surface albedo from *smrf.distribute.albedo*. The net radiation is the sum of the of beam and diffuse canopy adjusted radiation multipled by one minus the albedo.

Parameters

- **config** – full configuration dictionary contain at least the sections albedo, and solar
- **stoporad_in** – file path to the stoporad_in file created from *smrf.data.loadTopo.Topo*
- **tempDir** – location of temp/working directory (default=None, which is the 'WORKDIR' environment variable)

albedoConfig
configuration from [albedo] section

config
configuration from [albedo] section

clear_ir_beam
numpy array modeled clear sky infrared beam radiation

clear_ir_diffuse
numpy array modeled clear sky infrared diffuse radiation

clear_vis_beam
numpy array modeled clear sky visible beam radiation

clear_vis_diffuse
numpy array modeled clear sky visible diffuse radiation

cloud_factor
numpy array distributed cloud factor

cloud_ir_beam
numpy array cloud adjusted infrared beam radiation

cloud_ir_diffuse
numpy array cloud adjusted infrared diffuse radiation

cloud_vis_beam
numpy array cloud adjusted visible beam radiation

cloud_vis_diffuse
numpy array cloud adjusted visible diffuse radiation

ir_file
temporary file from `stoporad` for infrared clear sky radiation

metadata
metadata for the station data

net_solar
numpy array for the calculated net solar radiation

stations
stations to be used in alphabetical order

stoporad_in
file path to the stoporad_in file created from *smrf.data.loadTopo.Topo*

tempDir
temporary directory for `stoporad`, will default to the `WORKDIR` environment variable

veg_height
numpy array of vegetation heights from *smrf.data.loadTopo.Topo*

veg_ir_beam
numpy array vegetation adjusted infrared beam radiation

veg_ir_diffuse
numpy array vegetation adjusted infrared diffuse radiation

veg_k
numpy array of vegetation extinction coefficient from `smrf.data.loadTopo.Topo`

veg_tau
numpy array of vegetation optical transmissivity from `smrf.data.loadTopo.Topo`

veg_vis_beam
numpy array vegetation adjusted visible beam radiation

veg_vis_diffuse
numpy array vegetation adjusted visible diffuse radiation

vis_file
temporary file from `stoporad` for visible clear sky radiation

calc_ir (*min_storm_day*, *wy_day*, *tz_min_west*, *wyear*, *cosz*, *azimuth*)
Run `stoporad` for the infrared bands

Parameters

- **min_storm_day** – decimal day of last storm for the entire basin, from `smrf.distribute.precip.ppt.last_storm_day_basin`
- **wy_day** – day of water year, from `radiation_dates`
- **tz_min_west** – time zone in minutes west from UTC, from `radiation_dates`
- **wyear** – water year, from `radiation_dates`
- **cosz** – cosine of the zenith angle for the basin, from `smrf.envphys.radiation.sunang`
- **azimuth** – azimuth to the sun for the basin, from `smrf.envphys.radiation.sunang`

calc_net (*albedo_vis*, *albedo_ir*)
Calculate the net radiation using the vegetation adjusted radiation. Sets `net_solar`.

Parameters

- **albedo_vis** – numpy array for visible albedo, from `smrf.distribute.albedo.albedo.albedo_vis`
- **albedo_ir** – numpy array for infrared albedo, from `smrf.distribute.albedo.albedo.albedo_ir`

calc_vis (*min_storm_day*, *wy_day*, *tz_min_west*, *wyear*, *cosz*, *azimuth*)
Run `stoporad` for the visible bands

Parameters

- **min_storm_day** – decimal day of last storm for the entire basin, from `smrf.distribute.precip.ppt.last_storm_day_basin`
- **wy_day** – day of water year, from `radiation_dates`
- **tz_min_west** – time zone in minutes west from UTC, from `radiation_dates`
- **wyear** – water year, from `radiation_dates`

- **cosz** – cosine of the zenith angle for the basin, from `smrf.envphys.radiation.sunang`
- **azimuth** – azimuth to the sun for the basin, from `smrf.envphys.radiation.sunang`

cloud_correct()

Correct the modeled clear sky radiation for cloud cover using *smrf.envphys.radiation.cf_cloud*. Sets *cloud_vis_beam* and *cloud_vis_diffuse*.

distribute(*t*, *cloud_factor*, *illum_ang*, *cosz*, *azimuth*, *min_storm_day*, *albedo_vis*, *albedo_ir*)

Distribute air temperature given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

If the sun is up, i.e. `cosz > 0`, then the following steps are performed:

1. Model clear sky radiation
2. Cloud correct with `smrf.distribute.solar.solar.cloud_correct`
3. **vegetation correct with** `smrf.distribute.solar.solar.veg_correct`
4. **Calculate net radiation with** `smrf.distribute.solar.solar.calc_net`

If sun is down, then all calculated values will be set to `None`, signaling the output functions to put zeros in their place.

Parameters

- **cloud_factor** – Numpy array of the domain for cloud factor
- **cosz** – cosine of the zenith angle for the basin, from `smrf.envphys.radiation.sunang`
- **azimuth** – azimuth to the sun for the basin, from `smrf.envphys.radiation.sunang`
- **min_storm_day** – decimal day of last storm for the entire basin, from `smrf.distribute.precip.ppt.last_storm_day_basin`
- **albedo_vis** – numpy array for visible albedo, from *smrf.distribute.albedo.albedo.albedo_vis*
- **albedo_ir** – numpy array for infrared albedo, from *smrf.distribute.albedo.albedo.albedo_ir*

distribute_thread(*queue*, *data*)

Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step following the methods outlined in *smrf.distribute.solar.solar.distribute*. The data queues puts the distributed data into:

- *net_solar*

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

distribute_thread_clear(*queue*, *data*, *calc_type*)

Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and model clear sky radiation with `stoporad`. The data queues puts the distributed data into:

- *clear_vis_beam*
- *clear_vis_diffuse*
- *clear_ir_beam*
- *clear_ir_diffuse*

initialize (*topo*, *data*)
Initialize the distribution, soley calls `smrf.distribute.image_data.image_data._initialize`. Sets the following attributes:

- *veg_height*
- *veg_tau*
- *veg_k*

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **data** – data Pandas dataframe containing the station data, from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'clear_ir_beam': {'long_name': 'Clear sky infrared beam solar ra

post_process_variables = {}

radiation_dates (*date_time*)
Calculate some times based on the date for `stoporad`

Parameters **date_time** – date time object

Returns

tuple containing:

- **wy_day** - day of water year from October 1
- **wyear** - water year
- **tz_min_west** - minutes west of UTC for timezone

Return type (tuple)

variable = 'solar'

veg_correct (*illum_ang*)
Correct the cloud adjusted radiation for vegetation using *smrf.envphys.radiation.veg_beam* and *smrf.envphys.radiation.veg_diffuse*. Sets *veg_vis_beam*, *veg_vis_diffuse*, *veg_ir_beam*, and *veg_ir_diffuse*.

Parameters **illum_ang** – numpy array of the illumination angle over the DEM, from `smrf.envphys.radiation.sunang`

smrf.distribute.thermal module

class `smrf.distribute.thermal.`**th**(*thermalConfig*)

Bases: *smrf.distribute.image_data.image_data*

The *th* class allows for variable specific distributions that go beyond the base class.

Thermal radiation, or long-wave radiation, is calculated based on the clear sky radiation emitted by the atmosphere. Multiple methods for calculating thermal radition exist and SMRF has 4 options for estimating clear sky thermal radiation. Selecting one of the options below will change the equations used. The methods were chosen based on the study by Flerchinger et al (2009) [9] who performed a model comparison using 21 AmeriFlux sites from North America and China.

Marks1979 The methods follow those developed by Marks and Dozier (1979) [10] that calculates the effective clear sky atmospheric emissivity using the distributed air temperature, distributed dew point temperature, and the elevation. The clear sky radiation is further adjusted for topographic affects based on the percent of the sky visible at any given point.

Dilley1998

$$L_{clear} = 59.38 + 113.7 * \left(\frac{T_a}{273.16}\right)^6 + 96.96\sqrt{w/25}$$

References: Dilley and O'Brian (1998) [11]

Prata1996

$$\epsilon_{clear} = 1 - (1 + w) * exp(-1.2 + 3w)^{1/2}$$

References: Prata (1996) [12]

Angstrom1918

$$\epsilon_{clear} = 0.83 - 0.18 * 10^{-0.067e_a}$$

References: Angstrom (1918) [13] as cityed by Niemela et al (2001) [14]

The topographic correct clear sky thermal radiation is further adjusted for cloud affects. Cloud correction is based on fraction of cloud cover, a cloud factor close to 1 meaning no clouds are present, there is little radiation added. When clouds are present, or a cloud factor close to 0, then additional long wave radiation is added to account for the cloud cover. Selecting one of the options below will change the equations used. The methods were chosen based on the study by Flerchinger et al (2009) [9], where $c = 1 - cloud\_factor$.

Garen2005 Cloud correction is based on the relationship in Garen and Marks (2005) [15] between the cloud factor and measured long wave radiation using measurement stations in the Boise River Basin.

$$L_{cloud} = L_{clear} * (1.485 - 0.488 * cloud\_factor)$$

Unsworth1975

$$\begin{aligned}
L_d &= L_{clear} + \tau_8 c f_8 \sigma T_c^4 \\
\tau_8 &= 1 - \epsilon_{8z}(1.4 - 0.4\epsilon_{8z}) \\
\epsilon_{8z} &= 0.24 + 2.98 \times 10^{-6} e_o^2 exp(3000/T_o) \\
f_8 &= -0.6732 + 0.6240 \times 10^{-2} T_c - 0.9140 \times 10^{-5} T_c^2
\end{aligned}$$

References: Unsworth and Monteith (1975) [16]

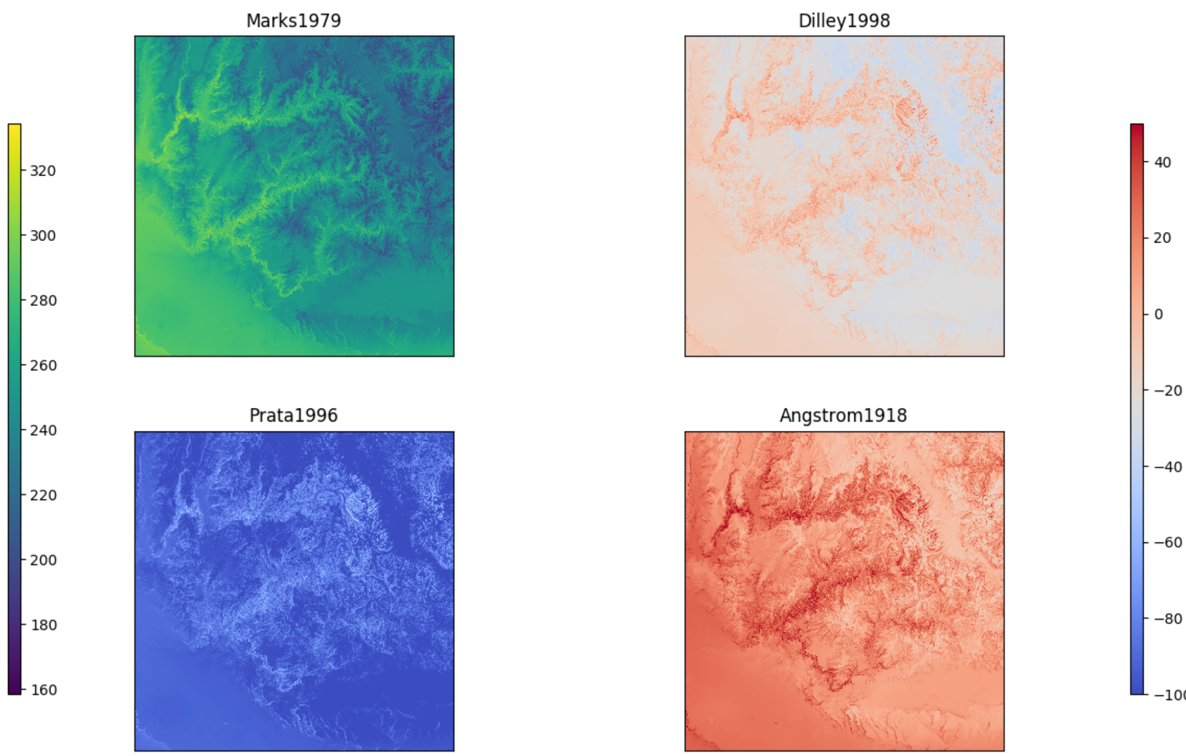


Fig. 1.5: The 4 different methods for estimating clear sky thermal radiation for a single time step. As compared to the Mark1979 method, the other methods provide a wide range in the estimated value of thermal radiation.

Kimball1982

$$L_d = L_{clear} + \tau_8 c \sigma T_c^4$$

where the original Kimball et al. (1982) [17] was for multiple cloud layers, which was simplified to one layer. T_c is the cloud temperature and is assumed to be 11 K cooler than T_a.

References: Kimball et al. (1982) [17]

Crawford1999

$$\epsilon_a = (1 - cloud\_factor) + cloud\_factor * \epsilon_{clear}$$

References: Crawford and Duchon (1999) [18] where $cloud\_factor$ is the ratio of measured solar radiation to the clear sky irradiance.

The results from Flerchinger et al (2009) [9] showed that the Kimball1982 cloud correction with Dilley1998 clear sky algorthim had the lowest RMSD. The Crawford1999 worked best when combined with Angstrom1918, Dilley1998, or Prata1996.

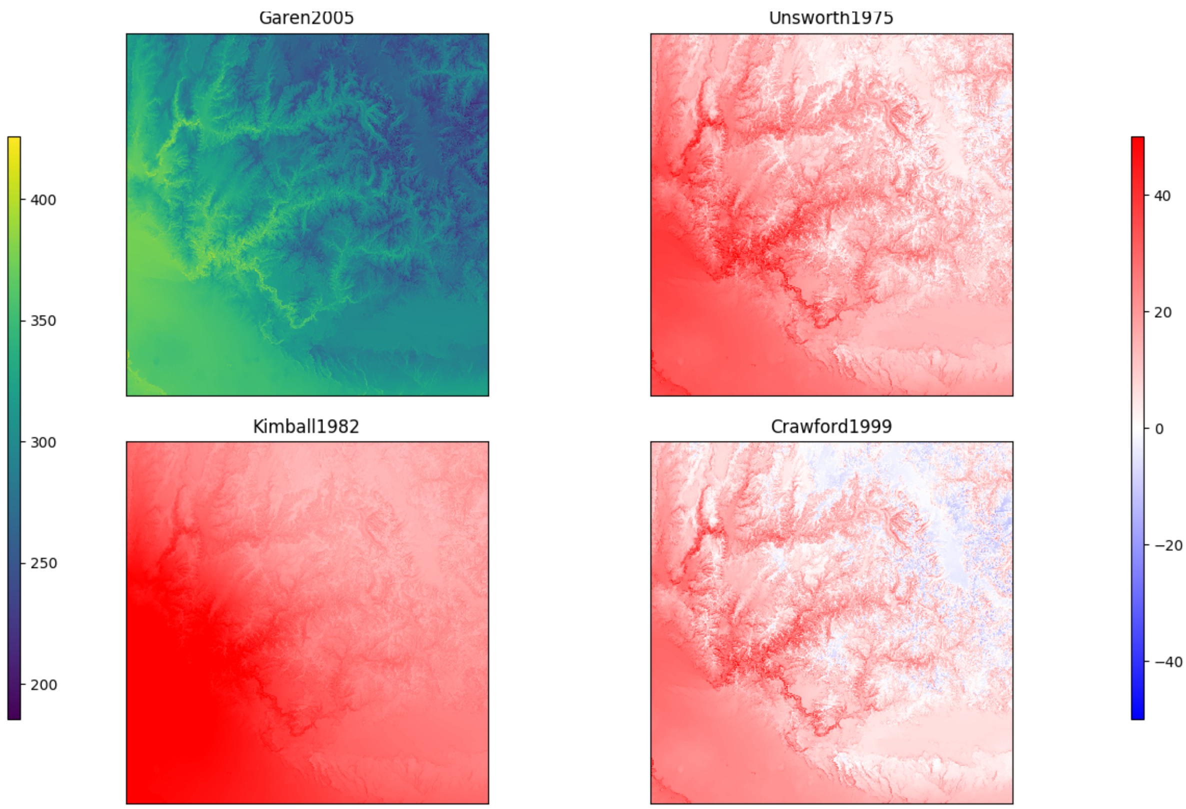


Fig. 1.6: The 4 different methods for correcting clear sky thermal radiation for cloud affects at a single time step. As compared to the Garen2005 method, the other methods are typically higher where clouds are present (i.e. the lower left) where the cloud factor is around 0.4.

The thermal radiation is further adjusted for canopy cover after the work of Link and Marks (1999) [8]. The correction is based on the vegetation's transmissivity, with the canopy temperature assumed to be the air temperature for vegetation greater than 2 meters. The thermal radiation is adjusted by

$$L_{canopy} = \tau_d * L_{cloud} + (1 - \tau_d) \epsilon \sigma T_a^4$$

where τ_d is the optical transmissivity, L_{cloud} is the cloud corrected thermal radiation, ϵ is the emissivity of the canopy (0.96), σ is the Stephan-Boltzmann constant, and T_a is the distributed air temperature.

Parameters **thermalConfig** – The [thermal] section of the configuration file

config
configuration from [thermal] section

thermal
numpy array of the precipitation

min
minimum value of thermal is -600 W/m^2

max
maximum value of thermal is 600 W/m^2

stations
stations to be used in alphabetical order

dem
numpy array for the DEM, from *smrf.data.loadTopo.Topo.dem*

veg_type
numpy array for the veg type, from *smrf.data.loadTopo.Topo.veg_type*

veg_height
numpy array for the veg height, from *smrf.data.loadTopo.Topo.veg_height*

veg_k
numpy array for the veg K, from *smrf.data.loadTopo.Topo.veg_k*

veg_tau
numpy array for the veg transmissivity, from *smrf.data.loadTopo.Topo.veg_tau*

sky_view
numpy array for the sky view factor, from *smrf.data.loadTopo.Topo.sky_view*

distribute(*date_time*, *air_temp*, *vapor_pressure=None*, *dew_point=None*, *cloud_factor=None*)
Distribute for a single time step.

The following steps are taken when distributing thermal:

1. **Calculate the clear sky thermal radiation from** *smrf.envphys.core.envphys_c.ctopotherm*
2. Correct the clear sky thermal for the distributed cloud factor
3. Correct for canopy affects

Parameters

- **date_time** – datetime object for the current step
- **air_temp** – distributed air temperature for the time step
- **vapor_pressure** – distributed vapor pressure for the time step
- **dew_point** – distributed dew point for the time step
- **cloud_factor** – distributed cloud factor for the time step measured/modeled

distribute_thermal (*data*, *air_temp*)

Distribute given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`. Used when thermal is given (i.e. gridded datasets from WRF). Follows these steps:

1. Distribute the thermal radiation from point values
2. Correct for vegetation

Parameters

- **data** – thermal values
- **air_temp** – distributed air temperature values

distribute_thermal_thread (*queue*, *data*)

Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call *smrf.distribute.thermal.th.distribute_thermal* then puts the distributed data into the queue for *thermal*. Used when thermal is given (i.e. gridded datasets from WRF).

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

distribute_thread (*queue*, *date*)

Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call *smrf.distribute.thermal.th.distribute* then puts the distributed data into the queue for *thermal*.

Parameters

- **queue** – queue dictionary for all variables
- **data** – pandas dataframe for all data, indexed by date time

initialize (*topo*, *data*)

Initialize the distribution, calls `smrf.distribute.image_data.image_data._initialize` for gridded distirbution. Sets the following from *smrf.data.loadTopo.Topo*

- *veg_height*
- *veg_tau*
- *veg_k*
- *sky_view*
- *dem*

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **data** – data Pandas dataframe containing the station data, from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'thermal': {'long_name': 'Thermal (longwave) radiation', 'standa

post_process_variables = {}

variable = 'thermal'

smrf.distribute.vapor_pressure module

class `smrf.distribute.vapor_pressure.`**vp**(*vpConfig*, *precip_temp_method*)

Bases: `smrf.distribute.image_data.image_data`

The `vp` class allows for variable specific distributions that go beyond the base class

Vapor pressure is provided as an argument and is calculated from coincident air temperature and relative humidity measurements using utilities such as IPW's `rh2vp`. The vapor pressure is distributed instead of the relative humidity as it is an absolute measurement of the vapor within the atmosphere and will follow elevational trends (typically negative). Were as relative humidity is a relative measurement which varies in complex ways over the topography. From the distributed vapor pressure, the dew point is calculated for use by other distribution methods. The dew point temperature is further corrected to ensure that it does not exceed the distributed air temperature.

Parameters **vpConfig** – The [vapor_pressure] section of the configuration file

config

configuration from [vapor_pressure] section

vapor_pressure

numpy matrix of the vapor pressure

dew_point

numpy matrix of the dew point, calculated from vapor_pressure and corrected for dew_point greater than air_temp

min

minimum value of vapor pressure is 10 Pa

max

maximum value of vapor pressure is 7500 Pa

stations

stations to be used in alphabetical order

distribute(*data*, *ta*)

Distribute air temperature given a Panda's dataframe for a single time step. Calls `smrf.distribute.image_data.image_data._distribute`.

The following steps are performed when distributing vapor pressure:

1. Distribute the point vapor pressure measurements
2. **Calculate dew point temperature using** `smrf.envphys.core.envphys_c.cdewpt`
3. Adjust dew point values to not exceed the air temperature

Parameters

- **data** – Pandas dataframe for a single time step from precip
- **ta** – air temperature numpy array that will be used for calculating dew point temperature

distribute_thread(*queue*, *data*)

Distribute the data using threading and queue. All data is provided and `distribute_thread` will go through each time step and call `smrf.distribute.vapor_pressure.vp.distribute` then puts the distributed data into the queue for:

- `vapor_pressure`

- *dew_point*

  Parameters

  - **queue** – queue dictionary for all variables
  - **data** – pandas dataframe for all data, indexed by date time

initialize (*topo*, *data*)
Initialize the distribution, calls `smrf.distribute.image_data.image_data._initialize`. Preallocates the following class attributes to zeros:

Parameters

- **topo** – *smrf.data.loadTopo.Topo* instance contain topographic data and infomation
- **data** – data Pandas dataframe containing the station data, from *smrf.data.loadData* or *smrf.data.loadGrid*

output_variables = {'dew_point': {'long_name': 'Dew point temperature', 'standard_nar

post_process_variables = {}

variable = 'vapor_pressure'

Module contents

1.3.3 smrf.envphys package

Subpackages

smrf.envphys.core package

Submodules

smrf.envphys.core.envphys_c module

C implementation of some radiation functions

`smrf.envphys.core.envphys_c.`**cdewpt** (*ndarray vp*, *ndarray dwpt*, *float tolerance=0*, *int nthreads=1*)

Parameters **vp** –

Out: dwpt changed in place

20160505 Scott Havens

`smrf.envphys.core.envphys_c.`**ctopotherm** (*ndarray ta*, *ndarray tw*, *ndarray z*, *ndarray skvfac*, *ndarray thermal*, *int nthreads=1*)
Call the function krige_grid in krige.c which will iterate over the grid within the C code

Parameters **tw, z, skvfac** (*ta,*) –

Out: thermal changed in place

20160325 Scott Havens

`smrf.envphys.core.envphys_c.`**`cwbt`**`(`*ndarray ta*, *ndarray td*, *ndarray z*, *ndarray tw*, *float tolerance=0*, *int nthreads=1*`)`

Call the function iwbt in iwbt.c which will iterate over the grid within the C code

Parameters **`td, z, tw_o`**`(ta,)` –

Out: tw changed in place (wet bulb temperature)

20180611 Micah Sandusky

smrf.envphys.core.envphys_c module

C implementation of some radiation functions

`smrf.envphys.core.envphys_c.`**`cdewpt`**`(`*ndarray vp*, *ndarray dwpt*, *float tolerance=0*, *int nthreads=1*`)`

Parameters **`vp`** –

Out: dwpt changed in place

20160505 Scott Havens

`smrf.envphys.core.envphys_c.`**`ctopotherm`**`(`*ndarray ta*, *ndarray tw*, *ndarray z*, *ndarray skvfac*, *ndarray thermal*, *int nthreads=1*`)`

Call the function krige_grid in krige.c which will iterate over the grid within the C code

Parameters **`tw, z, skvfac`**`(ta,)` –

Out: thermal changed in place

20160325 Scott Havens

`smrf.envphys.core.envphys_c.`**`cwbt`**`(`*ndarray ta*, *ndarray td*, *ndarray z*, *ndarray tw*, *float tolerance=0*, *int nthreads=1*`)`

Call the function iwbt in iwbt.c which will iterate over the grid within the C code

Parameters **`td, z, tw_o`**`(ta,)` –

Out: tw changed in place (wet bulb temperature)

20180611 Micah Sandusky

Module contents

Submodules

smrf.envphys.phys module

Created April 15, 2015

Collection of functions to calculate various physical parameters

@author: Scott Havens

`smrf.envphys.phys.`**`idewpt`**`(`*vp*`)`

Calculate the dew point given the vapor pressure

Parameters – array of vapor pressure values in [Pa] (vp) –

Returns

dewpt - array same size as vp of the calculated dew point temperature [C] (see Dingman 2002).

smrf.envphys.phys.**rh2vp**(*ta*, *rh*)

Calculate the vapor pressure given the air temperature and relative humidity

Parameters

- **ta** – array of air temperature in [C]
- **rh** – array of relative humidity from 0-100 [%]

Returns vapor pressure

smrf.envphys.phys.**satvp**(*dpt*)

Calculate the saturation vapor pressure at the dew point temperature.

Parameters **dwpt** – array of dew point temperature in [C]

Returns vapor_pressure

smrf.envphys.precip module

Created on Apr 15, 2015

@author: scott

smrf.envphys.precip.**adjust_for_undercatch**(*p_vec*, *wind*, *temp*, *sta_type*, *metadata*)

Adjusts the vector precip station data for undercatchment. Relationships should be added to `catchment_ratio()`.

Parameters

- **– The station vector data in pandas series** (*p_vec*) –
- **– The vector wind data** (*wind*) –
- **– The vector air_temp data** (*temp*) –
- **– A dictionary of station names and the type of correction to apply** (*sta_type*) –
- **– station metadata TODO merge in the station_dict info to metadata** (*station_metadata*) –

Returns adj_precip - Adjust precip accoding to the corrections applied.

smrf.envphys.precip.**catchment_ratios**(*ws*, *gauge_type*, *snowing*)

Point models for catchment ratios of the

smrf.envphys.precip.**dist_precip_wind**(*precip*, *precip_temp*, *az*, *dir_round_cell*, *wind_speed*, *cell_maxus*, *tbreak*, *tbreak_direction*, *veg_type*, *veg_fact*, *cfg*, *mask=None*)

Redistribute the precip based on wind speed and direciton to account for drifting.

Parameters

- **precip** – distributed precip
- **precip_temp** – precip_temp array

- **az** – wind direction
- **dir_round_cell** – from wind module
- **wind_speed** – wind speed array
- **cell_maxus** – max upwind slope from maxus file
- **tbreak** – relative local slope from tbreak file
- **tbreak_direction** – direction array from tbreak file
- **veg_type** – user input veg types to correct
- **veg_factor** – interception correction for veg types. ppt mult is multiplied by 1/veg_factor

Returns numpy array of precip redistributed for wind

Return type precip_drift

smrf.envphys.precip.**mkprecip**(*precipitation*, *temperature*)

Follows the IPW command mkprecip

The precipitation phase, or the amount of precipitation falling as rain or snow, can significantly alter the energy and mass balance of the snowpack, either leading to snow accumulation or inducing melt [2] [3]. The precipitation phase and initial snow density are based on the precipitation temperature (the distributed dew point temperature) and are estimated after Susong et al (1999) [4]. The table below shows the relationship to precipitation temperature:

Min Temp [deg C]	Max Temp [deg C]	Percent snow [%]	Snow density [kg/m^3]
-Inf	-5	100	75
-5	-3	100	100
-3	-1.5	100	150
-1.5	-0.5	100	175
-0.5	0	75	200
0	0.5	25	250
0.5	Inf	0	0

Parameters

- **– array of precipitation values [mm]** (*precipitation*) –
- **– array of temperature values, use dew point temperature** (*temperature*) – if available [degree C]

Output: returns the percent snow and estimated snow density

smrf.envphys.precip.**storms**(*precipitation*, *perc_snow*, *mass=1*, *time=4*, *stormDays=None*, *stormPrecip=None*, *ps_thresh=0.5*)

Calculate the decimal days since the last storm given a precip time series, percent snow, mass threshold, and time threshold

- Will look for pixels where perc_snow > 50% as storm locations
- **A new storm will start if the mass at the pixel has exceeded the mass** limit, this ensures that the enough has accumulated

Parameters

- **– precipitation values** (*precipitation*) –
- **– precent of precipitation that was snow** (*perc_snow*) –
- **– threshold for the mass to start a new storm** (*mass*) –
- **– threshold for the time to start a new storm** (*time*) –
- **– if specified, this is the output from a previous run** (*stormDays*) – of storms
- **– keeps track of the total storm precip** (*stormPrecip*) –

Returns

(stormDays, stormPrecip) - the timesteps since the last storm and the total precipitation for the storm

Created April 17, 2015 @author: Scott Havens

`smrf.envphys.precip.`**`storms_time`**(*precipitation*, *perc_snow*, *time_step=0.041666666666666664*, *mass=1*, *time=4*, *stormDays=None*, *stormPrecip=None*, *ps_thresh=0.5*)

Calculate the decimal days since the last storm given a precip time series, percent snow, mass threshold, and time threshold

- Will look for pixels where perc_snow > 50% as storm locations
- **A new storm will start if the mass at the pixel has exceeded the mass** limit, this ensures that the enough has accumulated

Parameters

- **– precipitation values** (*precipitation*) –
- **– precent of precipitation that was snow** (*perc_snow*) –
- **time_step** – step in days of the model run
- **– threshold for the mass to start a new storm** (*mass*) –
- **– threshold for the time to start a new storm** (*time*) –
- **– if specified, this is the output from a previous run** (*stormDays*) – of storms else it will be set to the date_time value
- **– keeps track of the total storm precip** (*stormPrecip*) –

Returns

(stormDays, stormPrecip) - the timesteps since the last storm and the total precipitation for the storm

Created Janurary 5, 2016 @author: Scott Havens

smrf.envphys.radiation module

smrf.envphys.radiation.**albedo**(*telapsed*, *cosz*, *gsize*, *maxgsz*, *dirt=2*)

Calculate the abedo, adapted from IPW function albedo

Parameters

- **- time since last snow storm** (*telapsed*) –
- **- cosine local solar illumination angle matrix** (*cosz*) –
- **- gsize is effective grain radius of snow after last storm** (*gsize*) –
- **- maxgsz is maximum grain radius expected from grain growth** (*maxgsz*) –
- **- dirt is effective contamination for adjustment to visible albedo** (*dirt*) –

Returns

Returns a tuple containing the visible and IR spectral albedo

- **alb_v** (*numpy.array*) - albedo for visible specturm
- **alb_ir** (*numpy.array*) - albedo for ir spectrum

Return type tuple

smrf.envphys.radiation.**beta_0**(*cosz*, *g*)

we find the integral-sum

sum (n=0 to inf) g^n * (2*n+1) * Pn(u0) * int (u'=0 to 1) Pn(u')

note that int of Pn vanishes for even values of n (Abramowitz & Stegun, eq 22.13.8-9); therefore the series becomes

sum (n=0 to inf) g^n * (2*n+1) * Pn(u0) * f(m)

where 2*m+1 = n and the f's are computed recursively

Parameters

- **cosz** – cosine illumination angle
- **g** – scattering asymmetry param

Returns beta_0

smrf.envphys.radiation.**cf_cloud**(*beam*, *diffuse*, *cf*)

Correct beam and diffuse irradiance for cloud attenuation at a single time, using input clear-sky global and diffuse radiation calculations supplied by locally modified toporad or locally modified stoporad

Parameters

- **beam** – global irradiance
- **diffuse** – diffuse irradiance
- **cf** – cloud attenuation factor - actual irradiance / clear-sky irradiance

Returns cloud corrected gobal irradiance c_drad: cloud corrected diffuse irradiance

Return type c_grad

20150610 Scott Havens - adapted from cloudcalc.c `smrf.envphys.radiation.`**`decay_alb_hardy`**(*litter*, *veg_type*, *storm_day*, *alb_v*, *alb_ir*)

Find a decrease in albedo due to litter acccumulation using method from [21] with storm_day as input.

$$lc = 1.0 - (1.0 - lr)^{day}$$

Where lc is the fractional litter coverage and lr is the daily litter rate of the forest. The new albedo is a weighted average of the calculated albedo for the clean snow and the albedo of the litter.

Note: uses input of l_rate (litter rate) from config which is based on veg type. This is decimal percent litter coverage per day

Parameters

- **`litter`** – A dictionary of values for default,albedo,41,42,43 veg types
- **`veg_type`** – An image of the basin's NLCD veg type
- **`storm_day`** – numpy array of decimal day since last storm
- **`alb_v`** – numpy array of albedo for visibile spectrum
- **`alb_ir`** – numpy array of albedo for IR spectrum
- **`alb_litter`** – albedo of pure litter

Returns

Returns a tuple containing the corrected albedo arrays based on date, veg type - **alb_v** (*numpy.array*) - albedo for visible specturm

- **alb_ir** (*numpy.array*) - albedo for ir spectrum

Return type tuple

Created July 19, 2017 Micah Sandusky

`smrf.envphys.radiation.`**`decay_alb_power`**(*veg*, *veg_type*, *start_decay*, *end_decay*, *t_curr*, *pwr*, *alb_v*, *alb_ir*)

Find a decrease in albedo due to litter acccumulation. Decay is based on max decay, decay power, and start and end dates. No litter decay occurs before start_date. Fore times between start and end of decay,

$$\alpha = \alpha - (dec_{max}^{\frac{1.0}{pwr}} \times \frac{t - start}{end - start})^{pwr}$$

Where α is albedo, dec_{max} is the maximum decay for albedo, pwr is the decay power, t, $start$, and end are the current, start, and end times for the litter decay.

Parameters

- **`start_decay`** – date to start albedo decay (datetime)
- **`end_decay`** – date at which to end albedo decay curve (datetime)
- **`t_curr`** – datetime object of current timestep
- **`pwr`** – power for power law decay
- **`alb_v`** – numpy array of albedo for visibile spectrum
- **`alb_ir`** – numpy array of albedo for IR spectrum

Returns

Returns a tuple containing the corrected albedo arrays based on date, veg type - **alb_v** (*numpy.array*) - albedo for visible specturm

- **alb_ir** (*numpy.array*) - albedo for ir spectrum

Return type tuple

Created July 18, 2017 Micah Sandusky

smrf.envphys.radiation.**deg_to_dms**(*deg*)
Decimal degree to degree, minutes, seconds

smrf.envphys.radiation.**find_horizon**(*i*, *H*, *xr*, *yr*, *Z*, *mu*)

smrf.envphys.radiation.**get_hrrr_cloud**(*df_solar*, *df_meta*, *logger*, *lat*, *lon*)
Take the combined solar from HRRR and use the two stream calculation at the specific HRRR pixels to find the cloud_factor.

Parameters

- **- solar dataframe from hrrr** (*df_solar*) –
- **- meta_data from hrrr** (*df_meta*) –
- **- smrf logger** (*logger*) –
- **- basin lat** (*lat*) –
- **- basin lon** (*lon*) –

Returns df_cf - cloud factor dataframe in same format as df_solar input

smrf.envphys.radiation.**growth**(*t*)
Calculate grain size growth From IPW albedo > growth

smrf.envphys.radiation.**hor1f**(*x*, *z*, *offset=1*)
BROKEN: Haven't quite figured this one out

Calculate the horizon pixel for all x,z This mimics the algorthim from Dozier 1981 and the hor1f.c from IPW

Works backwards from the end but looks forwards for the horizon

xrange stops one index before [stop]

Parameters

- **- horizontal distances for points** (*x*) –
- **- elevations for the points** (*z*) –

Returns h - index to the horizon point

20150601 Scott Havens

smrf.envphys.radiation.**hor1f_simple**(*z*)
Calculate the horizon pixel for all x,z This mimics the simple algorthim from Dozier 1981 to help understand how it's working

Works backwards from the end but looks forwards for the horizon 90% faster than rad.horizon

Parameters

- **- horizontal distances for points** (*x*) –
- **- elevations for the points** (*z*) –

Returns h - index to the horizon point

20150601 Scott Havens

smrf.envphys.radiation.**hord**(*z*)
Calculate the horizon pixel for all x,z This mimics the simple algorthim from Dozier 1981 to help understand how it's working

Works backwards from the end but looks forwards for the horizon 90% faster than rad.horizon

Args:: x - horizontal distances for points z - elevations for the points

Returns h - index to the horizon point

20150601 Scott Havens

`smrf.envphys.radiation.`**`ihorizon`**(*x*, *y*, *Z*, *azm*, *mu=0*, *offset=2*, *ncores=0*)

Calculate the horizon values for an entire DEM image for the desired azimuth

Assumes that the step size is constant

Parameters

- **`- vector of x-coordinates`** (*X*) –
- **`- vector of y-coordinates`** (*Y*) –
- **`- matrix of elevation data`** (*Z*) –
- **`- azimuth to calculate the horizon at`** (*azm*) –
- **`- 0 -> calculate cos`** (*mu*) –
  - >0 -> calculate a mask whether or not the point can see the sun

Returns

H - if mask=0 cosine of the local horizonal angles

- if mask=1 index along line to the point

20150602 Scott Havens

`smrf.envphys.radiation.`**`model_solar`**(*dt*, *lat*, *lon*, *tau=0.2*, *tzone=0*)

Model solar radiation at a point Combines sun angle, solar and two stream

Parameters

- **`- datetime object`** (*dt*) –
- **`- latitude`** (*lat*) –
- **`- longitude`** (*lon*) –
- **`- optical depth`** (*tau*) –
- **`- time zone`** (*tzone*) –

Returns corrected solar radiation

`smrf.envphys.radiation.`**`mwgamma`**(*cosz*, *omega*, *g*)

gamma's for phase function for input using the MEADOR WEAVER method

Two-stream approximations to radiative transfer in planetary atmospheres: a unified description of existing methods and a new improvement, Meador & Weaver, 1980

Parameters

- **`cosz`** – cosine illumination angle
- **`omega`** – single-scattering albedo
- **`g`** – scattering asymmetry param

Returns gamma values

`smrf.envphys.radiation.`**`shade`**(*slope*, *aspect*, *azimuth*, *cosz=None*, *zenith=None*)

Calculate the cosize of the local illumination angle over a DEM

Solves the following equation cos(ts) = cos(t0) * cos(S) + sin(t0) * sin(S) * cos(phi0 - A)

where t0 is the illumination angle on a horizontal surface phi0 is the azimuth of illumination S is slope in radians A is aspect in radians

Slope and aspect are expected to come from the IPW gradient command. Slope is stored as sin(S) with range from 0 to 1. Aspect is stored as radians from south (aspect 0 is toward the south) with range from -pi to pi, with negative values to the west and positive values to the east

Parameters

- **`slope`** – numpy array of sine of slope angles sin(S)
- **`aspect`** – numpy array of aspect in radians from south
- **`azimuth`** – azimuth in degrees to the sun -180..180 (comes from sunang)
- **`cosz`** – cosize of the zeinith angle 0..1 (comes from sunang)
- **`zenith`** – the solar zenith angle 0..90 degrees

At least on of the cosz or zenith must be specified. If both are specified the zenith is ignored

Returns numpy matrix of the cosize of the local illumination angle cos(ts)

Return type mu

The python shade() function is an interpretation of the IPW shade() function and follows as close as possible. All equations are based on Dozier & Frew, 1990. 'Rapid calculation of Terrain Parameters For Radiation Modeling From Digitial Elevation Data,' IEEE TGARS

20150106 Scott Havens

`smrf.envphys.radiation.`**`shade_thread`**(*queue*, *date*, *slope*, *aspect*, *zenith=None*)

See shade for input argument descriptions

Parameters

- **`queue`** – queue with illum_ang, cosz, azimuth
- **`date_time`** – loop through dates to accesss queue

20160325 Scott Havens

`smrf.envphys.radiation.`**`solar`**(*d*, *w=[0.28, 2.8]*)

Solar calculates exoatmospheric direct solar irradiance. If two arguments to -w are given, the integral of solar irradiance over the range will be calculated. If one argument is given, the spectral irradiance will be calculated.

If no wavelengths are specified on the command line, single wavelengths in um will be read from the standard input and the spectral irradiance calculated for each.

Parameters

- **`- [um um2] If two arguments are given, the integral of solar`** (*w*) – irradiance in the range um to um2 will be calculated. If one argument is given, the spectral irradiance will be calculated.
- **`- date object, This is used to calculate the solar radius vector`** (*d*) – which divides the result

Returns s - direct solar irradiance

`smrf.envphys.radiation.`**`solar_data`**()

Solar data from Thekaekara, NASA TR-R-351, 1979 `smrf.envphys.radiation.`**`solar_ipw`**(*d*, *w=[0.28, 2.8]*)

Wrapper for the IPW solar function

Solar calculates exoatmospheric direct solar irradiance. If two arguments to -w are given, the integral of solar irradiance over the range will be calculated. If one argument is given, the spectral irradiance will be calculated.

If no wavelengths are specified on the command line, single wavelengths in um will be read from the standard input and the spectral irradiance calculated for each.

Parameters

- **- [um um2] If two arguments are given, the integral of solar** (*w*) – irradiance in the range um to um2 will be calculated. If one argument is given, the spectral irradiance will be calculated.
- **- date object, This is used to calculate the solar radius vector** (*d*) – which divides the result

Returns s - direct solar irradiance

20151002 Scott Havens

`smrf.envphys.radiation.`**`solint`**(*a*, *b*)

integral of solar constant from wavelengths a to b in micometers

This uses scipy functions which will produce different results from the IPW equvialents of 'akcoef' and 'splint'

`smrf.envphys.radiation.`**`sunang_ipw`**(*date*, *lat*, *lon*, *zone=0*, *slope=0*, *aspect=0*)

Wrapper for the IPW sunang function

Parameters

- **- date to calculate sun angle for** (*date*) –
- **- latitude in decimal degrees** (*lat*) –
- **- longitude in decimal degrees** (*lon*) –
- **- The time values are in the time zone which is min minutes** (*zone*) – west of Greenwich (default: 0). For example, if input times are in Pacific Standard Time, then min would be 480.
- **slope** (*default=0*) –
- **aspect** (*default=0*) –

Returns cosz - cosine of the zeinith angle azimuth - solar azimuth

Created April 17, 2015 Scott Havens

`smrf.envphys.radiation.`**`twostream`**(*cosz*, *S0*, *tau=0.2*, *omega=0.85*, *g=0.3*, *R0=0.5*)

Provides twostream solution for single-layer atmosphere over horizontal surface, using solution method in: Two-stream approximations to radiative transfer in planetary atmospheres: a unified description of existing methods and a new improvement, Meador & Weaver, 1980, or will use the delta-Eddington method, if the -d flag is set (see: Wiscombe & Joseph 1977).

Parameters

- **cosz** – The cosine of the incidence angle is cos (from program sunang). An error if cosz is <= 0.0; set all outputs to 0.0 and go on. Program will fail if incidence angle is <= 0.0, unless -0 has been set.
- **S0** – The direct beam irradiance is S0 This is usually the solar constant for the specified wavelength band, on the specified date, at the top of the atmosphere, from radiation.solar.
- **tau** – The optical depth is tau. 0 implies an infinite optical depth.

- **omega** – The single-scattering albedo
- **g** – The asymmetry factor is g.
- **R0** – The reflectance of the substrate is R0. If R0 is negative, it will be set to zero.

Returns R[0] - reflectance R[1] - transmittance R[2] - direct transmittance R[3] - upwelling irradiance R[4] - total irradiance at bottom R[5] - direct irradiance normal to beam

smrf.envphys.radiation.**twostream_ipw**(*mu0*, *S0*, *tau=0.2*, *omega=0.85*, *g=0.3*, *R0=0.5*, *d=False*)

Wrapper for the twostream.c IPW function

Provides twostream solution for single-layer atmosphere over horizontal surface, using solution method in: Two-stream approximations to radiative transfer in planetary atmospheres: a unified description of existing methods and a new improvement, Meador & Weaver, 1980, or will use the delta-Eddington method, if the -d flag is set (see: Wiscombe & Joseph 1977).

Parameters

- **– The cosine of the incidence angle is cos** (*mu0*) –
- **– Do not force an error if mu0 is <= 0.0; set all outputs to 0.0 and** (*0*) – go on. Program will fail if incidence angle is <= 0.0, unless -0 has been set.
- **– The optical depth is tau. 0 implies an infinite optical depth.** (*tau*) –
- **– The single-scattering albedo is omega.** (*omega*) –
- **– The asymmetry factor is g.** (*g*) –
- **– The reflectance of the substrate is R0. If R0 is negative, it** (*R0*) – will be set to zero.
- **– The direct beam irradiance is S0 This is usually the solar** (*S0*) – constant for the specified wavelength band, on the specified date, at the top of the atmosphere, from program solar. If S0 is negative, it will be set to 1/cos, or 1 if cos is not specified.
- **– The delta-Eddington method will be used.** (*d*) –

Returns R[0] - reflectance R[1] - transmittance R[2] - direct transmittance R[3] - upwelling irradiance R[4] - total irradiance at bottom R[5] - direct irradiance normal to beam

20151002 Scott Havens

smrf.envphys.radiation.**veg_beam**(*data*, *height*, *cosz*, *k*)

Apply the vegetation correction to the beam irradiance using the equation from Links and Marks 1999

S_b,f = S_b,o * exp[-k h sec(theta)] or S_b,f = S_b,o * exp[-k h / cosz]

20150610 Scott Havens

smrf.envphys.radiation.**veg_diffuse**(*data*, *tau*)

Apply the vegetation correction to the diffuse irradiance using the equation from Links and Marks 1999

S_d,f = tau * S_d,o

20150610 Scott Havens

smrf.envphys.snow module

Created on March 14, 2017 Originally written by Scott Havens in 2015 @author: Micah Johnson

Creating Custom NASDE Models

When creating a new NASDE model make sure you adhere to the following:

1. Add a new method with the other models with a unique name ideally with some reference to the origin of the model. For example see `susong1999()`.
2. Add the new model to the dictionary `available_models` at the bottom of this module so that `calc_phase_and_density()` can see it.
3. Create a custom distribution function with a unique in `distribute()` to create the structure for the new model. For an example see `distribute_for_susong1999()`.
4. Update documentation and run smrf!

`smrf.envphys.snow.`**`calc_perc_snow`**(*Tpp*, *Tmax=0.0*, *Tmin=- 10.0*)

Calculates the percent snow for the nasde_models piecewise_susong1999 and marks2017.

Parameters

- **`Tpp`** – A numpy array of temperature, use dew point temperature if available [degree C].
- **`Tmax`** – Max temperature that the percent snow is estimated. Default is 0.0 Degrees C.
- **`Tmin`** – Minimum temperature that percent snow is changed. Default is -10.0 Degrees C.

Returns A fraction of the precip at each pixel that is snow provided by Tpp.

Return type numpy.array

`smrf.envphys.snow.`**`calc_phase_and_density`**(*temperature*, *precipitation*, *nasde_model*)

Uses various new accumulated snow density models to estimate the snow density of precipitation that falls during sub-zero conditions. The models all are based on the dew point temperature and the amount of precipitation, All models used here must return a dictionary containing the keywords pcs and rho_s for percent snow and snow density respectively.

Parameters

- **`temperature`** – a single timestep of the distributed dew point temperature
- **`precipitation`** – a numpy array of the distributed precipitation
- **`nasde_model`** – string value set in the configuration file representing the method for estimating density of new snow that has just fallen.

Returns

Returns a tuple containing the snow density field and the percent snow as determined by the NASDE model.

- **snow_density** (*numpy.array*) - Snow density values in kg/m^3
- **perc_snow** (*numpy.array*) - Percent of the precip that is snow in values 0.0-1.0.

Return type tuple

`smrf.envphys.snow.`**`check_temperature`**(*Tpp*, *Tmax=0.0*, *Tmin=- 10.0*)

Sets the precipitation temperature and snow temperature.

Parameters

- **Tpp** – A numpy array of temperature, use dew point temperature if available [degrees C].
- **Tmax** – Thresholds the max temperature of the snow [degrees C].
- **Tmin** – Minimum temperature that the precipitation temperature [degrees C].

Returns

- **Tpp (*numpy.array*) - Modified precipitation temperature that** is thresholded with a minimum set by tmin.
- **tsnow (*numpy.array*) - Temperature of the surface of the snow** set by the precipitation temperature and thresholded by tmax where tsnow > tmax = tmax.

Return type tuple

`smrf.envphys.snow.`**`marks2017`**`(`*Tpp*, *pp*`)`

A new accumulated snow density model that accounts for compaction. The model builds upon `piecewise_susong1999()` by adding effects from compaction. Of four mechanisms for compaction, this model accounts for compaction by destructive metmorphosism and overburden. These two processes are accounted for by calculating a proportionalility using data from Kojima, Yosida and Mellor. The overburden is in part estimated using total storm deposition, where storms are defined in `tracking_by_station()`. Once this is determined the final snow density is applied through the entire storm only varying with hourly temperature.

Snow Density:

$$\rho_s = \rho_{ns} + (\Delta\rho_c + \Delta\rho_m)\rho_{ns}$$

Overburden Proportionality:

$$\Delta\rho_c = 0.026e^{-0.08(T_z - T_{snow})} SWE * e^{-21.0\rho_{ns}}$$

Metmorphism Proportionality:

$$\Delta\rho_m = 0.01c_{11}e^{-0.04(T_z - T_{snow})}$$
$$c_{11} = c_m in + (T_z - T_{snow})C_{factor} + 1.0$$

Constants:

$$C_{factor} = 0.0013$$
$$Tz = 0.0$$
$$ex_{max} = 1.75$$
$$exr = 0.75$$
$$ex_{min} = 1.0$$
$$c1r = 0.043$$
$$c_{min} = 0.0067$$
$$c_{fac} = 0.0013$$
$$T_{min} = -10.0$$
$$T_{max} = 0.0$$
$$T_z = 0.0$$
$$T_{r0} = 0.5$$
$$P_{cr0} = 0.25$$
$$P_{c0} = 0.75$$

Parameters

- **Tpp** – Numpy array of a single hour of temperature, use dew point if available [degrees C].
- **pp** – Numpy array representing the total amount of precip deposited during a storm in millimeters

Returns

- **rho_s** (*numpy.array*) - Density of the fresh snow in kg/m^3.
- **swe** (*numpy.array*) - Snow water equivalent.
- **pcs (*numpy.array*) - Percent of the precipitation that is** snow in values 0.0-1.0.
- **rho_ns (*numpy.array*) - Density of the uncompacted snow, which** is equivalent to the output from `piecewise_susong1999()`.
- **d_rho_c (*numpy.array*) - Prportional coefficient for** compaction from overburden.
- **d_rho_m (*numpy.array*) - Proportional coefficient for** compaction from melt.
- **rho_s** (*numpy.array*) - Final density of the snow [kg/m^3].
- **rho (*numpy.array*) - Density of the precipitation, which** continuously ranges from low density snow to pure liquid water (50-1000 kg/m^3).
- **zs** (*numpy.array*) - Snow height added from the precipitation.

Return type dictionary

`smrf.envphys.snow.`**`piecewise_susong1999`**(*Tpp*, *precip*, *Tmax=0.0*, *Tmin=- 10.0*, *check_temps=True*)

Follows `susong1999()` but is the piecewise form of table shown there. This model adds to the former by accounting for liquid water effect near 0.0 Degrees C.

The table was estimated by Danny Marks in 2017 which resulted in the piecewise equations below:

Percent Snow:

$$\%_{snow} = \begin{cases} \frac{-T}{T_{r0}} P_{cr0} + P_{c0}, & -0.5°C \leq T \leq 0.0°C \\ \frac{-T_{pp}}{T_{max}+1.0} P_{c0} + P_{c0}, & 0.0°C \leq T \leq T_{max} \end{cases}$$

Snow Density:

$$\rho_s = 50.0 + 1.7 * (T_{pp} + 15.0)^{ex}$$

$$ex = \begin{cases} ex_{min} + \frac{T_{range} + T_{snow} - T_{max}}{T_{range}} * ex_r, & ex < 1.75 \\ 1.75, & ex > 1.75 \end{cases}$$

Parameters

- **Tpp** – A numpy array of temperature, use dew point temperature if available [degree C].
- **precip** – A numpy array of precip in millimeters.
- **Tmax** – Max temperature that snow density is modeled. Default is 0.0 Degrees C.
- **Tmin** – Minimum temperature that snow density is changing. Default is -10.0 Degrees C.
- **check_temps** – A boolean value check to apply special temperature constraints, this is done using `check_temperature()`. Default is True.

Returns

- **pcs** (*numpy.array*) **- Percent of the precipitation that is snow** in values 0.0-1.0.
- **rho_s** (*numpy.array*) - Density of the fresh snow in kg/m^3.

Return type dictionary

`smrf.envphys.snow.`**`susong1999`**(*temperature*, *precipitation*)

Follows the IPW command mkprecip

The precipitation phase, or the amount of precipitation falling as rain or snow, can significantly alter the energy and mass balance of the snowpack, either leading to snow accumulation or inducing melt [2] [3]. The precipitation phase and initial snow density are based on the precipitation temperature (the distributed dew point temperature) and are estimated after Susong et al (1999) [4]. The table below shows the relationship to precipitation temperature:

Min Temp	Max Temp	Percent snow	Snow density
[deg C]	[deg C]	[%]	[kg/m^3]
-Inf	-5	100	75
-5	-3	100	100
-3	-1.5	100	150
-1.5	-0.5	100	175
-0.5	0	75	200
0	0.5	25	250
0.5	Inf	0	0

Parameters

- **`- numpy array of precipitation values [mm]`** (*`precipitation`*) –
- **`- array of temperature values, use dew point temperature`** (*`temperature`*) –
- **`available [degrees C]`** (*`if`*) –

Returns

Return type

dictionary

- **perc_snow** (*numpy.array*) **- Percent of the precipitation that is snow** in values 0.0-1.0.
- **rho_s** (*numpy.array*) - Snow density values in kg/m^3.

smrf.envphys.storms module

Created on March 14, 2017 Originally written by Scott Havens in 2015 @author: Micah Johnson

`smrf.envphys.storms.`**`clip_and_correct`**(*precip*, *storms*, *stations=[]*)

Meant to go along with the storm tracking, we correct the data here by adding in the precip we would miss by ignoring it. This is mostly because will get rain on snow events when there is snow because of the storm definitions and still try to distribute precip data.

Parameters

- **`precip`** – Vector station data representing the measured precipitation
- **`storms`** – Storm list with dictionaries as defined in *`tracking_by_station()`*

- **stations** – Desired stations that are being used for clipping. If stations is not passed, then use all in the dataframe

Returns The correct precip that ensures there is no precip outside of the defined storms with the clipped amount of precip proportionally added back to storms.

Created May 3, 2017 @author: Micah Johnson

smrf.envphys.storms.**storms**(*precipitation*, *perc_snow*, *mass=1*, *time=4*, *stormDays=None*, *stormPrecip=None*, *ps_thresh=0.5*)

Calculate the decimal days since the last storm given a precip time series, percent snow, mass threshold, and time threshold

- Will look for pixels where perc_snow > 50% as storm locations
- **A new storm will start if the mass at the pixel has exceeded the mass** limit, this ensures that the enough has accumulated

Parameters

- **precipitation** – Precipitation values
- **perc_snow** – Precent of precipitation that was snow
- **mass** – Threshold for the mass to start a new storm
- **time** – Threshold for the time to start a new storm
- **stormDays** – If specified, this is the output from a previous run of storms
- **stormPrecip** – Keeps track of the total storm precip

Returns

- **stormDays - Array representing the days since the last storm at** a pixel
- **stormPrecip - Array representing the precip accumulated during** the most recent storm

Return type tuple

Created April 17, 2015 @author: Scott Havens

smrf.envphys.storms.**time_since_storm**(*precipitation*, *perc_snow*, *time_step=0.041666666666666664*, *mass=1.0*, *time=4*, *stormDays=None*, *stormPrecip=None*, *ps_thresh=0.5*)

Calculate the decimal days since the last storm given a precip time series, percent snow, mass threshold, and time threshold

- Will look for pixels where perc_snow > 50% as storm locations
- **A new storm will start if the mass at the pixel has exceeded the mass** limit, this ensures that the enough has accumulated

Parameters

- **precipitation** – Precipitation values
- **perc_snow** – Percent of precipitation that was snow
- **time_step** – Step in days of the model run
- **mass** – Threshold for the mass to start a new storm
- **time** – Threshold for the time to start a new storm

- **stormDays** – If specified, this is the output from a previous run of storms else it will be set to the date_time value
- **stormPrecip** – Keeps track of the total storm precip

Returns

- **stormDays - Array representing the days since the last storm at** a pixel
- **stormPrecip - Array representing the precip accumulated during** the most recent storm

Return type tuple

Created Janurary 5, 2016 @author: Scott Havens

`smrf.envphys.storms.`**`time_since_storm_basin`**(*precipitation, storm, stormid, storming, time, time_step=0.041666666666666664, stormDays=None*)

Calculate the decimal days since the last storm given a precip time series, days since last storm in basin, and if it is currently storming

- Will assign uniform decimal days since last storm to every pixel

Parameters

- **precipitation** – Precipitation values
- **storm** – current or most recent storm
- **time_step** – step in days of the model run
- **last_storm_day_basin** – time since last storm for the basin
- **stormid** – ID of current storm
- **storming** – if it is currently storming
- **time** – current time
- **stormDays** – unifrom days since last storm on pixel basis

Returns unifrom days since last storm on pixel basis

Return type stormDays

Created May 9, 2017 @author: Scott Havens modified by Micah Sandusky

`smrf.envphys.storms.`**`time_since_storm_pixel`**(*precipitation, dpt, perc_snow, storming, time_step=0.041666666666666664, stormDays=None, mass=1.0, ps_thresh=0.5*)

Calculate the decimal days since the last storm given a precip time series

- Will assign decimal days since last storm to every pixel

Parameters

- **precipitation** – Precipitation values
- **dpt** – dew point values
- **perc_snow** – percent_snow values
- **storming** – if it is stomring
- **time_step** – step in days of the model run

- **stormDays** – unifrom days since last storm on pixel basis
- **mass** – Threshold for the mass to start a new storm
- **ps_thresh** – Threshold for percent_snow

Returns days since last storm on pixel basis

Return type stormDays

Created October 16, 2017 @author: Micah Sandusky

smrf.envphys.storms.**tracking_by_basin**(*precipitation*, *time*, *storm_lst*, *time_steps_since_precip*, *is_storming*, *mass_thresh=0.01*, *steps_thresh=2*)

Parameters

- **precipitation** – precipitation values
- **time** – Time step that smrf is on
- **time_steps_since_precip** – time steps since the last precipitation
- **storm_lst** – list that store the storm cycles in order. A storm is recorded by its start and its end. The list is passed by reference and modified internally. Each storm entry should be in the format of: [{start:Storm Start, end:Storm End}]

  e.g. [{start:date_time1,end:date_time2}, {start:date_time3,end:date_time4},]

  #would be a two storms
- **mass_thresh** – mass amount that constitutes a real precip event, default = 0.0.
- **steps_thresh** – Number of time steps that constitutes the end of a precip event, default = 2 steps (typically 2 hours)

Returns storm_lst - updated storm_lst time_steps_since_precip - updated time_steps_since_precip is_storming - True or False whether the storm is ongoing or not

Return type tuple

Created March 3, 2017 @author: Micah Johnson

smrf.envphys.storms.**tracking_by_station**(*precip*, *mass_thresh=0.01*, *steps_thresh=3*)

Processes the vector station data prior to the data being distributed

Parameters

- **precipitation** – precipitation values
- **time** – Time step that smrf is on
- **time_steps_since_precip** – time steps since the last precipitation
- **storm_lst** – list that store the storm cycles in order. A storm is recorded by its start and its end. The list is passed by reference and modified internally. Each storm entry should be in the format of: [{start:Storm Start, end:Storm End}]

  e.g. [{start:date_time1,end:date_time2,'BOG1':100, 'ATL1':85}, {start:date_time3,end:date_time4,'BOG1':50, 'ATL1':45},]

  #would be a two storms at stations BOG1 and ATL1
- **mass_thresh** – mass amount that constitutes a real precip event, default = 0.01.
- **steps_thresh** – Number of time steps that constitutes the end of a precip event, default = 2 steps (typically 2 hours)

Returns

- **storms - A list of dictionaries containing storm start,stop,** mass accumulated, of given storm.
- **storm_count** - A total number of storms found

Return type tuple

Created April 24, 2017 @author: Micah Johnson

smrf.envphys.sunang module

`smrf.envphys.sunang.`**`dsign`**`(`*a*, *b*`)`
modified from /usr/src/lib/libF77/d_sign.c

`smrf.envphys.sunang.`**`ephemeris`**`(`*dt*`)`
Calculates radius vector, declination, and apparent longitude of sun, as function of the given date and time.

The routine is adapted from:

W. H. Wilson, Solar ephemeris algorithm, Reference 80-13, 70

pp., Scripps Institution of Oceanography, University of California, San Diego, La Jolla, CA, 1980.

Parameters **`dt`** – date time python object

Returns solar declination angle, in radians omega: sun longitude, in radians r: Earth-Sun radius vector

Return type declin

`smrf.envphys.sunang.`**`leapyear`**`(`*year*`)`
leapyear determines if the given year is a leap year or not. year must be positive, and must not be abbreviated; i.e. 89 is 89 A.D. not 1989.

Parameters **`year`** –

Returns True if a leap year, False if not a leap year

`smrf.envphys.sunang.`**`numdays`**`(`*year*, *month*`)`
numdays returns the number of days in the given month of the given year.

Parameters

- **`year`** –
- **`month`** –

Returns number of days in month

Return type ndays

`smrf.envphys.sunang.`**`rotate`**`(`*mu*, *azm*, *mu_r*, *lam_r*`)`
Calculates new spherical coordinates if system rotated about origin. Coordinates are right-hand system. All angles are in radians.

Parameters

- **`mu`** – cosine of angle theta from z-axis in old coordinate system, sin(declination)
- **`azm`** – azimuth (+ccw from x-axis) in old coordinate system, hour angle of sun (long. where sun is vertical)

- **mu_r** – cosine of angle theta_r of rotation of z-axis, sin(latitude)
- **lam_r** – azimuth (+ccw) of rotation of x-axis, longitude

Returns cosine of the solar zenith aPrime: solar azimuth in radians

Return type muPrime

`smrf.envphys.sunang.`**`sunang`**(*date_time*, *latitude*, *longitude*, *truncate=True*)

Calculate the sun angle (the azimuth and zenith angles of the sun's position) for a given geodetic location for a single date time and coordinates. The function can take either latitude longitude position as a single point or numpy array.

Parameters

- **date_time** – python datetime object
- **latitude** – value or np.ndarray (in degrees)
- **longitude** – value or np.ndarray (in degrees)
- **truncate** – True will replicate the IPW output precision, not applied if position is an array

Returns cosz - cosine of the zenith angle, same shape as input position azimuth - solar azimuth, same shape as input position rad_vec - Earth-Sun radius vector

`smrf.envphys.sunang.`**`sunang_thread`**(*queue*, *date*, *lat*, *lon*)

See sunang for input descriptions

Parameters

- **queue** – queue with cosz, azimuth
- **date** – loop through dates to accesss queue, must be same as rest of queues

`smrf.envphys.sunang.`**`sunpath`**(*latitude*, *longitude*, *declination*, *omega*)

Sun angle from solar declination and longtitude

Parameters

- **latitude** – in radians
- **longitude** – in radians
- **declination** – solar declination (radians)
- **omega** – solar longitude (radians)

Returns cosz: cosine of solar zenith azimuth: solar azimuth in radians

`smrf.envphys.sunang.`**`yearday`**(*year*, *month*, *day*)

yearday returns the yearday for the given date. yearday is the 'day of the year', sometimes called (incorrectly) 'julian day'.

Parameters

- **year** –
- **month** –
- **day** –

Returns day of year

Return type yday

smrf.envphys.thermal_radiation module

The module contains various physics calculations needed for estimating the thermal radition and associated values.

`smrf.envphys.thermal_radiation.Angstrom1918(ta, ea)`

Estimate clear-sky downwelling long wave radiation from Angstrom (1918) [13] as cited by Niemela et al (2001) [14] using the equation:

$$\epsilon_{clear} = 0.83 - 0.18 * 10^{-0.067 e_a}$$

Where e_a is the vapor pressure.

Parameters

- **ta** – distributed air temperature [degree C]
- **ea** – distrubted vapor pressure [kPa]

Returns clear sky long wave radiation [W/m2]

20170509 Scott Havens

`smrf.envphys.thermal_radiation.Crawford1999(th, ta, cloud_factor)`

Cloud correction is based on Crawford and Duchon (1999) [18]

$$\epsilon_a = (1 - cloud\_factor) + cloud\_factor * \epsilon_{clear}$$

where $cloud\_factor$ is the ratio of measured solar radiation to the clear sky irradiance.

Parameters

- **th** – clear sky thermal radiation [W/m2]
- **ta** – temperature in Celcius that the clear sky thermal radiation was calcualted from [C]
- **cloud_factor** – fraction of sky that are not clouds, 1 equals no clouds, 0 equals all clouds

Returns cloud corrected clear sky thermal radiation

20170515 Scott Havens

`smrf.envphys.thermal_radiation.Dilly1998(ta, ea)`

Estimate clear-sky downwelling long wave radiation from Dilley & O'Brian (1998) [11] using the equation:

$$L_{clear} = 59.38 + 113.7 * \left(\frac{T_a}{273.16} \right)^6 + 96.96 \sqrt{w/25}$$

Where T_a is the air temperature and w is the amount of precipitable water. The preipitable water is estimated as $4650 e_o / T_o$ from Prata (1996) [12].

Parameters

- **ta** – distributed air temperature [degree C]
- **ea** – distrubted vapor pressure [kPa]

Returns clear sky long wave radiation [W/m2]

20170509 Scott Havens

`smrf.envphys.thermal_radiation.Garen2005(th, cloud_factor)`

Cloud correction is based on the relationship in Garen and Marks (2005) [15] between the cloud factor and measured long wave radiation using measurement stations in the Boise River Basin.

$$L_{cloud} = L_{clear} * (1.485 - 0.488 * cloud\_factor)$$

Parameters

- **th** – clear sky thermal radiation [W/m2]
- **cloud_factor** – fraction of sky that are not clouds, 1 equals no clouds, 0 equals all clouds

Returns cloud corrected clear sky thermal radiation

20170515 Scott Havens

`smrf.envphys.thermal_radiation.`**`Kimball1982`**(*th*, *ta*, *ea*, *cloud_factor*)

Cloud correction is based on Kimball et al. (1982) [17]

$$
\begin{aligned}
L_d &= L_{clear} + \tau_8 c f_8 \sigma T_c^4 \\
\tau_8 &= 1 - \epsilon_{8z}(1.4 - 0.4\epsilon_{8z}) \\
\epsilon_{8z} &= 0.24 + 2.98 \times 10^{-6} e_o^2 exp(3000/T_o) \\
f_8 &= -0.6732 + 0.6240 \times 10^{-2} T_c - 0.9140 \times 10^{-5} T_c^2
\end{aligned}
$$

where the original Kimball et al. (1982) [17] was for multiple cloud layers, which was simplified to one layer. T_c is the cloud temperature and is assumed to be 11 K cooler than T_a.

Parameters

- **th** – clear sky thermal radiation [W/m2]
- **ta** – temperature in Celcius that the clear sky thermal radiation was calcualted from [C]
- **ea** – distrubted vapor pressure [kPa]
- **cloud_factor** – fraction of sky that are not clouds, 1 equals no clouds, 0 equals all clouds

Returns cloud corrected clear sky thermal radiation

20170515 Scott Havens

`smrf.envphys.thermal_radiation.`**`Prata1996`**(*ta*, *ea*)

Estimate clear-sky downwelling long wave radiation from Prata (1996) [12] using the equation:

$$\epsilon_{clear} = 1 - (1 + w) * exp(-1.2 + 3w)^{1/2}$$

Where w is the amount of precipitable water. The preipitable water is estimated as $4650 e_o/T_o$ from Prata (1996) [12].

Parameters

- **ta** – distributed air temperature [degree C]
- **ea** – distrubted vapor pressure [kPa]

Returns clear sky long wave radiation [W/m2]

20170509 Scott Havens

`smrf.envphys.thermal_radiation.`**`Unsworth1975`**(*th*, *ta*, *cloud_factor*)

Cloud correction is based on Unsworth and Monteith (1975) [16]

$$\epsilon_a = (1 - 0.84)\epsilon_{clear} + 0.84c$$

where $c = 1 - cloud\_factor$

Parameters

- **th** – clear sky thermal radiation [W/m2]

- **ta** – temperature in Celcius that the clear sky thermal radiation was calcualted from [C] cloud_factor: fraction of sky that are not clouds, 1 equals no clouds, 0 equals all clouds

Returns cloud corrected clear sky thermal radiation

20170515 Scott Havens

`smrf.envphys.thermal_radiation.`**`brutsaert`**(*ta*, *l*, *ea*, *z*, *pa*)

Calculate atmosphere emissivity from Brutsaert (1975):cite:*Brutsaert:1975*

Parameters

- **ta** – air temp (K)
- **l** – temperature lapse rate (deg/m)
- **ea** – vapor pressure (Pa)
- **z** – elevation (z)
- **pa** – air pressure (Pa)

Returns atmosphericy emissivity

20151027 Scott Havens

`smrf.envphys.thermal_radiation.`**`calc_long_wave`**(*e*, *ta*)

Apply the Stephan-Boltzman equation for longwave

`smrf.envphys.thermal_radiation.`**`hysat`**(*pb*, *tb*, *L*, *h*, *g*, *m*)

integral of hydrostatic equation over layer with linear temperature variation

Parameters

- **pb** – base level pressure
- **tb** – base level temp [K]
- **L** – lapse rate [deg/km]
- **h** – layer thickness [km]
- **g** – grav accel [m/s^2]
- **m** – molec wt [kg/kmole]

Returns hydrostatic results

20151027 Scott Havens

`smrf.envphys.thermal_radiation.`**`precipitable_water`**(*ta*, *ea*)

Estimate the precipitable water from Prata (1996) [12]

`smrf.envphys.thermal_radiation.`**`sati`**(*tk*)

saturation vapor pressure over ice. From IPW sati

Parameters **tk** – temperature in Kelvin

Returns saturated vapor pressure over ice

20151027 Scott Havens

`smrf.envphys.thermal_radiation.`**`satw`**(*tk*)

Saturation vapor pressure of water. from IPW satw

Parameters **tk** – temperature in Kelvin

Returns saturated vapor pressure over water

20151027 Scott Havens

`smrf.envphys.thermal_radiation.`**`thermal_correct_canopy`**(*th*, *ta*, *tau*, *veg_height*, *height_thresh=2*)

Correct thermal radiation for vegitation. It will only correct for pixels where the veg height is above a threshold. This ensures that the open areas don't get this applied. Vegitation temp is assumed to be at air temperature

Parameters

- **`th`** – thermal radiation
- **`ta`** – air temperature [C]
- **`tau`** – transmissivity of the canopy
- **`veg_height`** – vegitation height for each pixel
- **`height_thresh`** – threshold hold for height to say that there is veg in the pixel

Returns corrected thermal radiation

Equations from Link and Marks 1999 [8]

20150611 Scott Havens

`smrf.envphys.thermal_radiation.`**`thermal_correct_terrain`**(*th*, *ta*, *viewf*)

Correct the thermal radiation for terrain assuming that the terrain is at the air temperature and the pixel and a sky view

Parameters

- **`th`** – thermal radiation
- **`ta`** – air temperature [C]
- **`viewf`** – sky view factor from view_f

Returns corrected thermal radiation

20150611 Scott Havens

`smrf.envphys.thermal_radiation.`**`topotherm`**(*ta*, *tw*, *z*, *skvfac*)

Calculate the clear sky thermal radiation. topotherm calculates thermal radiation from the atmosphere corrected for topographic effects, from near surface air temperature Ta, dew point temperature DPT, and elevation. Based on a model by Marks and Dozier (1979) :citeL`Marks&Dozier:1979`.

Parameters

- **`ta`** – air temperature [C]
- **`tw`** – dew point temperature [C]
- **`z`** – elevation [m]
- **`skvfac`** – sky view factor

Returns Long wave (thermal) radiation corrected for terrain

20151027 Scott Havens

Module contents

1.3.4 smrf.framework package

Submodules

smrf.framework.model_framework module

The module *model_framework* contains functions and classes that act as a major wrapper to the underlying packages and modules contained with SMRF. A class instance of *SMRF* is initialized with a configuration file indicating where data is located, what variables to distribute and how, where to output the distributed data, or run as a threaded application. See the help on the configuration file to learn more about how to control the actions of *SMRF*.

Example

The following examples shows the most generic method of running SMRF. These commands will generate all the forcing data required to run iSnobal. A complete example can be found in run_smrf.py

```
>>> import smrf
>>> s = smrf.framework.SMRF(configFile) # initialize SMRF
>>> s.loadTopo() # load topo data
>>> s.initializeDistribution() # initialize the distribution
>>> s.initializeOutput() # initialize the outputs if desired
>>> s.loadData() # load weather data  and station metadata
>>> s.distributeData() # distribute
```

class smrf.framework.model_framework.**SMRF**(*config*, *external_logger=None*)

Bases: `object`

SMRF - Spatial Modeling for Resources Framework

Parameters **configFile** (*str*) – path to configuration file.

Returns SMRF class instance.

start_date
start_date read from configFile

end_date
end_date read from configFile

date_time
Numpy array of date_time objects between start_date and end_date

config
Configuration file read in as dictionary

distribute
Dictionary the contains all the desired variables to distribute and is initialized in `initializeDistribution()`

create_distributed_threads()
Creates the threads for a distributed run in smrf. Designed for smrf runs in memory

Returns t: list of threads for distirbution q: queue

distributeData()
Wrapper for various distribute methods. If threading was set in configFile, then *distributeData_threaded()* will be called. Default will call *distributeData_single()*.

distributeData_single()

Distribute the measurement point data for all variables in serial. Each variable is initialized first using the *smrf.data.loadTopo.Topo()* instance and the metadata loaded from *loadData()*. The function distributes over each time step, all the variables below.

Steps performed:

1. Sun angle for the time step
2. Illumination angle
3. Air temperature
4. Vapor pressure
5. Wind direction and speed
6. Precipitation
7. Cloud Factor
8. Solar radiation
9. Thermal radiation
10. Soil temperature
11. Output time step if needed

distributeData_threaded()

Distribute the measurement point data for all variables using threading and queues. Each variable is initialized first using the *smrf.data.loadTopo.Topo()* instance and the metadata loaded from *loadData()*. A *DateQueue* is initialized for *all threading variables*. Each variable in *smrf.distribute()* is passed all the required point data at once using the distribute_thread function. The distribute_thread function iterates over *date_time* and places the distributed values into the *DateQueue*.

initializeDistribution()

This initializes the distirbution classes based on the configFile sections for each variable. *initializeDistribution()* will initialize the variables within the *smrf.distribute()* package and insert into a dictionary 'distribute' with variable names as the keys.

Variables that are intialized are:

- *Air temperature*
- *Vapor pressure*
- *Wind speed and direction*
- *Precipitation*
- *Albedo*
- *Solar radiation*
- *Thermal radiation*
- *Soil Temperature*

initializeOutput()

Initialize the output files based on the configFile section ['output']. Currently only *NetCDF files* is supported.

loadData()

Load the measurement point data for distributing to the DEM, must be called after the distributions are initialized. Currently, data can be loaded from three different sources:

- *CSV files*
- *MySQL database*
- *Gridded data source (WRF)*

After loading, loadData() will call *smrf.framework.model_framework.find_pixel_location()* to determine the pixel locations of the point measurements and filter the data to the desired stations if CSV files are used.

loadTopo(*calcInput=True*)

Load the information from the configFile in the ['topo'] section. See *smrf.data.loadTopo.Topo()* for full description.

modules = ['air_temp', 'albedo', 'precip', 'soil_temp', 'solar', 'cloud_factor', 'therm

output(*current_time_step*, *module=None*, *out_var=None*)

Output the forcing data or model outputs for the current_time_step.

Parameters

- **current_time_step** (*date_time*) – the current time step datetime object
- **–** (*var_name*) –
- **–** –

post_process()

Execute all the post processors

thread_variables = ['cosz', 'azimuth', 'illum_ang', 'air_temp', 'dew_point', 'vapor_pr

title(*option*)

A little title to go at the top of the logger file

smrf.framework.model_framework.**can_i_run_smrf**(*config*)

Function that wraps run_smrf in try, except for testing purposes

Parameters **config** – string path to the config file or inicheck UserConfig instance

smrf.framework.model_framework.**find_pixel_location**(*row*, *vec*, *a*)

Find the index of the stations X/Y location in the model domain

Parameters

- **row** (*pandas.DataFrame*) – metadata rows
- **vec** (*nparray*) – Array of X or Y locations in domain
- **a** (*str*) – Column in DataFrame to pull data from (i.e. 'X')

Returns Pixel value in vec where row[a] is located

smrf.framework.model_framework.**run_smrf**(*config*)

Function that runs smrf how it should be operate for full runs.

Parameters **config** – string path to the config file or inicheck UserConfig instance

Module contents

1.3.5 smrf.output package

Submodules

smrf.output.output_hru module

Functions to output the gridded data for a HRU

class smrf.output.output_hru.**output_hru**(*variable_list*, *topo*, *date_time*, *config*)
Bases: object

Class output_hru() to output values to a HRU dataframe, then to a file

date_cols = ['year', 'month', 'day', 'hour', 'minute', 'second']

fmt = '%Y-%m-%d %H:%M:%S'

generate_prms_header()
Generate the header for the PRMS output file

output(*variable*, *data*, *date_time*)
Output a time step

Parameters

- **variable** – variable name that will index into variable list
- **data** – the variable data
- **date_time** – the date time object for the time step

smrf.output.output_netcdf module

Functions to output as a netCDF

class smrf.output.output_netcdf.**output_netcdf**(*variable_list*, *topo*, *time*, *outConfig*)
Bases: object

Class output_netcdf() to output values to a netCDF file

cs = (6, 10, 10)

fmt = '%Y-%m-%d %H:%M:%S'

output(*variable*, *data*, *date_time*)
Output a time step

Parameters

- **variable** – variable name that will index into variable list
- **data** – the variable data
- **date_time** – the date time object for the time step

type = 'netcdf'

Module contents

1.3.6 smrf.spatial package

Subpackages

smrf.spatial.dk package

Submodules

smrf.spatial.dk.detrended_kriging module

Compiling dk's kriging function

20160205 Scott Havens

`smrf.spatial.dk.detrended_kriging.`**`call_grid`**(*ad*, *dgrid*, *ndarray elevations*, *ndarray weights*, *int nthreads=1*)

Call the function krige_grid in krige.c which will iterate over the grid within the C code

Parameters

- **– [nsta x nsta] matrix of distances between stations** (*ad*) –
- **– [ngrid x nsta] matrix of distances between grid points and stations** (*dgrid*) –
- **– [nsta] array of station elevations** (*elevations*) –
- **weights** (*return*) –
- **– number of threads to use in parallel processing** (*nthreads*) –

Out: weights changed in place

20160222 Scott Havens

smrf.spatial.dk.detrended_kriging module

Compiling dk's kriging function

20160205 Scott Havens

`smrf.spatial.dk.detrended_kriging.`**`call_grid`**(*ad*, *dgrid*, *ndarray elevations*, *ndarray weights*, *int nthreads=1*)

Call the function krige_grid in krige.c which will iterate over the grid within the C code

Parameters

- **– [nsta x nsta] matrix of distances between stations** (*ad*) –
- **– [ngrid x nsta] matrix of distances between grid points and stations** (*dgrid*) –
- **– [nsta] array of station elevations** (*elevations*) –
- **weights** (*return*) –
- **– number of threads to use in parallel processing** (*nthreads*) –

Out: weights changed in place

20160222 Scott Havens

smrf.spatial.dk.dk module

2016-02-22 Scott Havens

Distributed forcing data over a grid using detrended kriging

class `smrf.spatial.dk.dk.`**`DK`**(*mx*, *my*, *mz*, *GridX*, *GridY*, *GridZ*, *config*)

Bases: `object`

Detrended kriging class

`calculate`(*data*)

Calcluate the deternded kriging for the data and config

Arg: data: numpy array same length as m* config: configuration for dk

Returns returns the distributed and calculated value

Return type v

`calculateWeights`()

Calculate the weights given those stations with nan values for data

`detrendData`(*data*)

Detrend the data in val using the heights zmeas data - is the same size at mx,my flag - 1 for positive, -1 for negative, 0 for any trend imposed

`retrendData`(*r*)

Retrend the residual values

Module contents

Submodules

smrf.spatial.grid module

2016-03-07 Scott Havens

Distributed forcing data over a grid using interpolation

class `smrf.spatial.grid.`**`GRID`**(*config*, *mx*, *my*, *GridX*, *GridY*, *mz=None*, *GridZ=None*, *mask=None*, *metadata=None*)

Bases: `object`

Inverse distance weighting class - Standard IDW - Detrended IDW

`calculateInterpolation`(*data*, *grid_method='linear'*)

Interpolate over the grid

Parameters

- **`data`** – data to interpolate
- **`mx`** – x locations for the points
- **`my`** – y locations for the points

- **X** – x locations in grid to interpolate over
- **Y** – y locations in grid to interpolate over

detrendedInterpolation(*data*, *flag=0*, *grid_method='linear'*)
Interpolate using a detrended approach

Parameters

- **data** – data to interpolate
- **grid_method** – scipy.interpolate.griddata interpolation method

detrendedInterpolationLocal(*data*, *flag=0*, *grid_method='linear'*)
Interpolate using a detrended approach

Parameters

- **data** – data to interpolate
- **grid_method** – scipy.interpolate.griddata interpolation method

detrendedInterpolationMask(*data*, *flag=0*, *grid_method='linear'*)
Interpolate using a detrended approach

Parameters

- **data** – data to interpolate
- **grid_method** – scipy.interpolate.griddata interpolation method

smrf.spatial.idw module

class smrf.spatial.idw.**IDW**(*mx*, *my*, *GridX*, *GridY*, *mz=None*, *GridZ=None*, *power=2*, *zeroVal=-1*)
Bases: object

Inverse distance weighting class for distributing input data. Availables options are:

- Standard IDW
- Detrended IDW

calculateDistances()
Calculate the distances from the measurement locations to the grid locations

calculateIDW(*data*, *local=False*)
Calculate the IDW of the data at mx,my over GridX,GridY Inputs: data - is the same size at mx,my

calculateWeights()
Calculate the weights for

detrendData(*data*, *flag=0*, *zeros=None*)
Detrend the data in val using the heights zmeas data - is the same size at mx,my flag - 1 for positive, -1 for negative, 0 for any trend imposed

detrendedIDW(*data*, *flag=0*, *zeros=None*, *local=False*)
Calculate the detrended IDW of the data at mx,my over GridX,GridY Inputs: data - is the same size at mx,my

retrendData(*idw*)
Retrend the IDW values

smrf.spatial.kriging module

class smrf.spatial.kriging.**KRIGE**(*mx*, *my*, *mz*, *GridX*, *GridY*, *GridZ*, *config*)
Bases: `object`

Kriging class based on the pykrige package

calculate(*data*)
Estimate the variogram, calculate the model, then apply to the grid

Arg: data: numpy array same length as m* config: configuration for dk

Returns

Z-values of specified grid or at thespecified set of points. If style was specified as 'masked', zvalues will be a numpy masked array.

sigmasq: Variance at specified grid points or at the specified set of points. If style was specified as 'masked', sigmasq will be a numpy masked array.

Return type v

detrendData(*data*, *flag=0*, *zeros=None*)
Detrend the data in val using the heights zmeas

Parameters

- **data** – is the same size at mx,my
- **flag** –
  - 1 for positive, -1 for negative, 0 for any trend imposed

Returns data minus the elevation trend

retrendData(*idw*)
Retrend the IDW values

Module contents

1.3.7 smrf.utils package

Subpackages

smrf.utils.wind package

Submodules

smrf.utils.wind.model module

class smrf.utils.wind.model.**wind_model**(*x*, *y*, *dem*, *nthreads=1*)
Bases: `object`

Estimating wind speed and direction is complex terrain can be difficult due to the interaction of the local topography with the wind. The methods described here follow the work developed by Winstral and Marks (2002) and Winstral et al. (2009) [19] [20] which parameterizes the terrain based on the upwind direction. The underlying method calulates the maximum upwind slope (maxus) within a search distance to determine if a cell is sheltered or exposed.

The azimuth **A** is the direction of the prevailing wind for which the maxus value will be calculated within a maximum search distance **dmax**. The maxus (**Sx**) parameter can then be estimated as the maximum value of the slope from the cell of interest to all of the grid cells along the search vector. The efficiency in selection of the maximum value can be increased by using the techniques from the horizon functio which calculates the horizon for each pixel. Therefore, less calculations can be performed. Negative **Sx** values indicate an exposed pixel location (shelter pixel was lower) and positive **Sx** values indicate a sheltered pixel (shelter pixel was higher).

After all the upwind direction are calculated, the average **Sx** over a window is calculated. The average **Sx** accounts for larger lanscape obsticles that may be adjacent to the upwind direction and affect the flow. A window size in degrees takes the average of all **Sx**.

Parameters

- **x** – array of x locations
- **y** – array of y locations
- **dem** – matrix of the dem elevation values
- **nthread** – number of threads to use for maxus calculation

bresenham (*start*, *end*)

Python implementation of the Bresenham algorthim to find all the pixels that a line between start and end interscet

Parameters

- **start** – list of start point
- **end** – list of end point

Returns Array path of all points between start and end

find_maxus (*index*)

Calculate the maxus given the start and end point

Parameters **index** – index to a point in the array

Returns maxus value for the point

hord (*x*, *y*, *z*)

Calculate the horizon pixel for all z This mimics the simple algorthim from Dozier 1981 but was adapated for use in finding the maximum upwind slope

Works backwards from the end but looks forwards for the horizon

Parameters

- **x** – x locations for the points
- **y** – y locations for the points
- **z** – elevations for the points

Returns array of the horizon index for each point

ismember (*a*, *b*)

maxus (*dmax*, *inc=5*, *inst=2*, *out_file='smrf_maxus.nc'*)

Calculate the maxus values

Parameters

- **dmax** – length of outlying upwind search vector (meters)
- **inc** – increment between direction calculations (degrees)

- **inst** – Anemometer height (meters)
- **out_file** – NetCDF file for output results

Returns None, outputs maxus array straight to file

maxus_angle (*angle*, *dmax*)
Calculate the maxus for a single direction for a search distance dmax

---

Note: This will produce different results than the original maxus program. The differences are due to:

1. Using dtype=double for the elevations
2. Using different type of search method to find the endpoints.

However, if the elevations are rounded to integers, the cardinal directions will reproduce the original results.

Parameters

- **angle** – middle upwind direction around which to run model (degrees)
- **dmax** – length of outlying upwind search vector (meters)

Returns array of maximum upwind slope values within dmax

Return type maxus

output (*ptype*, *index*)
Output the data into the out file that has previously been initialized.

Parameters

- **ptype** – type of calculation that will be saved, either 'maxus' or 'tbreak'
- **index** – index into the file for where to place the output

output_init (*ptype*, *filename*, *ex_att=None*)
Initialize a NetCDF file for outputing the maxus values or tbreak

Parameters

- **ptype** – type of calculation that will be saved, either 'maxus' or 'tbreak'
- **filename** – filename to save the output into
- **ex_att** – extra attributes to add

tbreak (*dmax*, *sepdist*, *inc=5*, *inst=2*, *out_file='smrf_tbreak.nc'*)
Calculate the topobreak values

Parameters

- **dmax** – length of outlying upwind search vector (meters)
- **sepdist** – length of local max upwind slope search vector (meters)
- **angle** – middle upwind direction around which to run model (degrees)
- **inc** – increment between direction calculations (degrees)
- **inst** – Anemometer height (meters)
- **out_file** – NetCDF file for output results

Returns None, outputs maxus array straight to file

windower (*maxus_file*, *window_width*, *wtype*)
Take the maxus output and average over the window width

Parameters

- **maxus_file** – location of the previously calculated maxus values
- **window_width** – window width about the wind direction
- **wtype** – type of wind calculation 'maxus' or 'tbreak'

Returns New file containing the windowed values

smrf.utils.wind.wind_c module

Cython wrapper to the underlying C code

20160816

smrf.utils.wind.wind_c.**call_maxus**()
Call the function maxus_grid in calc_wind.c which will iterate over the grid within the C code

Parameters

- **- [nsta x nsta] matrix of distances between stations** (*ad*) –
- **- [ngrid x nsta] matrix of distances between grid points and stations** (*dgrid*) –
- **- [nsta] array of station elevations** (*elevations*) –
- **weights** (*return*) –
- **- number of threads to use in parallel processing** (*nthreads*) –

Out: weights changed in place

20160222 Scott Havens

smrf.utils.wind.wind_c module

Cython wrapper to the underlying C code

20160816

smrf.utils.wind.wind_c.**call_maxus**()
Call the function maxus_grid in calc_wind.c which will iterate over the grid within the C code

Parameters

- **- [nsta x nsta] matrix of distances between stations** (*ad*) –
- **- [ngrid x nsta] matrix of distances between grid points and stations** (*dgrid*) –
- **- [nsta] array of station elevations** (*elevations*) –
- **weights** (*return*) –
- **- number of threads to use in parallel processing** (*nthreads*) –

Out: weights changed in place

20160222 Scott Havens

Module contents

Submodules

smrf.utils.gitinfo module

smrf.utils.gradient module

`smrf.utils.gradient.`**`aspect`**(*dz_dx*, *dz_dy*)

Calculate the aspect from the finite difference. Aspect is degrees clockwise from North (0/360 degrees)

See below for a referance to how ArcGIS calculates slope http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/How_Aspect_works/00q900000023000000/

Parameters

- **`dz_dx`** – finite difference in the x direction
- **`dz_dy`** – finite difference in the y direction

Returns aspect in degrees

`smrf.utils.gradient.`**`aspect_to_ipw_radians`**(*a*)

IPW defines aspect differently than most GIS programs so convert an aspect in degrees from due North (0/360) to the IPW definition.

Aspect is radians from south (aspect 0 is toward the south) with range from -pi to pi, with negative values to the west and positive values to the east

Parameters **`a`** – aspect in degrees from due North

Returns a: aspect in radians from due South

`smrf.utils.gradient.`**`gradient_d4`**(*dem*, *dx*, *dy*, *aspect_rad=False*)

Calculate the slope and aspect for provided dem, this will mimic the original IPW gradient method that does a finite difference in the x/y direction

Given a center cell e and it's neighbors:

a | b | c |
d | e | f |
g | h | i |

The rate of change in the x direction is [dz/dx] = (f - d) / (2 * dx)

The rate of change in the y direction is [dz/dy] = (h - b) / (2 * dy)

The slope is calculated as slope_radians = arctan (sqrt ([dz/dx]^2 + [dz/dy]^2))

Parameters

- **`dem`** – array of elevation values
- **`dx`** – cell size along the x axis

- **dy** – cell size along the y axis
- **aspect_rad** – turn the aspect from degrees to IPW radians

Returns slope in radians aspect in degrees or IPW radians

smrf.utils.gradient.**gradient_d8**(*dem*, *dx*, *dy*, *aspect_rad=False*)

Calculate the slope and aspect for provided dem, using a 3x3 cell around the center

Given a center cell e and it's neighbors:

a | b | c |
d | e | f |
g | h | i |

The rate of change in the x direction is [dz/dx] = ((c + 2f + i) - (a + 2d + g) / (8 * dx)

The rate of change in the y direction is [dz/dy] = ((g + 2h + i) - (a + 2b + c)) / (8 * dy)

The slope is calculated as slope_radians = arctan (sqrt ([dz/dx]^2 + [dz/dy]^2))

Parameters

- **dem** – array of elevation values
- **dx** – cell size along the x axis
- **dy** – cell size along the y axis
- **aspect_rad** – turn the aspect from degrees to IPW radians

Returns slope in radians aspect in degrees or IPW radians

smrf.utils.io module

Input/Output functions Adapted from the UW-Hydro tonic project

smrf.utils.io.**isbool**(*x*)

Test if str is an bolean

smrf.utils.io.**isfloat**(*x*)

Test if value is a float

smrf.utils.io.**isint**(*x*)

Test if value is an integer

smrf.utils.io.**isscalar**(*x*)

Test if a value is a scalar

smrf.utils.pycompat module

smrf.utils.pycompat.**iteritems**(*d*)

smrf.utils.pycompat.**itervalues**(*d*)

smrf.utils.queue module

Create classes for running on multiple threads

20160323 Scott Havens

class smrf.utils.queue.**DateQueue_Threading**(*maxsize=0*, *timeout=None*, *name=None*)

Bases: queue.Queue

DateQueue extends Queue.Queue module Stores the items in a dictionary with date_time keys When values are retrieved, it will not remove them and will require cleaning at the end to not have to many values

20160323 Scott Havens

clean(*index*)

Need to clean it out so mimic the original get

get(*index*, *block=True*, *timeout=None*)

Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

This is from queue.Queue but with modifcation for supplying what to get

Parameters

- **index** – datetime object representing the date/time being processed
- **block** – boolean determining whether to wait for a variable to become available
- **timeout** – Number of seconds to wait before dropping error, none equates to forever.

put(*item*, *block=True*, *timeout=None*)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

class smrf.utils.queue.**QueueCleaner**(*date_time*, *queue*)

Bases: threading.Thread

QueueCleaner that will go through all the queues and check if they all have a date in common. When this occurs, all the threads will have processed that time step and it's not longer needed

run()

Go through the date times and look for when all the queues have that date_time

class smrf.utils.queue.**QueueOutput**(*queue*, *date_time*, *out_func*, *out_frequency*, *nx*, *ny*)

Bases: threading.Thread

Takes values from the queue and outputs using 'out_func'

run()

Output the desired variables to a file.

Go through the date times and look for when all the queues have that date_time

smrf.utils.utils module

20160104 Scott Havens

Collection of utility functions

class smrf.utils.utils.**CheckStation**(***kwargs*)

Bases: inicheck.checkers.CheckType

Custom check for ensuring our stations are always capitalized

type_func(*value*)

Attempt to convert all the values to upper case.

Parameters **value** – A single string in a a config entry representing a station name

Returns A single station name all upper case

Return type value

smrf.utils.utils.**backup_input**(*data*, *config_obj*)

Backs up input data files so a user can rerun a run with the exact data used for a run.

Parameters

- **data** – Pandas dataframe containing the station data
- **config_obj** – The config object produced by inicheck

smrf.utils.utils.**check_station_colocation**(*metadata_csv=None*, *metadata=None*)

Takes in a data frame representing the metadata for the weather stations as produced by *smrf.framework.model_framework.SMRF.loadData* and check to see if any stations have the same location.

Parameters

- **metadata_csv** – CSV containing the metdata for weather stations
- **metadata** – Pandas Dataframe containing the metdata for weather stations

Returns list of station primary_id that are colocated

Return type repeat_sta

smrf.utils.utils.**find_configs**(*directory*)

Searches through a directory and returns all the .ini fulll filenames.

Parameters **directory** – string path to directory.

Returns list of paths pointing to the config file.

Return type configs

smrf.utils.utils.**getConfigHeader**()

Generates string for inicheck to add to config files

Returns string for cfg headers

Return type cfg_str

smrf.utils.utils.**get_asc_stats**(*fp*)

Returns header of ascii dem file

`smrf.utils.utils.`**`get_config_doc_section_hdr`**`()`
Returns the header dictionary for linking modules in smrf to the documentation generated by inicheck auto doc functions

`smrf.utils.utils.`**`getgitinfo`**`()`
gitignored file that contains specific SMRF version and path

Returns git version from 'git describe'

Return type str

`smrf.utils.utils.`**`getqotw`**`()`

`smrf.utils.utils.`**`grid_interpolate`**`(`*values*, *vtx*, *wts*, *shp*, *fill_value=nan*`)`
Broken out gridded interpolation from scipy.interpolate.griddata that takes the vertices and wts from interp_weights function

Parameters

- **`values`** – flattened WindNinja wind speeds
- **`vtx`** – vertices for interpolation
- **`wts`** – weights for interpolation
- **`shape`** – shape of SMRF grid
- **`fill_value`** – value for extrapolated points

Returns interpolated values

Return type ret

`smrf.utils.utils.`**`grid_interpolate_deconstructed`**`(`*tri*, *values*, *grid_points*, *method='linear'*`)`
Underlying methods from scipy grid_data broken out to pass in the tri values returned from qhull.Delaunay. This is done to improve the speed of using grid_data

Parameters

- **`tri`** – values returned from qhull.Delaunay
- **`values`** – values at HRRR stations generally
- **`grid_points`** – tuple of vectors for X,Y coords of grid stations
- **`method`** – either linear or cubic

Returns result of interpolation to gridded points

`smrf.utils.utils.`**`handle_run_script_options`**`(`*config_option*`)`
Handle function for dealing with args in the SMRF run script

Parameters **`config_option`** – string path to a directory or a specific config file.

Returns Full path to an existing config file.

Return type configFile

`smrf.utils.utils.`**`interp_weights`**`(`*xy*, *uv*, *d=2*`)`
Find vertices and weights of LINEAR interpolation for gridded interp. This routine follows the methods of scipy.interpolate.griddata as outlined here: https://stackoverflow.com/questions/20915502/speedup-scipy-griddata-for-multiple-interpolations-between-two-irregular-grids This function finds the vertices and weights which is the most computationally expensive part of the routine. The interpolateion can then be done quickly.

Parameters

- **xy** – n by 2 array of flattened meshgrid x and y coords of WindNinja grid
- **uv** – n by 2 array of flattened meshgrid x and y coords of SMRF grid
- **d** – dimensions of array (i.e. 2 for our purposes)

Returns wts:

Return type vertices

`smrf.utils.utils.`**`is_leap_year`**`(`*year*`)`

`smrf.utils.utils.`**`nan_helper`**`(`*y*`)`

Helper to handle indices and logical indices of NaNs.

Example

```
>>> # linear interpolation of NaNs
>>> nans, x= nan_helper(y)
>>> y[nans]= np.interp(x(nans), x(~nans), y[~nans])
```

Parameters **y** – 1d numpy array with possible NaNs

Returns **nans** - logical indices of NaNs **index** - a function

Return type tuple

`smrf.utils.utils.`**`set_min_max`**`(`*data*, *min_val*, *max_val*`)`

Ensure that the data is in the bounds of min and max

Parameters

- **data** – numpy array of data to be min/maxed
- **min_val** – minimum threshold to trim data
- **max_val** – Maximum threshold to trim data

Returns numpy array of data trimmed at min_val and max_val

Return type data

`smrf.utils.utils.`**`water_day`**`(`*indate*`)`

Determine the decimal day in the water year

Parameters **indate** – datetime object

Returns **dd** - decimal day from start of water year **wy** - Water year

Return type tuple

20160105 Scott Havens

Module contents

1.4 References

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[1] Danny Marks, Jeff Dozier, and Robert E. Davis. Climate and Energy Exchange at the Snow Surface in the Alpine Region of the Sierra Nevada 1. Meteorolocical Measurements and Monitoring. *Water Resources Research*, 28(11)(92):3029–3042, 1992.

[2] Danny Marks, John Kimball, Dave Tingey, and Tim Link. The sensitivity of snowmelt processes to climate conditions and forest during rain on snow: a case study of the 1996 Pacific Northwest flood. *Hydrological Processes*, 1587(March):1569–1587, 1998.

[3] Patrick R Kormos, Danny Marks, James P McNamara, H P Marshall, Adam Winstral, and Alejandro N Flores. Snow distribution, melt and surface water inputs to the soil in the mountain rain-snow transition zone. *Journal of Hydrology*, 519, Part(0):190–204, 2014. URL: http://www.sciencedirect.com/science/article/pii/S0022169414005113, doi:http://dx.doi.org/10.1016/j.jhydrol.2014.06.051.

[4] David Susong, Danny Marks, and David Garen. Methods for developing time-series climate surfaces to drive topographically distributed energy- and water-balance models. *Hydrological Processes*, 13(May 1998):2003–2021, 1999. URL: ftp://ftp.nwrc.ars.usda.gov/publications/1999/Marks-Methodsfordevelopingtime-seriesclimatesurfacestodrivetopographicallydistributedenergy-andwater-balancemodels.pdf, doi:10.1002/(SICI)1099-1085(199909)13:12/13<2003::AID-HYP884>3.0.CO;2-K.

[5] Jeff Dozier. A Clear-Sky Spectral Solar Radiation Model. *Water Resources Research*, 16(4):709–718, 1980. URL: http://fiesta.bren.ucsb.edu/\protect\T1\textbraceleft~\protect\T1\textbracerightdozier/Pubs/WR016i004p00709.pdf, doi:10.1029/WR016i004p00709.

[6] Jeff Dozier and James Frew. Atmospheric corrections to satellite radiometric data over rugged terrain. *Remote Sensing of Environment*, 11(C):191–205, 1981. doi:10.1016/0034-4257(81)90019-5.

[7] Ralph C. Dubayah. Modeling a solar radiation topoclimatology for the Rio Grande River Basin. *Journal of Vegetation Science*, 5(5):627–640, 1994. doi:10.2307/3235879.

[8] Timothy Link and Danny Marks. Distributed simulation of snowcover mass- and energy-balance in the boreal forest. *Hydrological Processes*, 13(April 1998):2439–2452, 1999. URL: http://doi.wiley.com/10.1002/(SICI)1099-1085(199910)13:14/15\protect\T1\textbraceleft\T1\textbackslash{ }textless\protect\T1\textbraceright2439::AID-HYP866\protect\T1\textbraceleft\T1\textbackslash{ }textgreater\protect\T1\textbraceright3.0.CO;2-1, doi:10.1002/(sici)1099-1085(199910)13:14/15<2439::aid-hyp866>3.0.co;2-1.

[9] G. N. Flerchinger, Wei Xaio, Danny Marks, T. J. Sauer, and Qiang Yu. Comparison of algorithms for incoming atmospheric long-wave radiation. *Water Resources Research*, 45(3):n/a–n/a, 2009. W03423. URL: http://dx.doi.org/10.1029/2008WR007394, doi:10.1029/2008WR007394.

[10] Danny Marks and Jeff Dozier. A clear-sky longwave radiation model for remote alpine areas. *Archiv for Meteorologie, Geophysik und Bioklimatologie Serie B*, 27(2-3):159–187, 1979. doi:10.1007/BF02243741.

[11] A. C. Dilley and D. M. O'Brien. Estimating downward clear sky long-wave irradiance at the surface from screen temperature and precipitable water. *Quarterly Journal of the Royal Meteorological Society*, 124(549):1391–1401, 1998. URL: http://dx.doi.org/10.1002/qj.49712454903, doi:10.1002/qj.49712454903.

[12] A. J. Prata. A new long-wave formula for estimating downward clear-sky radiation at the surface. *Quarterly Journal of the Royal Meteorological Society*, 122(533):1127–1151, 1996. URL: http://dx.doi.org/10.1002/qj.49712253306, doi:10.1002/qj.49712253306.

[13] A. Ångström. A study of the radiation of the atmosphere. *Smithson. Misc. Collect.*, 65:1–159, 1918.

[14] Sami Niemelä, Petri Räisänen, and Hannu Savijärvi. Comparison of surface radiative flux parameterizations: part i: longwave radiation. *Atmospheric Research*, 58(1):1 – 18, 2001. URL: http://www.sciencedirect.com/science/article/pii/S0169809501000849, doi:https://doi.org/10.1016/S0169-8095(01)00084-9.

[15] David C. Garen and Danny Marks. Spatially distributed energy balance snowmelt modelling in a mountainous river basin: Estimation of meteorological inputs and verification of model results. *Journal of Hydrology*, 315(1-4):126–153, 2005. doi:10.1016/j.jhydrol.2005.03.026.

[16] M. H. Unsworth and J. L. Monteith. Long-wave radiation at the ground i. angular distribution of incoming radiation. *Quarterly Journal of the Royal Meteorological Society*, 101(427):13–24, 1975. URL: http://dx.doi.org/10.1002/qj.49710142703, doi:10.1002/qj.49710142703.

[17] B. A. Kimball, S. B. Idso, and J. K. Aase. A model of thermal radiation from partly cloudy and overcast skies. *Water Resources Research*, 18(4):931–936, 1982. URL: http://dx.doi.org/10.1029/WR018i004p00931, doi:10.1029/WR018i004p00931.

[18] Todd M. Crawford and Claude E. Duchon. An improved parameterization for estimating effective atmospheric emissivity for use in calculating daytime downwelling longwave radiation. *Journal of Applied Meteorology*, 38(4):474–480, 1999. arXiv:http://dx.doi.org/10.1175/1520-0450(1999)038<0474:AIPFEE>2.0.CO;2, doi:10.1175/1520-0450(1999)038<0474:AIPFEE>2.0.CO;2.

[19] Adam Winstral, Kelly Elder, and Robert E. Davis. Spatial Snow Modeling of Wind-Redistributed Snow Using Terrain-Based Parameters. *Journal of Hydrometeorology*, 3(5):524–538, 2002. doi:10.1175/1525-7541(2002)003<0524:SSMOWR>2.0.CO;2.

[20] Adam Winstral, Danny Marks, and Robert Gurney. An efficient method for distributing wind speeds over heterogeneous terrain. *Hydrological Processes*, 23(17):2526–2535, 2009. doi:10.1002/hyp.7141.

[21] JP Hardy, R Melloh, P Robinson, R Jordan, and others. Incorporating effects of forest litter in a snow process model. *Hydrological Processes*, 14(18):3227–3237, 2000.

[22] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, 517–524. New York, NY, USA, 1968. ACM. doi:10.1145/800186.810616.

[23] David C Garen, Gregory L Johnson, and Clayton L Hanson. Mean Areal Precipitation for Daily Hydrologic Modeling in Mountainous Regions. *Journal of the American Water Resources Association*, 30(3):481–491, 1994. doi:10.1111/j.1752-1688.1994.tb03307.x.

[24] Richard Webster and Margaret A. Oliver. *Geostatistics for Environmental Scientists*. John Wiley & Sons, Ltd, 2008. ISBN 9780470517277. doi:10.1002/9780470517277.

[25] Wilfried Brutsaert. NOTES AND CORRESPONDENCE Comments on Surface Roughness Parameters and the Height of Dense Vegetation. *Journal of the Meteorological Society of Japan*, 1974.

[26] J Cataldo and M Zeballos. Roughness terrain consideration in a wind interpolation numerical model. In *11th Americas Conference on Wind Engineering—San Juan, Puerto Rico*. 2009.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y