Stereographic Cloud Base Reconstruction Documentation

Release 1.0.0

Florian M. Savoy, Soumyabrata Dev, Yee Hui Lee and Stefan Winkl

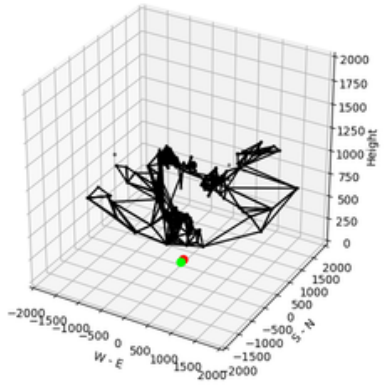
May 09, 2017

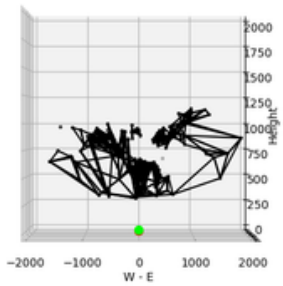
Contents

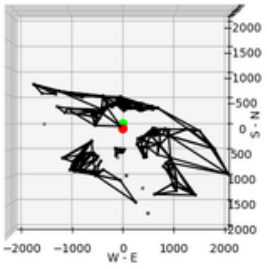
This is the documentation for the algorithm presented in the publication "Stereographic Cloud Base Reconstruction using High-Resolution Whole Sky Imagers". Reference to the paper will be provided here upon publication.

This algorithm estimates 3D models of the base of clouds in real world coordinates. It uses two images captured at the same time by two whole sky imagers positioned about 100 meters away. The resulting models have applications in meteorology, solar energy prediction or satellite communication signal attenuation.

The figure below illustrates the algorithm. Both top images are the input images captured at the same time by two sky imagers about 100 meters apart. The left image has been segmented, and displays the feature points used for the reconstruction in green as well as the triangulation in red. The three figures at the bottom show the reconstructed model, from 3D, side and top views. The axis refer to real world coordinates in meters. Please refer to the publication for further details.







CHAPTER 1

Code

The code is available on github at https://github.com/FSavoy/cloud-base-reconstruction. It consists of three classes encapsulating functions regarding sky imagers, segmentation and the reconstruction itself. A sample file which generates the figure shown above is provided too.

The documentation is organized as follows:

Classes

Three classes encapsulate the code logic:

Imager

class imager.**Imager**(*name*, *center*, *radius*, *position*, *rot*, *trans*, *longitude=None*, *latitude=None*, *elevation=None*)

Describes an imager and all the parameters related to it. Contains the methods dealing with lens distortions and mis-alignment correction.

name
str – Name of the imager

center
list[int] – List of two ints describing the center of the fish-eye lens in the captured pictures

radius
int – The radius of the circle created by the fish-eye lens

position
numpy.array – The coordinates of the relative position of the imager in real world coordinates (meters)

rot
numpy.array – A rotation matrix (3x3) to cope with the misalignement of the image

trans
numpy.array – A translation matrix (3x1) to cope with the misalignement of the image

longitude
str – The longitude coordinate of the imager (optional, only needed for segmentation)

latitude
str – The latitude coordinate of the imager (optional, only needed for segmentation)

elevation
int – The elevation of the integer in meters (optional, only needed for segmentation)

cam2world(*m*)
Returns the 3D coordinates with unit length corresponding to the pixels locations on the image.

Parameters **m** (*numpy.array*) – 2 x m matrix containing m pixel coordinates

Returns 3 x m matrix of the corresponding unit sphere vectors

Return type numpy.array

undistort_image(*image, sizeResult=[1000, 1000], altitude=500, bearing=0, elevation=0, pixelWidth=1*)
Undistort a fish-eye image to a standard camera image by a ray-tracing approach.

Parameters

- **image** (*numpy.array*) – the fish-eye image to undistort
- **sizeResult** (*list[int]*) – 2D size of the output undistorted image [pixels]
- **altitude** (*int*) – the altitude of the virtual image plane to form the undistorted image [meters]
- **bearing** (*float*) – the bearing angle of the undistorted plane (in degrees)
- **elevation** (*float*) – the elevation angle of the undistorted plane (in degrees)
- **pixelWidth** (*int*) – the width in real world coordinates of one pixel of the undistorted image [meters]

world2cam(*M*)
Returns the pixel locations corresponding to the given 3D points.

Parameters **M** (*numpy.array*) – 3 x m matrix containing M point coordinates

Returns 2 x m matrix of the corresponding pixel coordinates

Return type numpy.array

Segmentation

class segmentation.**Segmentation**(*image*, *imager*, *time*)
Contains all the methods needed for the segmentation a sky/cloud image

image
np.array – The image to segment

imager
Imager – The imager used to take the image

time
datetime.datetime – The time at which the image was captured (UTC+8)

cmask(*index*, *radius*, *array*)
Generates the mask for a given input image. The generated mask is needed to remove occlusions during post-processing steps.

Parameters

- **index** (*numpy array*) – Array containing the x- and y- co-ordinate of the center of the circular mask.
- **radius** (*float*) – Radius of the circular mask.
- **array** (*numpy array*) – Input sky/cloud image for which the mask is generated.

Returns Generated mask image.

Return type numpy array

getBRchannel(*input_image*, *mask_image*)
Extracts the ratio of red and blue blue channel from an input sky/cloud image. It is used in the clustering step to generate the binary sky/cloud image.

Parameters

- **input_image** (*numpy array*) – Input sky/cloud image.
- **mask_image** (*numpy array*) – Mask to remove occlusions from the input image. This mask contains boolean values indicating the allowable pixels from an image.

Returns Ratio image using red and blue color channels, normalized to [0,255].

Return type numpy array

make_cluster_mask(*input_matrix*, *mask_image*)
Clusters an input sky/cloud image to generate the binary image and the coverage ratio value.

Parameters

- **input_matrix** (*numpy array*) – Input matrix that needs to be normalized.
- **mask_image** (*numpy array*) – Mask to remove occlusions from the input image. This mask contains boolean values indicating the allowable pixels from an image.

Returns Binary output image, where white denotes cloud pixels and black denotes sky pixels. float: Cloud coverage ratio in the input sky/cloud image. float: The first (out of two) cluster center. float: The second (out of two) cluster center.

Return type numpy array

removeSunIfNonCloudy(*im_mask*)
Removes the circumsolar area from the mask if it is detected as non cloudy.

Parameters **im_mask** (*numpy array*) – Segmentation mask

Returns Modified segmentation mask.

Return type numpy array

segment()
Main function for sky/cloud segmentation. Perform all the necessary steps.

showasImage(*input_matrix*)
Normalizes an input matrix to the range [0,255]. It is useful in displaying the matrix as an image.

Parameters **input_matrix** (*numpy array*) – Input matrix that needs to be normalized.

Returns Returns the normalized matrix.

Return type numpy array

Reconstruction

class reconstruction.**Reconstruction**(*imager1, imager2, time, sizeUndistorted=[1000, 1000, 3], altitudes=[500], pixelWidths=[1], tilts=[[0, 0], [35, 0], [35, 90], [35, 180], [35, 270]]*)

Contains all the methods for creating a 3D reconstruction from the base height from two sky pictures taken at the same time. Most functions need to be applied in the correct order to complete the workflow. See main.py for an example.

imager1
Imager – Imager used to take the first image

imager2
Imager – Imager used to take the second image

time
datetime.datetime – Time at which the images were captured

sizeUndistorted
list[int] – Size of the undistorted planes

altitudes
list[int] – Altitudes of the undistorted planes

pixelWidths
list[int] – Widths in meters of one pixel of the undistorted plane

tilts
list[list[int] – List of angles [elevation, azimuth] of the undistorted planes

images
dict[list[numpy.array]] – Dictionary (over each imager) of arrays of LDR images captured by that imager

undistorted
dict[list[numpy.array]] – Dictionary of list of LDR undistorted planes for each combination of undistortion parameters for the first imager

tonemapped
dict[dict[numpy.array]] – Dictionary (over each imager) of dictionaries of tonemapped images for each combination of undistortion parameters for that imager

pts
dict[dict[numpy.array]] – Dictionary (over each imager) of dictionaries of n*2 feature points locations in the undistorted planes of the that imager

rays
dict[dict[numpy.array]] – Dictionary (over each imager) of dictionary of 3*n vectors from the origin at the center of the imager towards each feature point in the planes for the that imager

ptsWorld
dict[numpy.array] – Dictionary of 3*n' coordinates of the reconstructed points, at the midpoint of the shortest segment between both rays from both imagers

ptsWorldMerged
numpy.array – ptsWorld concatenated in a single array independently of undistortion planes

ptsWorldRays
dict[dict[numpy.array]] – Dictionary (over each imager) of dictionary of 3*n' coordinates of the reconstructed points, on the rays originating from the first imager

ptsWorldRaysMerged
dict[dict[numpy.array]] – Dictionary (over each imager) of the reconstructed points in ptsWorldRays (all undistortion planes merged in common arrays)

ptsPictureImager
dict[dict[numpy.array]] – Dictionary (over each imager) of dictionaries of 2*n' vectors of the image plane locations corresponding to each reconstructed points for that imager

ptsPictureImagerMerged
dict[numpy.array] – Dictionary (over each imager) of the image locations in ptsPictureImager (all undistortion planes merged in common arrays)

distances
dict[numpy.array] – Dictionary of n' vectors of the lenghts of the shortest segments between the rays of the first and the second imager

distancesMerged
numpy.array – distances concatenated in a single array independently of undistortion planes

triangulationError
float – Mean lenghth of the shortest segments between rays for each pair of matching features

meanHeight
float – Mean z coordinate of the reconstructed points

stdHeight
float – Standard deviation of the z coordinates of the reconstructed points

nbPts
int – Number of reconstructed points

cloudBaseHeight
float – 10th percentile of the z coordinates of the reconstructed points, assumed to be the cloud base height

segmentationMask
np.array – Sky/cloud segmentation mask for the first imager

triangulation
matplotlib.tri.Triangulation – Delaunay triangulation between the reconstructed points

computeMetrics()
Computes the light rays from the pairs of matching points *concatenateValues()* must be called before.

computeRays()
Computes the light rays from the pairs of matching points *match()* must be called before.

computeTriangulation()
Creates the triangulation between reconstructed points for visualization. *concatenateValues()* must be called before.

concatenateValues()
Concatenates all the computed values from all the different planes into a single vectors. *intersectRays()* must be called before.

intersectRays(*threshold=100*)
Computes the intersection the light rays from the matching pairs *computeRays()* must be called before.

Parameters **threshold** (*int*) – the minimum distance between two rays to discard a match.

iter_planes()
Helper function to iterate over all undistortion plane parameters (altitudes, pixel width and tilts).

Returns over all (altitude, pixel width, tilts) tuples

Return type iterator

load_images(*images1*, *images2*)
Loads the input LDR images captured at the same instant by both imagers.

Parameters

- **images1** (*list[numpy.array]*) – LDR images captured by imager1
- **images2** (*list[numpy.array]*) – LDR images captured by imager2

match()
Creates sift matching reconstruction between a pair of undistorted images. *tonemap()* must be called before.

process()
Executes the entire reconstruction process. Alternatively, every function can be called individually. *load_images()* must be called before.

removeOutliers()
Removes outliers in the set of reconstructed points *intersectRays()* must be called before.

tonemap()
Tonemapps the undistorted planes. *undistort()* must be called before.

undistort()
Generates the undistortion planes. *load_images()* must be called before.

writeFigure(*filename*)
Creates a visualization of the reconstructed elements. `createTriangulation()` must be called before.

- *Imager*: describes an imager and all the parameters related to it. Contains the methods dealing with lens distortions and mis-alignment correction.
- *Segmentation*: contains all the methods needed for the segmentation a sky/cloud image.
- *Reconstruction*: contains all the methods for creating a 3D reconstruction from the base height from two sky pictures taken at the same time.

Sample usage

The file main.py shows a sample usage for the code to generate the reconstruction of cloud base height. Here is a walkthrough the file:

First import the dependencies:

```
from imager import Imager
from reconstruction import Reconstruction
import numpy as np
import cv2
import datetime
```

Then create the first imager object. For the calibration parameters, please refer to the following publication: Georeferencing and stereo calibration of ground-based whole sky imagers using the sun trajectory, F. M. Savoy, S. Dev, Y. H. Lee, S. Winkler, Proc. IEEE International Geoscience and Remote Sensing Symposium (IGARSS), Beijing, China, July 10-15, 2016.

```
name3 = 'wahrsis3'
center3 = [1724, 2592]
radius3 = 1470
relativePosition3 = np.array([0,0,0])
calibRot3 = np.array([[0.99555536, 0.09404159, 0.00506982], [-0.09393761, 0.99541774,␣
↪-0.01786745], [-0.00672686, 0.01731178, 0.99982751]])
calibTrans3 = np.array([[ 0.00552915], [0.00141732], [0.00553584]])
longitude3 = '103:40:49.9'
lattitude3 = '1:20:35'
altitude3 = 59
wahrsis3 = Imager(name3, center3, radius3, relativePosition3, calibRot3, calibTrans3,␣
↪longitude3, lattitude3, altitude3)
```

The second imager does not need the longitude, lattitude and altitude information.

```
name4 = 'wahrsis4'
center4 = [2000, 2975]
radius4 = 1665
relativePosition4 = np.array([-2.334, 101.3731, -8.04])
calibRot4 = np.array([[0.9710936, -0.23401871, 0.04703662], [0.234924, 0.97190314, -0.
↪01466276], [-0.04228367, 0.02528894, 0.99878553]])
calibTrans4 = np.array([[-0.00274625], [-0.00316865], [0.00516088]])
wahrsis4 = Imager(name4, center4, radius4, relativePosition4, calibRot4, calibTrans4)
```

Then load the LDR images in two arrays:

```
images3 = [cv2.imread('wahrsis3/2015-10-29-12-58-01-wahrsis3-low.jpg'),
     cv2.imread('wahrsis3/2015-10-29-12-58-01-wahrsis3-med.jpg'),
     cv2.imread('wahrsis3/2015-10-29-12-58-01-wahrsis3-high.jpg')]
images4 = [cv2.imread('wahrsis4/2015-10-29-12-58-01-wahrsis4-low.jpg'),
     cv2.imread('wahrsis4/2015-10-29-12-58-01-wahrsis4-med.jpg'),
     cv2.imread('wahrsis4/2015-10-29-12-58-01-wahrsis4-high.jpg')]
```

Create the Reconstruction object, load the images, process and write the result to out.png.

```
reconst = Reconstruction(wahrsis3, wahrsis4, datetime.datetime(2015,10,29,12,58,1))
reconst.load_images(images3, images4)
reconst.process()
reconst.writeFigure("out.png")
```

Dependencies

This project was built with python 2.7.12

Furthermore, the following packages were used:

Package	Version	Link
openCV	3.1.0	http://opencv.org/
numpy	1.12.0	http://www.numpy.org/
scipy	0.18.1	https://www.scipy.org/
scikit-image	0.12.3	http://scikit-image.org/
scikit-learn	0.18.1	http://scikit-learn.org/
matplotlib	2.0.0	http://matplotlib.org/
PyEphem	3.7.6.0	http://rhodesmill.org/pyephem/
pycuda (optional)	2016.1.2	https://documen.tician.de/pycuda/

CHAPTER 2

License

This software is released under a 3-Clause BSD License. It is provided “as is” without any warranty. Users are required to obtain the licenses for the required dependencies.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

S

T

U

W