CameraTransform Documentation

Release 1.0

Richard Gerum, Sebastian Richter

Mar 13, 2019

Contents

CameraTransform is a python package which can be used to fit camera transformations and apply them to project points from the camera space to the world space and back.

CHAPTER 1

Installation

First of all download the software from bitbucket:

Download zip package

or clone with Mercurial:

```
hg clone https://bitbucket.org/fabry_biophysics/cameratransform
```

Then open the installed folder and execute the following command in a command line:

```
python setup.py install
```

Note: If you plan to update regularly, e.g. you have cloned repository, you can instead used `python setup.py develop` that will not copy the the package to the python directory, but will use the files in place. This means that you don't have to install the package again if you update the code.

CHAPTER 2

Getting Started

In this section, we will explaine some of the basics of the *CameraTransform* package.

2.1 Using Cameras

The most important object is the :py:class:‘ .Camera‘.

It can be created with different projections (for normal cameras: :py:class:‘ .RectilinearProjection‘ or pamoranic cameras: :py:class:‘ .CylindricalProjection‘, :py:class:‘ .EquirectangularProjection‘).

We define the camera parameters and create a camera with the rectinilar projection (default pin-hole camera) and the spatial orientation, 10m above ground and looking down 45°. Keep in mind the orientation of the tilt angle, where a 0° tilt angle corresponds to a view perpedicular on the ground plane

[1]:
```
import cameratransform as ct

# intrinsic camera parameters
f = 6.2    # in mm
sensor_size = (6.17, 4.55)    # in mm
image_size = (3264, 2448)    # in px

# initialize the camera
cam = ct.Camera(ct.RectilinearProjection(focallength_mm=f,
                                         sensor=sensor_size,
                                         image=image_size),
                ct.SpatialOrientation(elevation_m=10,
                                      tilt_deg=45))
```

We can also omit the SpatialOrientation and later change the parameters:

[2]:
```
cam.elevation_m = 34.027
cam.tilt_deg = 83.307926
cam.roll_deg = -1.916219
```

With the camera object we can project 3D world points (**space** coordinate system, in meters) to the 2D image pixels (**image** coordinate system, in pixels).

[3]:
```
cam.imageFromSpace([3.17, 8, 0])
```

[3]:
```
array([  2513.01903033,  10302.79024379])
```

The routines take lists and 1D numpy arrays or, to transform multiple points at once, stacked lists or 2D numpy arrays. As the calculations are internally handled as vectorized numpy calculations, transforming multiple points in one go is advantagous.

[4]:
```
cam.imageFromSpace([[3.17, 8, 0],
                    [3.17, 8, 10],
                    [4.12, 10, 0],
                    [4.12, 10, 10]])
```

[4]:
```
array([[  2513.01903033,  10302.79024379],
       [  2709.93369714,   8257.53176532],
       [  2613.01526893,   8957.64510958],
       [  2788.36057303,   7108.36064027]])
```

Or the other way around, from **image** to **space**. This transformation is not unique, as the information on how far the object was away from the camera is lost when transforming from **space** to the **image**. Therefore, an additional constraint has to be supplied when performing this back-transformation. The default is to require :math:‘Z=0‘.

[5]:
```
cam.spaceFromImage([2445, 1569])
```

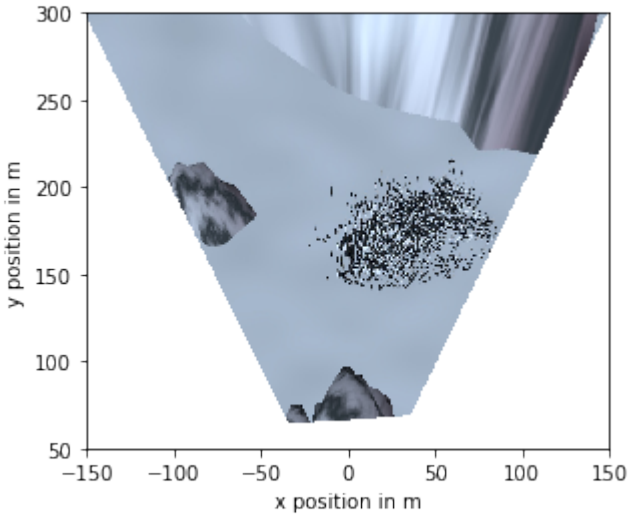
[5]:
```
array([  39.04004539,  156.98250489,    0.        ])
```

[6]:
```
cam.spaceFromImage([2445, 1569], X=10)
```

[6]:
```
array([ 10.        ,  43.4940148 ,  24.58546926])
```

Often it is also advantagouse to have a top vie projection of the camera image. This can be done using the :py:meth:‘Camera.getTopViewOfImage‘ method. The extend of the top view in meters can be specified and the resolution (how many pixels per meter). As a convienience, the image can be plotted directly.

[7]:
```
%matplotlib inline
import matplotlib.pyplot as plt
# display a top view of the image
im = plt.imread("CameraImage.jpg")
top_im = cam.getTopViewOfImage(im, [-150, 150, 50, 300], scaling=0.5, do_plot=True)
plt.xlabel("x position in m")
plt.ylabel("y position in m");
```



 Tip As CameraTransform cannot detect the occlusions, the top view projection is not a real top view, any extended object will cast a "projection shadow" on everything behind it.

2.2 Fitting

This section will cover the basics for estimating the camera parameters from information in the image.

First we define the camera with a focal length and the image size obtained from a loaded image.

[8]:
```
import CameraTransform as ct
import numpy as np
import matplotlib.pyplot as plt

im = plt.imread("CameraImage2.jpg")
camera = ct.Camera(ct.RectilinearProjection(focallength_px=3863.64, image=im))
```

CameraTransform provides different methods to provide the fitting routine with information.

The camera elevation, tilt (and roll) can be fitted using objects of known height in the image that stand on the Z=0 plane. We call the :py:meth:‘.Camera.addObjectHeightInformation‘ method with the pixel positions of the feed points, head points, the height of the objects, and the standard deviation of the height distribution.

[9]:
```
feet = np.array([[1968.73191418, 2291.89125757], [1650.27266115, 2189.75370951],␣
↪[1234.42623164, 2300.56639535], [ 927.4853119 , 2098.87724083], [3200.40162013,␣
↪1846.79042709], [3385.32781138, 1690.86859965], [2366.55011031, 1446.05084045],␣
↪[1785.68269333, 1399.83787022], [ 889.30386193, 1508.92532749], [4107.26569943,␣
↪2268.17045783], [4271.86353701, 1889.93651518], [4007.93773879, 1615.08452509],␣
↪[2755.63028039, 1976.00345458], [3356.54352228, 2220.47263494], [ 407.60113016,␣
↪1933.74694958], [1192.78987735, 1811.07247163], [1622.31086201, 1707.77946355],␣
↪[2416.53943619, 1775.68148688], [2056.81514201, 1946.4146027 ], [2945.35225814,␣
↪1617.28314118], [1018.41322935, 1499.63957113], [1224.2470045 , 1509.87120351],␣
↪[1591.81599888, 1532.33339856], [1701.6226147 , 1481.58276189], [1954.61833888,␣
↪1405.49985098], [2112.99329583, 1485.54970652], [2523.54106057, 1534.87590467],␣
↪[2911.95610793, 1448.87104305], [3330.54617013, 1551.64321531], [2418.21276457,␣
↪1541.28499777], [1771.1651859 , 1792.70568482], [1859.30409241, 1904.01744759],␣
↪[2805.02878512, 1881.00463747], [3138.67003071, 1821.05082989], [3355.45215983,␣
↪1910.47345815], [ 734.28038607, 1815.69614796], [ 978.36733356, 1896.3650782],␣
↪[1350.63202232, 1979.38798787], [3650.89052382, 1901.06620751], [3555.47087822,␣
↪2332.50027861], [ 865.71688784, 1662.27834394], [1115.89438493, 1664.09341647],␣
↪[1558.95825646, 1671.02167477], [1783.86089289, 1679.33599881], [2491.01579305,␣
↪1707.84219953], [3531.26955813, 1729.08486338], [3539.6318973 , 1776.5766387 ],␣
↪[4150.36451427, 1840.90968707], [2112.48684812, 1714.78834459], [2234.65444134,␣
↪1756.17059266]])
```

(continues on next page)

(continued from previous page)

```
heads = np.array([[1968.45971142, 2238.81171866], [1650.27266115, 2142.33767714],␣
↪[1233.79698528, 2244.77321846], [ 927.4853119 , 2052.2539967 ], [3199.94718145,␣
↪1803.46727222], [3385.32781138, 1662.23146061], [2366.63609066, 1423.52398752],␣
↪[1785.68269333, 1380.17615549], [ 889.30386193, 1484.13026407], [4107.73533808,␣
↪2212.98791584], [4271.86353701, 1852.85753597], [4007.93773879, 1586.36656606],␣
↪[2755.89171994, 1938.22544024], [3355.91105749, 2162.91833832], [ 407.60113016,␣
↪1893.63300333], [1191.97371829, 1777.60995028], [1622.11915337, 1678.63975025],␣
↪[2416.31761434, 1743.29549618], [2056.67597009, 1910.09072955], [2945.35225814,␣
↪1587.22557592], [1018.69818061, 1476.70099517], [1224.55272475, 1490.30510731],␣
↪[1591.81599888, 1510.72308329], [1701.45016126, 1460.88834824], [1954.734384  ,␣
↪1385.54008964], [2113.14023137, 1465.41953732], [2523.54106057, 1512.33125811],␣
↪[2912.08384338, 1428.56110628], [3330.40769371, 1527.40984208], [2418.21276457,␣
↪1517.88006678], [1770.94524662, 1761.25436746], [1859.30409241, 1867.88794433],␣
↪[2804.69006305, 1845.10009734], [3138.33130864, 1788.53351052], [3355.45215983,␣
↪1873.21402971], [ 734.49504829, 1780.27688131], [ 978.1022294 , 1853.9484135 ],␣
↪[1350.32991656, 1938.60371039], [3650.89052382, 1863.97713098], [3556.44897343,␣
↪2278.37901052], [ 865.41437575, 1633.53969555], [1115.59187284, 1640.49747358],␣
↪[1558.06918395, 1647.12218082], [1783.86089289, 1652.74740383], [2491.20950909,␣
↪1677.42878081], [3531.11177814, 1696.89774656], [3539.47411732, 1745.49398176],␣
↪[4150.01023142, 1803.35570469], [2112.84669376, 1684.92115685], [2234.65444134,␣
↪1724.86402238]])

camera.addObjectHeightInformation(feet, heads, 0.75, 0.03)
```

If a horizon is visible in the image, it can also be used to fit elevation, tilt (and roll) of the camera. Therefore, we call the :py:meth:‘ .Camera.addHorizonInformation‘ method with the pixel positions of points on the horizon and an uncertaintiy value how many pixels we expect horizon to deviate from the position we provided.

[10]:
```
horizon = np.array([[418.2195998, 880.253216], [3062.54424509, 820.94125636]])
camera.addHorizonInformation(horizon, uncertainty=10)
```

If we know the gps positions of landmarks in the image we can also use them to obtain camera parameters. These also allow to fit the heading angle of the camera. But for gps positions to be meaningful, we have to set the gps position of the camera first (they could also be fitted, if they are unknown).

We call the method :py:meth:‘ .Camera.addLandmarkInformation‘ and provide the pixel coordinates of the landmarks in the image, the positions of these landmarks in the **space** coordinate system, and an uncertainity value, here we use ±3m for latitude and longitude and ±5m for the height information of the landmarks.

[11]:
```
camera.setGPSpos("66°39'53.4\"S  140°00'34.8\"")

lm_points_px = np.array([[2091.300935, 892.072126], [2935.904577, 824.364956]])
lm_points_gps = ct.gpsFromString([("66°39'56.12862''S  140°01'20.39562''", 13.769),
                                  ("66°39'58.73922''S  140°01'09.55709''", 21.143)])
lm_points_space = camera.spaceFromGPS(lm_points_gps)

camera.addLandmarkInformation(lm_points_px, lm_points_space, [3, 3, 5])
```

CameraTransform stores all these information and creates a log probability value from all information. This log probability can now be used to estimate the camera parameters using metropolis sampling (:py:meth:‘ .Camera.metropolis‘).

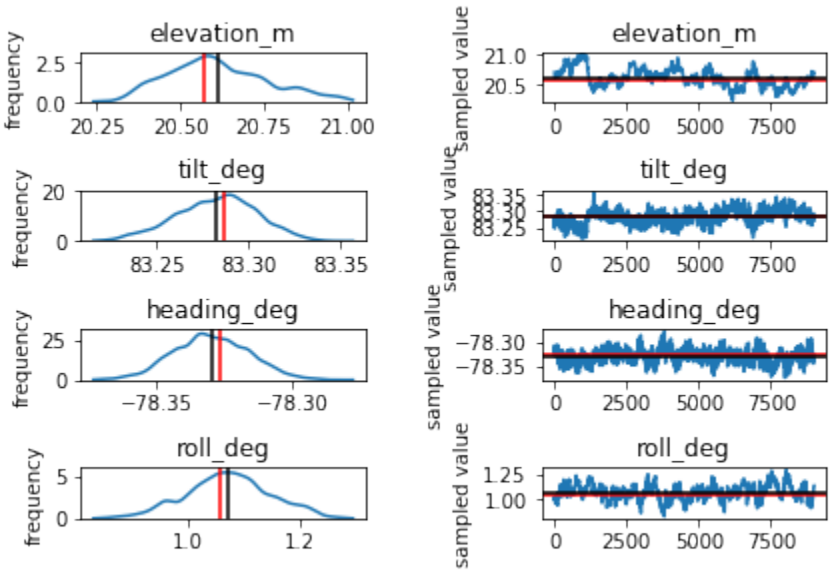
The sampling uses pymc2 statistical objects for the parameters. The parameters need to have the names that the different parts of the camera use. For reference see the chapters on Lens Distortion, Projections, Spacial Orientation, and GPS position.

[12]:
```
trace = camera.metropolis([
        ct.FitParameter("elevation_m", lower=0, upper=100, value=20),
        ct.FitParameter("tilt_deg", lower=0, upper=180, value=80),
        ct.FitParameter("heading_deg", lower=-180, upper=180, value=-77),
        ct.FitParameter("roll_deg", lower=-180, upper=180, value=0)
        ], iterations=1e4)
```

```
  0%|          | 38/10000 [00:00<00:26, 371.21it/s]/home/richard/miniconda3/lib/
↪python3.6/site-packages/scipy/stats/_distn_infrastructure.py:876: RuntimeWarning:␣
↪invalid value encountered in greater_equal
  return (self.a <= x) & (x <= self.b)
/home/richard/miniconda3/lib/python3.6/site-packages/scipy/stats/
↪_distn_infrastructure.py:876: RuntimeWarning: invalid value encountered in␣
↪less_equal
  return (self.a <= x) & (x <= self.b)
100%|| 10000/10000 [00:27<00:00, 369.32it/s, acc_rate=0.202, factor=0.0253]
```

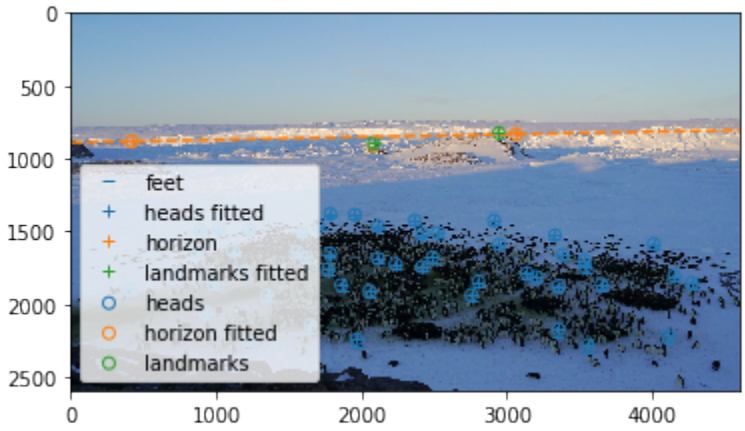
The camera internally stores the trace object and can plot it (:py:meth:‘ .Camera.plotTrace‘). The trace plot show on the left side the estimated probability distribution for every parameter and on the right side the trace through the sampling space. The sampler provided a good result if the traces fluctuate uncorrelatedly around their central value.

[13]:
```
camera.plotTrace()
plt.tight_layout()
```



For a visial estimate how correct the most probable value destribes the provided information, the method :py:meth:‘ .Camera.plotFitInformation‘ can be used. It takes the source image as an argument and uses the stored fit information plot plot all data.

[14]:
```
camera.plotFitInformation(im)
plt.legend();
```


0
500
1000
1500
2000
2500
0
1000
2000
3000
4000
feet
heads fitted
horizon
landmarks fitted
heads
horizon fitted
landmarks

CHAPTER 3

Camera

class CameraTransform.**Camera**(*projection*, *orientation=None*, *lens=None*)

This class is the core of the CameraTransform package and represents a camera. Each camera has a projection (subclass of *CameraProjection*), a spatial orientation (*SpatialOrientation*) and optionally a lens distortion (subclass of LensDistortion).

3.1 Save/Load Functions

Camera.**save**(*filename*)

Saves the camera parameters to a json file.

Parameters **filename** (*str*) – the filename where to store the parameters.

Camera.**load**(*filename*)

Load the camera parameters from a json file.

Parameters **filename** (*str*) – the filename of the file to load.

CameraTransform.**load_camera**(*filename*)

Create a *Camera* instance with the parameters from the file.

Parameters **filename** (*str*) – the filename of the file to load.

Returns **camera** – the camera with the given parameters.

Return type *Camera*

3.2 Transformations

Note: This section only covers transformations from **image** coordinates to **space** coordinates for **gps** coordinates see section *Earth Position (GPS)*.

Camera.**imageFromSpace**(*points*, *hide_backpoints=True*)

Convert points (Nx3) from the **space** coordinate system to the **image** coordinate system.

Parameters **points** (*ndarray*) – the points in **space** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **image** coordinate system, dimensions (2), (Nx2)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,␣
↪2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

transform a single point from the space to the image:

```
>>> cam.imageFromSpace([-4.17, 45.32, 0.])
[1969.52 2209.73]
```

or multiple points in one go:

```
>>> cam.imageFromSpace([[-4.03, 43.96,  0.], [-8.57, 47.91, 0.]]))
[[1971.05 2246.95]
 [1652.73 2144.53]]
```

Camera.**getRay**(*points*, *normed=False*)

As the transformation from the **image** coordinate system to the **space** coordinate system is not unique, **image** points can only be uniquely mapped to a ray in **space** coordinates.

Parameters **points** (*ndarray*) – the points in **image** coordinates for which to get the ray, dimensions (2), (Nx2)

Returns

- **offset** (*ndarray*) – the origin of the camera (= starting point of the rays) in **space** coordinates, dimensions (3)
- **rays** (*ndarray*) – the rays in the **space** coordinate system, dimensions (3), (Nx3)

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,␣
↪2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

get the ray of a point in the image:

```
>>> offset, ray = cam.getRay([1968, 2291]))
>>> offset
[0.00 0.00 15.40]
>>> ray
[-0.09 0.97 -0.35]
```

or the rays of multiple points in the image:

```
>>> offset, ray, cam.getRay([[1968, 2291], [1650, 2189]])
>>> offset
[0.00 0.00 15.40]
>>> ray
[[-0.09 0.97 -0.35]
 [-0.18 0.98 -0.33]]
```

Camera.**spaceFromImage**(*points*, *X=None*, *Y=None*, *Z=0*, *D=None*, *mesh=None*)

Convert points (Nx2) from the **image** coordinate system to the **space** coordinate system. This is not a unique transformation, therefore an additional constraint has to be provided. The X, Y, or Z coordinate(s) of the target points can be provided or the distance D from the camera.

Parameters

- **points** (*ndarray*) – the points in **image** coordinates to transform, dimensions (2), (Nx2)
- **X** (*number, ndarray, optional*) – the X coordinate in **space** coordinates of the target points, dimensions scalar, (N)
- **Y** (*number, ndarray, optional*) – the Y coordinate in **space** coordinates of the target points, dimensions scalar, (N)
- **Z** (*number, ndarray, optional*) – the Z coordinate in **space** coordinates of the target points, dimensions scalar, (N), default 0
- **D** (*number, ndarray, optional*) – the distance in **space** coordinates of the target points from the camera, dimensions scalar, (N)
- **mesh** (*ndarray, optional*) – project the image coordinates onto the mesh in **space** coordinates. The mesh is a list of M triangles, consisting of three 3D points each. Dimensions, (3x3), (Mx3x3)

Returns **points** – the points in the **space** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,␣
↪2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

transform a single point (impliying the condition Z=0):

```
>>> cam.spaceFromImage([1968 , 2291])
[-3.93 42.45 0.00]
```

transform multiple points:

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]])
[[-3.93 42.45 0.00]
 [-8.29 46.11 -0.00]]
```

points that cannot be projected on the image, because they are behind the camera (for the RectilinearProjection) are returned with nan entries:

```
>>> cam.imageFromSpace([-4.17, -10.1, 0.])
[nan nan]
```

specify a y coordinate as for the back projection.

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]], Y=45)
[[-4.17 45.00 -0.93]
 [-8.09 45.00 0.37]]
```

or different y coordinates for each point:

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]], Y=[43, 45])
[[-3.98 43.00 -0.20]
 [-8.09 45.00 0.37]]
```

3.3 Image Transformations

Camera.**undistortImage**(*image*, *extent=None*, *scaling=None*, *do_plot=False*, *alpha=None*, *skip_size_check=False*)

Applies the undistortion of the lens model to the image. The purpose of this function is mainly to check the sanity of a lens transformation. As CameraTransform includes the lens transformation in any calculations, it is not necessary to undistort images before using them.

Parameters

- **image** (*ndarray*) – the image to undistort.
- **extent** (*list, optional*) – the extent in pixels of the resulting image. This can be used to crop the resulting undistort image.
- **scaling** (*number, optional*) – the number of old pixels that are used to calculate a new pixel. A higher value results in a smaller target image.
- **do_plot** (*bool, optional*) – whether to plot the resulting image directly in a matplotlib plot.
- **alpha** (*number, optional*) – when plotting an alpha value can be specified, useful when comparing multiple images.
- **skip_size_check** (*bool, optional*) – if true, the size of the image is not checked to match the size of the cameras image.

Returns **image** – the undistorted image

Return type ndarray

Camera.**getTopViewOfImage**(*image*, *extent=None*, *scaling=None*, *do_plot=False*, *alpha=None*, *Z=0.0*, *skip_size_check=False*)

Project an image to a top view projection. This will be done using a grid with the dimensions of the extent ([x_min, x_max, y_min, y_max]) in meters and the scaling, giving a resolution. For convenience, the image can be plotted directly. The projected grid is cached, so if the function is called a second time with the same parameters, the second call will be faster.

Parameters

- **image** (*ndarray*) – the image as a numpy array.

- **extent** (*list, optional*) – the extent of the resulting top view in meters: [x_min, x_max, y_min, y_max]. If no extent is given a suitable extent is guessed. If a horizon is visible in the image, the guessed extent will in most cases be too streched.
- **scaling** (*number, optional*) – the scaling factor, how many meters is the side length of each pixel in the top view. If no scaling factor is given, a good scaling factor is guessed, trying to get about the same number of pixels in the top view as in the original image.
- **do_plot** (*bool, optional*) – whether to directly plot the resulting image in a matplotlib figure.
- **alpha** (*number, optional*) – an alpha value used when plotting the image. Useful if multiple images should be overlaid.
- **Z** (*number, optional*) – the "height" of the plane on which to project.
- **skip_size_check** (*bool, optional*) – if true, the size of the image is not checked to match the size of the cameras image.

Returns **image** – the top view projected image

Return type ndarray

3.4 Helper Functions

Camera.**distanceToHorizon**()

Calculates the distance of the camera's position to the horizon of the earth. The horizon depends on the radius of the earth and the elevation of the camera.

Returns **distance** – the distance to the horizon.

Return type number

Camera.**getImageHorizon**(*pointsX=None*)

This function calculates the position of the horizon in the image sampled at the points x=0, x=im_width/2, x=im_width.

Parameters **pointsX** (*ndarray, optional*) – the x positions of the horizon to determine, default is [0, image_width/2, image_width], dimensions () or (N)

Returns **horizon** – the points im camera image coordinates of the horizon, dimensions (2), or (Nx2).

Return type ndarray

Camera.**getImageBorder**(*resolution=1*)

Get the border of the image in a top view. Useful for drawing the field of view of the camera in a map.

Parameters **resolution** (*number, optional*) – the pixel distance between neighbouring points.

Returns **border** – the border of the image in **space** coordinates, dimensions (Nx3)

Return type ndarray

Camera.**getCameraCone**(*project_to_ground=False*, *D=1*)

The cone of the camera's field of view. This includes the border of the image and lines to the origin of the camera.

Returns **cone** – the cone of the camera in **space** coordinates, dimensions (Nx3)

Return type ndarray

Camera.**getObjectHeight**(*point_feet*, *point_heads*, *Z=0*)

Calculate the height of objects in the image, assuming the Z position of the objects is known, e.g. they are assumed to stand on the Z=0 plane.

Parameters

- **point_feet** (*ndarray*) – the positions of the feet, dimensions: (2) or (Nx2)
- **point_heads** (*ndarray*) – the positions of the heads, dimensions: (2) or (Nx2)
- **Z** (*number, ndarray, optional*) – the Z position of the objects, dimensions: scalar or (N), default 0

Returns **heights** – the height of the objects in meters, dimensions: () or (N)

Return type ndarray

Camera.**generateLUT**(*undef_value=0*)

Generate LUT to calculate area covered by one pixel in the image dependent on y position in the image

Parameters **undef_value** (*number, optional*) – what values undefined positions should have, default=0

Returns **LUT** – same length as image height

Return type ndarray

CHAPTER 4

Lens Distortions

Tip: Lens distortion transforms from the **distorted** to **image**. Parameters are k_1, k_2, k_3 or a, b, c.

As often the lenses of cameras do not provide a perfect projection on the image plane but introduce some distortions, applications that work with images need to include the distortions of the lens. The distortions are mostly radial distortions, but some use also skew and tangential components. CameraTransform does currently only allow for radial distortion corrections.

To apply the distortion, the coordinates are first centered on the optical axis and scaled using a scale factor, e.g. the focal length. Then the radial component of the coordinates is stretched or shrunken and the resulting coordinates are scaled back to pixels and shifted to have 0,0 at the lower left corner of the image. The distortions are always defined from the flat image to the distorted image. This means an undistortion of the image inverts the formulae.

As CameraTransform can includes the lens correction in the tool chain for projection from the image to the world or the other way around, there is no need to render an undistorted version of each image the is used.

4.1 No Distortion

class CameraTransform.**NoDistortion**
The default model for the lens distortion which does nothing.

4.2 Brown Model

class CameraTransform.**BrownLensDistortion**(*k1=None*, *k2=None*, *k3=None*, *projection=None*)
The most common distortion model is the Brown's distortion model. In CameraTransform, we only consider the radial part of the model, as this covers all common cases and the merit of tangential components is disputed. This model relies on transforming the radius with even polynomial powers in the coefficients k_1, k_2, k_3. This distortion model is e.g. also used by OpenCV or AgiSoft PhotoScan.

Adjust scale and offset of x and y to be relative to the center:

$$x' = \frac{x - c_x}{f_x}$$
$$y' = \frac{y - c_y}{f_y}$$

Transform the radius from the center with the distortion:

$$r = \sqrt{x'^2 + y'^2}$$
$$r' = r \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6)$$
$$x'_{\text{distorted}} = x'/r \cdot r'$$
$$y'_{\text{distorted}} = y'/r \cdot r'$$

Readjust scale and offset to obtain again pixel coordinates:

$$x_{\text{distorted}} = x'_{\text{distorted}} \cdot f_x + c_x$$
$$y_{\text{distorted}} = y'_{\text{distorted}} \cdot f_y + c_y$$

4.3 ABC Model

class CameraTransform.**ABCDistortion**(*a=None*, *b=None*, *c=None*)

The ABC model is a less common distortion model, that just implements radial distortions. Here the radius is transformed using a polynomial of 4th order. It is used e.g. in PTGui.

Adjust scale and offset of x and y to be relative to the center:

$$s = 0.5 \cdot \min(\text{im}_{\text{width}}, \text{im}_{\text{height}})$$
$$x' = \frac{x - c_x}{s}$$
$$y' = \frac{y - c_y}{s}$$

Transform the radius from the center with the distortion:

$$r = \sqrt{x^2 + y^2}$$
$$r' = d \cdot r + c \cdot r^2 + b \cdot r^3 + a \cdot r^4$$
$$d = 1 - a - b - c$$

Readjust scale and offset to obtain again pixel coordinates:

$$x_{\text{distorted}} = x'_{\text{distorted}} \cdot s + c_x$$
$$y_{\text{distorted}} = y'_{\text{distorted}} \cdot s + c_y$$

CHAPTER 5

Projections

Tip: Projections transforms from the **image** to **camera** coordinate system. Parameters are f_x, f_y, c_x, c_y and the image size in px.

This section describes the different projections which are available for the projection of the objects in the **camera coordinate** system to the **image coordinates**.

In the **camera coordinate** system, the camera is positioned at (0,0,0) and is pointing in z direction.

For each projection the projection formula is provided which allows to transform from the **camera coordinate** system (3D) to the **image coordinate** system (2D). As information is lost from transforming from 3D to 2D, the back transformation is not unique. All points that are projected on one image pixel lie on one line or ray. Therefore, for the "backtransformation", only a ray can be provided. To obtain a point this ray has e.g. to be intersected with a plane in the world.

The coordinates $x_{\mathrm{im}}, y_{\mathrm{im}}$ represent a point in the image pixel coordinates, x, y, z the same point in the camera coordinate system. The center of the image is c_x, c_y and f_x and f_y are the focal lengths in pixel (focal length in mm divided by the sensor width/height in mm times the image width/height in pixel), both focal lengths are the same for quadratic pixels on the sensor.

5.1 Parameters

- `focallength_x_px`, f_x: the focal length of the camera relative to the width of a pixel on the sensor.
- `focallength_y_px`, f_y: the focal length of the camera relative to the height of a pixel on the sensor.
- `center_x_px`, c_x: the central point of the image in pixels. Typically about half of the image width in pixels.
- `center_y_px`, c_y: the central point of the image in pixels. Typically about half of the image height in pixels.
- `image_width_px`, $\mathrm{im}_{\mathrm{width}}$: the width of the image in pixels.
- `image_height_px`, $\mathrm{im}_{\mathrm{height}}$: the height of the image in pixels.

5.2 Indirect Parameters

- `focallength_mm`, f_{mm}: the focal length of the camera in mm.
- `sensor_width_mm`, $\mathrm{sensor}_{\mathrm{width}}$: the width of the sensor in mm.
- `sensor_height_mm`, $\mathrm{sensor}_{\mathrm{height}}$: the height of the sensor in mm.
- `view_x_deg`, α_x: the field of view in x direction (width) in degree.
- `view_y_deg`, α_y: the field of view in y direction (height) in degree.

5.3 Functions

class CameraTransform.**CameraProjection**(*focallength_px=None*, *focallength_x_px=None*, *focallength_y_px=None*, *center_x_px=None*, *center_y_px=None*, *center=None*, *focallength_mm=None*, *image_width_px=None*, *image_height_px=None*, *sensor_width_mm=None*, *sensor_height_mm=None*, *image=None*, *sensor=None*, *view_x_deg=None*, *view_y_deg=None*)

Defines a camera projection. The necessary parameters are: focalllength_x_px, focalllength_y_px, center_x_px, center_y_px, image_width_px, image_height_px. Depending on the information available different initialisation routines can be used.

Note: This is the base class for projections. it the should not be instantiated. Available projections are *RectilinearProjection*, *CylindricalProjection*, or *EquirectangularProjection*.

Examples

This section provides some examples how the projections can be initialized.

```
>>> import CameraTransform as ct
```

Image Dimensions:

The image dimensions can be provided as two values:

```
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, image_width_
↪px=4608, image_height_px=3456)
```

or as a tuple:

```
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, image=(4608,␣
↪3456))
```

or by providing a numpy array of an example image:

```
>>> import matplotlib.pyplot as plt
>>> im = plt.imread("test.jpg")
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, image=im)
```

Focal Length:

The focal length can be provided in mm, when also a sensor size is provided:

```
>>> projection = ct.RectilinearProjection(focallength_mm=14, sensor=(17.3, 9.731),
↪ image=(4608, 3456))
```

or directly in pixels without the sensor size:

```
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, image=(4608,␣
↪3456))
```

or as a tuple to give different focal lengths in x and y direction, if the pixels on the sensor are not square:

```
>>> projection = ct.RectilinearProjection(focallength_px=(3772, 3774),␣
↪image=(4608, 3456))
```

or the focal length is given by providing a field of view angle:

```
>>> projection = ct.RectilinearProjection(view_x_deg=61.617, image=(4608, 3456))
```

```
>>> projection = ct.RectilinearProjection(view_y_deg=48.192, image=(4608, 3456))
```

Central Point:

If the position of the optical axis or center of the image is not provided, it is assumed to be in the middle of the image. But it can be specifided, as two values or a tuple:

```
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, center=(2304,␣
↪1728), image=(4608, 3456))
```

```
>>> projection = ct.RectilinearProjection(focallength_px=3863.64, center_x_
↪px=2304, center_y_px=1728, image=(4608, 3456))
```

CameraProjection.**imageFromCamera**(*points*)

Convert points (Nx3) from the **camera** coordinate system to the **image** coordinate system.

Parameters **points** (*ndarray*) – the points in **camera** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **image** coordinate system, dimensions (2), (Nx2)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> proj = ct.RectilinearProjection(focallength_px=3729, image=(4608, 2592))
```

transform a single point from the **camera** coordinates to the image:

```
>>> proj.imageFromCamera([-0.09, -0.27, -1.00])
[2639.61 2302.83]
```

or multiple points in one go:

```
>>> proj.imageFromCamera([[-0.09, -0.27, -1.00], [-0.18, -0.24, -1.00]])
[[2639.61 2302.83]
 [2975.22 2190.96]]
```

CameraProjection.**getRay**(*points*, *normed=False*)

As the transformation from the **image** coordinate system to the **camera** coordinate system is not unique, **image** points can only be uniquely mapped to a ray in **camera** coordinates.

Parameters **points** (*ndarray*) – the points in **image** coordinates for which to get the ray, dimensions (2), (Nx2)

Returns **rays** – the rays in the **camera** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> proj = ct.RectilinearProjection(focallength_px=3729, image=(4608, 2592))
```

get the ray of a point in the image:

```
>>> proj.getRay([1968, 2291])
[0.09 -0.27 -1.00]
```

or the rays of multiple points in the image:

```
>>> proj.getRay([[1968, 2291], [1650, 2189]])
[[0.09 -0.27 -1.00]
 [0.18 -0.24 -1.00]]
```

CameraProjection.**getFieldOfView**()

The field of view of the projection in x (width, horizontal) and y (height, vertical) direction.

Returns

- **view_x_deg** (*float*) – the horizontal field of view in degree.
- **view_y_deg** (*float*) – the vertical field of view in degree.

CameraProjection.**focallengthFromFOV**(*view_x=None*, *view_y=None*)

The focal length (in x or y direction) based on the given field of view.

Parameters

- **view_x** (*float*) – the field of view in x direction in degrees. If not given only view_y is processed.
- **view_y** (*float*) – the field of view in y direction in degrees. If not given only view_y is processed.

Returns **focallength_px** – the focal length in pixels.

Return type float

CameraProjection.**imageFromFOV**(*view_x=None*, *view_y=None*)

The image width or height in pixel based on the given field of view.

Parameters

- **view_x** (*float*) – the field of view in x direction in degrees. If not given only view_y is processed.
- **view_y** (*float*) – the field of view in y direction in degrees. If not given only view_y is processed.

Returns **width/height** – the width or height in pixels.

Return type float

5.4 Projections

All projections share the same interface, as explained above, but implement different image projections.

5.4.1 Rectilinear Projection

class CameraTransform.**RectilinearProjection**(*focallength_px=None*, *focallength_x_px=None*, *focallength_y_px=None*, *center_x_px=None*, *center_y_px=None*, *center=None*, *focallength_mm=None*, *image_width_px=None*, *image_height_px=None*, *sensor_width_mm=None*, *sensor_height_mm=None*, *image=None*, *sensor=None*, *view_x_deg=None*, *view_y_deg=None*)

This projection is the standard "pin-hole", or frame camera model, which is the most common projection for single images. The angles ± 180 are projected to $\pm\infty$. Therefore, the maximal possible field of view in this projection would be 180° for an infinitely large image.

Projection:

$$x_{\mathrm{im}} = f_x \cdot \frac{x}{z} + c_x$$
$$y_{\mathrm{im}} = f_y \cdot \frac{y}{z} + c_y$$

Rays:

$$\vec{r} = \begin{pmatrix} (x_{\mathrm{im}} - c_x)/f_x \\ (y_{\mathrm{im}} - c_y)/f_y \\ 1 \end{pmatrix}$$

Matrix:

The rectilinear projection can also be represented in matrix notation:

$$C_{\mathrm{intr.}} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

5.4.2 Cylindrical Projection

class CameraTransform.**CylindricalProjection**(*focallength_px=None*, *focallength_x_px=None*, *focallength_y_px=None*, *center_x_px=None*, *center_y_px=None*, *center=None*, *focallength_mm=None*, *image_width_px=None*, *image_height_px=None*, *sensor_width_mm=None*, *sensor_height_mm=None*, *image=None*, *sensor=None*, *view_x_deg=None*, *view_y_deg=None*)

This projection is a common projection used for panoranic images. This projection is often used for wide panoramic images, as it can cover the full 360° range in the x-direction. The poles cannot be represented in this projection, as they would be projected to $y = \pm\infty$.

Projection:

$$x_{\mathrm{im}} = f_x \cdot \arctan\left(\frac{x}{z}\right) + c_x$$
$$y_{\mathrm{im}} = f_y \cdot \frac{y}{\sqrt{x^2 + z^2}} + c_y$$

Rays:

$$\vec{r} = \begin{pmatrix} \sin\left(\frac{x_{\mathrm{im}} - c_x}{f_x}\right) \\ \frac{y_{\mathrm{im}} - c_y}{f_y} \\ \cos\left(\frac{x_{\mathrm{im}} - c_x}{f_x}\right) \end{pmatrix}$$

5.4.3 Equirectangular Projection

class CameraTransform.**EquirectangularProjection**(*focallength_px=None*, *focallength_x_px=None*, *focallength_y_px=None*, *center_x_px=None*, *center_y_px=None*, *center=None*, *focallength_mm=None*, *image_width_px=None*, *image_height_px=None*, *sensor_width_mm=None*, *sensor_height_mm=None*, *image=None*, *sensor=None*, *view_x_deg=None*, *view_y_deg=None*)

This projection is a common projection used for panoranic images. The projection can cover the full range of angles in both x and y direction.

Projection:

$$x_{\mathrm{im}} = f_x \cdot \arctan\left(\frac{x}{z}\right) + c_x$$
$$y_{\mathrm{im}} = f_y \cdot \arctan\left(\frac{y}{\sqrt{x^2 + z^2}}\right) + c_y$$

Rays:

$$\vec{r} = \begin{pmatrix} \sin\left(\frac{x_{\mathrm{im}} - c_x}{f_x}\right) \\ \tan\left(\frac{y_{\mathrm{im}} - c_y}{f_y}\right) \\ \cos\left(\frac{x_{\mathrm{im}} - c_x}{f_x}\right) \end{pmatrix}$$

CHAPTER 6

Camera Orientation

Tip: Projections transforms from the **camera** to the **space** coordinate system. Parameters are $\alpha_{\text{tilt}}, \alpha_{\text{roll}}, \alpha_{\text{heading}}$ and the position x, y, z.

Most cameras are not just heading down in z direction but have an orientation in 3D space. Therefore, the module *spatial* allows to transform from **camera coordinates** (3D, origin at the camera, z=distance from camera) to the **space coordinates** (3D, camera can have an arbitrary orientation).

6.1 Parameters

- `heading_deg`, α_{heading}: the direction in which the camera is looking. (0°: the camera faces "north", 90°: east, 180°: south, 270°: west)
- `tilt_deg`, α_{tilt}: the tilt of the camera. (0°: camera faces down, 90°: camera faces parallel to the ground, 180°: camera faces upwards)
- `roll_deg`, α_{roll}: the rotation of the image. (0°: camera image is not rotated (landscape format), 90°: camera image is in portrait format, 180°: camera is in upside down landscape format)
- `pos_x_m`, x: the x position of the camera.
- `pos_y_m`, y: the y position of the camera.
- `elevation_m`, z: the z position of the camera, or the elevation above the xy plane.

6.2 Transformation

class CameraTransform.**SpatialOrientation**(*elevation_m=None*, *tilt_deg=None*, *roll_deg=None*, *heading_deg=None*, *pos_x_m=None*, *pos_y_m=None*)

The orientation can be represented as a matrix multiplication in *projective coordinates*. First, we define rotation

matrices around the three angles: *tilt*, *roll*, *heading*:

$$R_{\text{roll}} = \begin{pmatrix} \cos(\alpha_{\text{roll}}) & \sin(\alpha_{\text{roll}}) & 0 \\ -\sin(\alpha_{\text{roll}}) & \cos(\alpha_{\text{roll}}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$R_{\text{tilt}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha_{\text{tilt}}) & \sin(\alpha_{\text{tilt}}) \\ 0 & -\sin(\alpha_{\text{tilt}}) & \cos(\alpha_{\text{tilt}}) \end{pmatrix}$$
$$R_{\text{heading}} = \begin{pmatrix} \cos(\alpha_{\text{heading}}) & -\sin(\alpha_{\text{heading}}) & 0 \\ \sin(\alpha_{\text{heading}}) & \cos(\alpha_{\text{heading}}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

These angles correspond to ZXZ-Euler angles.

And the position *x*, *y*, *z* (=elevation):

$$t = \begin{pmatrix} x \\ y \\ \text{elevation} \end{pmatrix}$$

We combine the rotation matrices to a single rotation matrix:

$$R = R_{\text{roll}} \cdot R_{\text{tilt}} \cdot R_{\text{heading}}$$

and use this matrix to convert from the **camera coordinates** to the **space coordinates** and vice versa:

$$x_{\text{camera}} = R \cdot (x_{\text{space}} - t)$$
$$x_{\text{space}} = R^{-1} \cdot x_{\text{space}} + t$$

`SpatialOrientation.`**`cameraFromSpace`**(*points*)

Convert points (Nx3) from the **space** coordinate system to the **camera** coordinate system.

Parameters **`points`** (*ndarray*) – the points in **space** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **camera** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> orientation = ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85)
```

transform a single point from the space to the image:

```
>>> orientation.spaceFromCamera([-0.09, -0.27, -1.00])
[0.09 0.97 15.04]
```

or multiple points in one go:

```
>>> orientation.spaceFromCamera([[-0.09, -0.27, -1.00], [-0.18, -0.24, -1.00]])
[[0.09 0.97 15.04]
 [0.18 0.98 15.07]]
```

`SpatialOrientation.`**`spaceFromCamera`**(*points*, *direction=False*)

Convert points (Nx3) from the **camera** coordinate system to the **space** coordinate system.

Parameters

- **points** (*ndarray*) – the points in **camera** coordinates to transform, dimensions (3), (Nx3)
- **direction** (*bool, optional*) – whether to transform a direction vector (used for the rays) which should just be rotated and not translated. Default False

Returns **points** – the points in the **space** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> orientation = ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85)
```

transform a single point from the space to the image:

```
>>> orientation.spaceFromCamera([0.09 0.97 15.04])
[-0.09 -0.27 -1.00]
```

or multiple points in one go:

```
>>> orientation.spaceFromCamera([[0.09, 0.97, 15.04], [0.18, 0.98, 15.07]])
[[-0.09 -0.27 -1.00]
 [-0.18 -0.24 -1.00]]
```

CHAPTER 7

Earth Position (GPS)

Tip: Transforms from the **space** to the **gps** coordinate system. Parameters are $\mathrm{lat}, \mathrm{lon}, \alpha_{\mathrm{heading}}$.

Often the camera is looking at landmarks where the GPS position is known or the GPS position of the camera itself is known and the GPS position of landmarks has to be determined.

7.1 Parameters

- `lat`, lat: the latitude of the camera position.
- `lon`, lon: the longitude of the camera position.
- `heading_deg`, $\alpha_{\mathrm{heading}}$: the direction in which the camera is looking, also used by the spatial orientation. (0°: the camera faces "north", 90°: east, 180°: south, 270°: west)

7.2 Functions

`Camera.`**`setGPSpos`**(*lat*, *lon=None*, *elevation=None*)

Provide the earth position for the camera.

Parameters

- **`lat`** (*`number, string`*) – the latitude of the camera or the string representing the gps position.
- **`lon`** (*`number, optional`*) – the longitude of the camera.
- **`elevation`** (*`number, optional`*) – the elevation of the camera.

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera()
```

Supply the gps position of the camera as floats:

```
>>> cam.setGPSpos(-66.66, 140.00, 19)
```

or as a string:

```
>>> cam.setGPSpos("66°39'53.4"S 140°00'34.8"")
```

Camera.**gpsFromSpace**(*points*)

Convert points (Nx3) from the **space** coordinate system to the **gps** coordinate system.

Parameters **points** (*ndarray*) – the points in **space** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **gps** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Camera.**spaceFromGPS**(*points*)

Convert points (Nx3) from the **gps** coordinate system to the **space** coordinate system.

Parameters **points** (*ndarray*) – the points in **gps** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **space** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Camera.**gpsFromImage**(*points*, *X=None*, *Y=None*, *Z=0*, *D=None*)

Convert points (Nx2) from the **image** coordinate system to the **gps** coordinate system.

Parameters **points** (*ndarray*) – the points in **image** coordinates to transform, dimensions (2), (Nx2)

Returns **points** – the points in the **gps** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Camera.**imageFromGPS**(*points*)

Convert points (Nx3) from the **gps** coordinate system to the **image** coordinate system.

Parameters **points** (*ndarray*) – the points in **gps** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **image** coordinate system, dimensions (2), (Nx2)

Return type ndarray

7.3 Transformation

7.3.1 Distance

CameraTransform.**getDistance**(*point1*, *point2*)

Calculate the great circle distance between two points $(\mathrm{lat}_1, \mathrm{lon}_1)$ and $(\mathrm{lat}_2, \mathrm{lon}_2)$ on the earth (specified in

decimal degrees)

$$\begin{aligned}\Delta\text{lon} &= \text{lon}_2 - \text{lon}_1\\ \Delta\text{lat} &= \text{lat}_2 - \text{lat}_1\\ a &= \sin(\Delta\text{lat}/2)^2 + \cos(\text{lat}_1)\cdot\cos(\text{lat}_2)\cdot\sin(\Delta\text{lat}/2)^2\\ d &= 6371\,\text{km}\cdot 2\arccos(\sqrt{a})\end{aligned}$$

Parameters

- **point1** (*ndarray*) – the start point from which to calculate the distance, dimensions (2), (3), (Nx2), (Nx3)
- **point2** (*ndarray*) – the end point to which to calculate the distance, dimensions (2), (3), (Nx2), (Nx3)

Returns **distance** – the distance in m, dimensions (), (N)

Return type float, ndarray

Examples

```
>>> import CameraTransform as ct
```

Calculate the distance in m between two gps positions:

```
>>> ct.getDistance([52.51666667, 13.4], [48.13583333, 11.57988889])
503926.75849507266
```

or between a list of gps positions:

```
>>> ct.getDistance([[52.51666667, 13.4], [52.51666667, 13.4]], [[49.597854, 11.
↪005092], [48.13583333, 11.57988889]])
array([365127.04999716, 503926.75849507])
```

7.3.2 Bearing

CameraTransform.**getBearing**(*point1*, *point2*)

The angle relative β to the north direction from point $(\text{lat}_1, \text{lon}_1)$ to point $(\text{lat}_2, \text{lon}_2)$:

$$\begin{aligned}\Delta\text{lon} &= \text{lon}_2 - \text{lon}_1\\ X &= \cos(\text{lat}_2)\cdot\sin(\Delta\text{lon})\\ Y &= \cos(\text{lat}_1)\cdot\sin(\text{lat}_2) - \sin(\text{lat}_1)\cdot\cos(\text{lat}_2)\cdot\cos(\Delta\text{lon})\\ \beta &= \arctan 2(X, Y)\end{aligned}$$

Parameters

- **point1** (*ndarray*) – the first point from which to calculate the bearing, dimensions (2), (3), (Nx2), (Nx3)
- **point2** (*ndarray*) – the second point to which to calculate the bearing, dimensions (2), (3), (Nx2), (Nx3)

Returns **bearing** – the bearing angle in degree, dimensions (), (N)

Return type float, ndarray

Examples

```
>>> import CameraTransform as ct
```

Calculate the bearing in degrees between two gps positions:

```
>>> ct.getBearing([85.3205556, 4.52777778], [-66.66559128, 140.02233212])
53.34214977328738
```

or between a list of gps positions:

```
>>> ct.getBearing([[85.3205556, 4.52777778], [65.3205556, 7.52777778]], [[-66.
↪66559128, 140.02233212], [-60.66559128, 80.02233212]])
array([ 53.34214977, 136.82109976])
```

7.3.3 Move Distance

`CameraTransform.`**`moveDistance`**(*start*, *distance*, *bearing*)

Moving from $(\mathrm{lat}_1, \mathrm{lon}_1)$ a distance of d in the direction of β:

$$
\begin{aligned}
R &= 6371\,\mathrm{km} \\
\mathrm{lat}_2 &= \arcsin(\sin(\mathrm{lat}_1)\cdot\cos(d/R) + \cos(\mathrm{lat}_1)\cdot\sin(d/R)\cdot\cos(\beta)) \\
\mathrm{lon}_2 &= \mathrm{lon}_1 + \arctan\left(\frac{\sin(\beta)\cdot\sin(d/R)\cdot\cos(\mathrm{lat}_1)}{\cos(d/R) - \sin(\mathrm{lat}_1)\cdot\sin(\mathrm{lat}_2)}\right)
\end{aligned}
$$

Parameters

- **`start`** (*ndarray*) – the start point from which to calculate the distance, dimensions (2), (3), (Nx2), (Nx3)
- **`distance`** (*float, ndarray*) – the distance to move in m, dimensions (), (N)
- **`bearing`** (*float, ndarray*) – the bearing angle in degrees, specifying in which direction to move, dimensions (), (N)

Returns **target** – the target point, dimensions (2), (3), (Nx2), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
```

Move from 52.51666667°N 13.4°E, 503.926 km in the direction -164°:

```
>>> ct.moveDistance([52.51666667, 13.4], 503926, -164)
array([48.14444416, 11.52952357])
```

Batch process multiple positions at once:

```
>>> ct.moveDistance([[52.51666667, 13.4], [49.597854, 11.005092]], [10, 20], -164)
array([[52.51658022, 13.39995926],
       [49.5976811 , 11.00501551]])
```

Or one positions in multiple ways:

```
>>> ct.moveDistance([52.51666667, 13.4], [503926, 103926], [-164, -140])
array([[48.14444416, 11.52952357],
       [51.79667095, 12.42859387]])
```

7.3.4 GPS - String Conversion

CameraTransform.**formatGPS**(*lat*, *lon*, *format=None*, *asLatex=False*)

Formats a latitude, longitude pair in degrees according to the format string. The format string can contain a %s, to denote the letter symbol (N, S, W, E) and up to three number formaters (%d or %f), to denote the degrees, minutes and seconds. To not lose precision, the last one can be float number.

common formats are e.g.:

format	output	
%2d° %2d' %6.3f" %s (default)	70° 37' 4.980" S	8° 9' 26.280" W
%2d° %2.3f' %s	70° 37.083' S	8° 9.438' W
%2.3f°	-70.618050°	-8.157300°

Parameters

- **lat** (*number*) – the latitude in degrees
- **lon** (*number*) – the longitude in degrees
- **format** (*string*) – the format string
- **asLatex** (*bool*) – whether to encode the degree symbol

Returns

- **lat** (*string*) – the formatted latitude
- **lon** (*string*) – the formatted longitude

Examples

```
>>> import CameraTransform as ct
```

Convert a coordinate pair to a formatted string:

```
>>> lat, lon = ct.formatGPS(-70.61805, -8.1573)
>>> lat
'70° 37\'  4.980" S'
>>> lon
' 8°  9\' 26.280" W'
```

or use a custom format:

```
>>> lat, lon = ct.formatGPS(-70.61805, -8.1573, format="%2d° %2.3f %s")
>>> lat
'70° 37.083 S'
>>> lon
' 8° 9.438 W'
```

CameraTransform.**gpsFromString**(*gps_string*, *height=None*)

Read a gps coordinate from a text string in different formats, e.g. *70° 37' 4.980" S 8° 9' 26.280" W*, *70° 37.083 S 8° 9.438 W*, or *-70.618050° -8.157300°*.

Parameters

- **gps_string** (*str, list*) – the string of the point, containing both latitude and longitude, or a tuple with two strings one for latitude and one for longitude To batch process multiple strings, a list of strings can also be provided.
- **height** (*float, optional*) – the height of the gps point.

Returns **point** – a list containing, lat, lon, (height) of the given point.

Return type list

Examples

```
>>> import CameraTransform as ct
```

Convert a coordinate string to a tuple:

```
>>> ct.gpsFromString("85°1914N, 000°0243E")
array([8.53205556e+01, 4.52777778e-02])
```

Add a height information:

```
>>> ct.gpsFromString("66° 39´56.12862´´S  140°01´20.39562´´ E", 13.769)
array([-66.66559128, 140.02233212,  13.769     ])
```

Use a tuple:

```
>>> ct.gpsFromString(["66°39'56.12862''S", "140°01'20.39562'' E"])
array([-66.66559128, 140.02233212])
```

Or supply multiple coordinates with height information:

```
>>> ct.gpsFromString([["-66.66559128° 140.02233212°", 13.769], ["66°39'58.73922'
↪'S  140°01'09.55709'' E", 13.769]])
array([[-66.66559128, 140.02233212,  13.769     ],
       [-66.66631645, 140.01932141,  13.769     ]])
```

CHAPTER 8

Stereo Images

CameraTransform comes with a basic functionality to use stereo rigs. To calibrate a stereo rig, point correspondences in both images are needed. Stereo rigs should have two cameras that have different positions and orientations in space.

We start with an example of two camera images:

[1]:
```
import matplotlib.pyplot as plt

imA = plt.imread("StereoLeft.jpg")
imB = plt.imread("StereoRight.jpg")
```

.. figure:: StereoLeft.jpg :alt: The left stereo image

The left image of the stereo rig.

.. figure:: StereoRight.jpg :alt: The right stereo image

The right image of the stereo rig.

8.1 Setting up

We assume that the internal parameters of the camera (lens distortion or projection) have already been obtained. We now start with creating two cameras that share the same projection and create a CameraGroup object of both.

[2]:
```
import cameratransform as ct

cam1 = ct.Camera(ct.RectilinearProjection(focallength_px=3863.64, image=[4608, 2592]))
cam2 = ct.Camera(cam1.projection)

cam_group = ct.CameraGroup(cam1.projection, (cam1.orientation, cam2.orientation))
```

As a stereo setup cannot determine the length of objects in the scene if no length is provided, we have to define a baseline (distance between the two cameras). If the baseline is not known, it can be set to 1 and later the baseline can be scaled so that an object in the scene with known length has the right dimenstions.

[3]:
```
baseline = 0.87
```

Now we define an initial guess of the parameters:

[4]:
```
cam1.tilt_deg = 90
cam1.heading_deg = 35
cam1.roll_deg = 0
cam1.pos_x_m = 0
cam1.pos_y_m = 0
cam1.elevation_m = 0

cam2.tilt_deg = 90
cam2.heading_deg = -18
cam2.roll_deg = 0
cam2.pos_x_m = baseline
cam2.pos_y_m = 0
cam2.elevation_m = 0
```

8.2 Fitting

To fit the relative orientations of the images, we need point correspondences. This is then added as information to the camera group.

[5]:
```
import numpy as np
corresponding1 = np.array([[1035.19720133, 1752.47539918], [1191.16681219, 1674.
↪08229634], [1342.64515152, 1595.28089609], [1487.18243488, 1525.87033629], [1623.
↪55377003, 1456.86807389], [1753.39234661, 1391.94878561], [1881.18943613, 1331.
↪92906624], [2002.45376709, 1275.17572617], [2119.63512393, 1220.46387315], [2229.
↪46712739, 1167.79350718], [1132.21312835, 1106.84930585], [ 902.38609626, 1016.
↪70756572], [1594.61961207,  628.61640974], [1794.85813404,  704.30794726], [1791.
↪41760961,  663.70975895], [1956.56278237,  634.80935372], [2121.70795513,  600.
↪40410939], [2288.22933766,  571.50370416], [2439.61241269,  544.66761359], [2583.
↪42633397,  517.14341813], [2736.18561877,  488.93111778], [2876.55901561,  462.
↪7831321 ], [3010.05136359,  437.3232513 ], [3144.23181646,  413.92768515], [3270.
↪84311557,  393.28453856], [3393.32578537,  369.88897242], [2688.73357512, 1033.
↪7464922 ], [2885.97789523, 1144.56386559], [3068.37586866, 1053.89510554], [2870.
↪60132189,  952.09158548], [3237.51817542,  641.90898531], [3288.95016212,  879.
↪98075878], [4105.49922925,  943.60795881], [3965.51938917,  692.28051867], [4037.
↪41700677, 1135.08413333], [3253.6366414 , 1071.87603935], [3769.95210584, 1204.
↪30954628], [4207.84687809, 1265.15387253], [3808.67122255, 1451.37438621], [3587.
↪10046207, 1264.66938952], [3639.1604923 , 1353.23128002]])
corresponding2 = np.array([[ 352.50914077,  801.67993473], [ 460.08683446,  810.
↪68133359], [ 570.95772285,  820.12182507], [ 682.48725018,  829.56231656], [ 797.
↪35908021,  838.80579923], [ 908.18086062,  848.66284585], [1020.80811952,  858.
↪76197628], [1135.19174893,  870.61747721], [1249.57537835,  882.03388552], [1365.
↪71537827,  893.45029383], [ 911.70331034,  537.73870329], [ 956.46082091,  454.
↪7508191 ], [1718.73717286,  432.37206381], [1706.61534708,  518.62351648], [2062.
↪64027012,  573.54205381], [2203.84185338,  600.78475058], [2355.65939432,  631.
↪55257909], [2518.58129093,  665.72644033], [2673.15840747,  699.10556061], [2837.
↪669786  ,  733.27942185], [3018.47335419,  768.64539453], [3202.45588621,  807.
↪58770152], [3386.83578872,  844.9405266 ], [3584.72628753,  889.04864983], [3786.
↪59049113,  932.3620321 ], [3991.63365858,  976.47015533], [2525.23601839, 1218.
↪44516641], [2474.07408151, 1366.28811638], [2773.14569785, 1406.5405226 ], [2806.
↪62667125, 1256.44042836], [3392.99303531, 1137.69520116], [2908.10263735, 1298.
↪38150822], [3615.97183145, 1832.82274977], [4128.46912797, 1593.56296216], [3558.
↪36872853, 2064.51618063], [2885.45082314, 1509.32652806], [3106.67717245, 1803.
↪49557987], [3439.6617577 , 2276.44773767], [2628.22670423, 2051.78724561], [2789.
↪70873377, 1806.36653735], [2665.39954766, 1885.90842814]])
```

(continues on next page)

(continued from previous page)

```
cam_group.addPointCorrespondenceInformation(corresponding1, corresponding2)
```

We can now run a metropolis sampling to obtain the parameters for the fit. The setup has 5 degrees of freedom (in total 6, but we have fixed one already with the baseline). We keep the tilt of the first camera constant, as this tilt and the direction of the baseline define the orientation of the coordinate system in space.

[6]:
```
trace = cam_group.metropolis([
        ct.FitParameter("C0_heading_deg", lower=0, upper=45, value=15),
        ct.FitParameter("C0_roll_deg", lower=-5, upper=5, value=0),
        ct.FitParameter("C1_tilt_deg", lower=45, upper=135, value=85),
        ct.FitParameter("C1_heading_deg", lower=-45, upper=0, value=-15),
        ct.FitParameter("C1_roll_deg", lower=-5, upper=5, value=0)
    ], iterations=1e6)
```

```
100%|| 1000000/1000000 [36:55<00:00, 451.28it/s, acc_rate=0.37, factor=0.00363]
```

The result can be saved to a csv file:

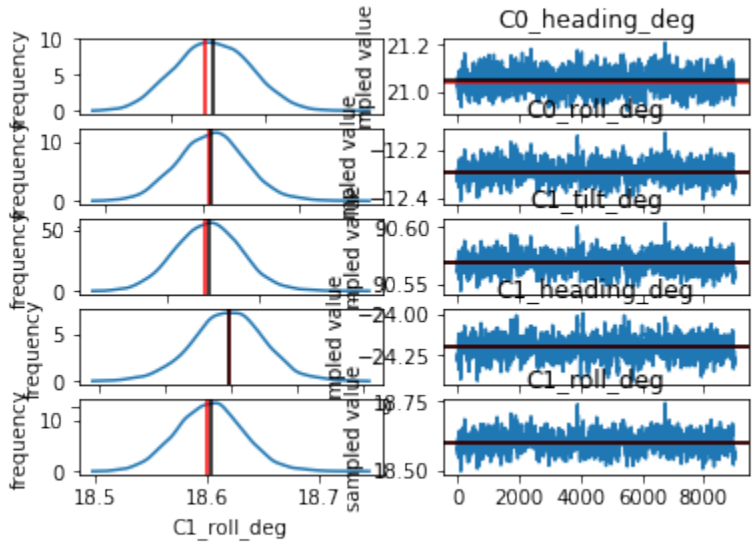
[7]:
```
trace[::100].to_csv("trace_stereo.csv", index=False)
```

And loaded again:

[8]:
```
import pandas as pd
trace = pd.read_csv("trace_stereo.csv")
cam_group.set_trace(trace)
```

And the traces can be ploted:

[9]:
```
cam_group.plotTrace()
```



8.3 Measuring

This section will use point correspondences in a stereo rig to measure distances. We define the start points and the end points of the distances we want to measure in both images:

[10]:
```
points1_start = np.array([[4186.75025841, 1275.33146571], [2331.92641872, 1019.
↪27786453], [2001.20914719,  644.51861935], [1133.13081198, 1109.00803712], [ 900.
↪20742144, 1016.83613865], [3287.47982902,  882.47189246], [3965.52607095,  693.
↪78282123], [2689.04269923, 1033.94065467], [2889.82546079, 1144.44899631]])
points1_end = np.array([[3833.11271185, 1441.03505624], [2234.96105241, 1095.
↪94815417], [2141.25664041,  614.81157533], [1797.43541128,  705.61811479], [1594.
↪1321895 ,  630.02825115], [4107.48373669,  942.87002974], [3238.54411195,  640.
↪438481  ], [2868.55390207,  951.44851233], [3070.37430064, 1053.65575788]])

points2_start = np.array([[3390.15132692, 2262.6779465 ], [2312.57037143, 1067.
↪65991742], [2112.03817032,  583.67614881], [ 910.64487701,  538.92782709], [ 955.
↪7855831 ,  454.96068822], [2907.87901391, 1297.11142789], [4125.81321176, 1596.
↪76190514], [2524.07911816, 1219.84947435], [2472.59638039, 1365.81424465]])
points2_end = np.array([[2674.06029559, 2066.93032418], [2136.89506898, 1071.
↪98893465], [2200.59552941,  598.28354825], [1706.9610645 ,  517.94260225], [1718.
↪87393798,  432.45627336], [3612.29928906, 1832.37417555], [3393.60277139, 1137.
↪62004484], [2807.91089842, 1255.11024902], [2773.34835691, 1405.2303788 ]])
```

This allows to calculate the 3D points of the start and end points and measure the distance between them:

[11]:
```
points_start_3D = cam_group.spaceFromImages(points1_start, points2_start)
points_end_3D = cam_group.spaceFromImages(points1_end, points2_end)

distances = np.linalg.norm(points_end_3D-points_start_3D, axis=-1)
print(distances)
```

```
[0.13885105 0.04962405 0.03691601 0.21049065 0.21220163 0.21331054
 0.21138547 0.07524718 0.0751257 ]
```

If we do not know the baseline of the camera rig, we can now use one of those distances to scale the setup:

[12]:
```
# the first distance is known to be 14cm
scale = 0.14/distances[0]
# we use this ratio to scale the space
cam_group.scaleSpace(scale)
```

CHAPTER 9

API

9.1 CameraTransform

class CameraTransform.**Camera**(*projection*, *orientation=None*, *lens=None*)

This class is the core of the CameraTransform package and represents a camera. Each camera has a projection (subclass of *CameraProjection*), a spatial orientation (*SpatialOrientation*) and optionally a lens distortion (subclass of LensDistortion).

addHorizonInformation(*horizon*, *uncertainty=1*, *only_plot=False*, *plot_color=None*)

Add a term to the camera probability used for fitting. This term includes the probability to observe the horizon at the given pixel positions.

Parameters

- **horizon** (*ndarray*) – the pixel positions of points on the horizon in the image, dimension (2) or (Nx2)
- **uncertainty** (*number, ndarray*) – the pixels offset, how clear the horizon is visible in the image, dimensions () or (N)
- **only_plot** (*bool, optional*) – when true, the information will be ignored for fitting and only be used to plot.

addLandmarkInformation(*lm_points_image*, *lm_points_space*, *uncertainties*, *only_plot=False*, *plot_color=None*)

Add a term to the camera probability used for fitting. This term includes the probability to observe the given landmarks and the specified positions in the image.

Parameters

- **lm_points_image** (*ndarray*) – the pixel positions of the landmarks in the image, dimension (2) or (Nx2)
- **lm_points_space** (*ndarray*) – the **space** positions of the landmarks, dimension (3) or (Nx3)

- **uncertainties** (*number, ndarray*) – the standard deviation uncertainty of the positions in the **space** coordinates. Typically for landmarks obtained by gps, it could be e.g. [3, 3, 5], dimensions scalar, (3) or (Nx3)
- **only_plot** (*bool, optional*) – when true, the information will be ignored for fitting and only be used to plot.

addObjectHeightInformation(*points_feet*, *points_head*, *height*, *variation*, *only_plot=False*, *plot_color=None*)

Add a term to the camera probability used for fitting. This term includes the probability to observe the objects with the given feet and head positions and a known height and height variation.

Parameters

- **points_feet** (*ndarray*) – the position of the objects feet, dimension (2) or (Nx2)
- **points_head** (*ndarray*) – the position of the objects head, dimension (2) or (Nx2)
- **height** (*number, ndarray*) – the mean height of the objects, dimensions scalar or (N)
- **variation** (*number, ndarray*) – the standard deviation of the heights of the objects, dimensions scalar or (N). If the variation is not known a pymc2 stochastic variable object can be used.
- **only_plot** (*bool, optional*) – when true, the information will be ignored for fitting and only be used to plot.

distanceToHorizon()

Calculates the distance of the camera's position to the horizon of the earth. The horizon depends on the radius of the earth and the elevation of the camera.

Returns **distance** – the distance to the horizon.

Return type number

generateLUT(*undef_value=0*)

Generate LUT to calculate area covered by one pixel in the image dependent on y position in the image

Parameters **undef_value** (*number, optional*) – what values undefined positions should have, default=0

Returns **LUT** – same length as image height

Return type ndarray

getCameraCone(*project_to_ground=False*, *D=1*)

The cone of the camera's field of view. This includes the border of the image and lines to the origin of the camera.

Returns **cone** – the cone of the camera in **space** coordinates, dimensions (Nx3)

Return type ndarray

getImageBorder(*resolution=1*)

Get the border of the image in a top view. Useful for drawing the field of view of the camera in a map.

Parameters **resolution** (*number, optional*) – the pixel distance between neighbouring points.

Returns **border** – the border of the image in **space** coordinates, dimensions (Nx3)

Return type ndarray

getImageHorizon (*pointsX=None*)
This function calculates the position of the horizon in the image sampled at the points x=0, x=im_width/2, x=im_width.

Parameters **pointsX** (*ndarray, optional*) – the x positions of the horizon to determine, default is [0, image_width/2, image_width], dimensions () or (N)

Returns **horizon** – the points im camera image coordinates of the horizon, dimensions (2), or (Nx2).

Return type ndarray

getObjectHeight (*point_feet*, *point_heads*, *Z=0*)
Calculate the height of objects in the image, assuming the Z position of the objects is known, e.g. they are assumed to stand on the Z=0 plane.

Parameters

- **point_feet** (*ndarray*) – the positions of the feet, dimensions: (2) or (Nx2)
- **point_heads** (*ndarray*) – the positions of the heads, dimensions: (2) or (Nx2)
- **Z** (*number, ndarray, optional*) – the Z position of the objects, dimensions: scalar or (N), default 0

Returns **heights** – the height of the objects in meters, dimensions: () or (N)

Return type ndarray

getRay (*points*, *normed=False*)
As the transformation from the **image** coordinate system to the **space** coordinate system is not unique, **image** points can only be uniquely mapped to a ray in **space** coordinates.

Parameters **points** (*ndarray*) – the points in **image** coordinates for which to get the ray, dimensions (2), (Nx2)

Returns

- **offset** (*ndarray*) – the origin of the camera (= starting point of the rays) in **space** coordinates, dimensions (3)
- **rays** (*ndarray*) – the rays in the **space** coordinate system, dimensions (3), (Nx3)

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,
↪ 2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

get the ray of a point in the image:

```
>>> offset, ray = cam.getRay([1968, 2291])
>>> offset
[0.00 0.00 15.40]
>>> ray
[-0.09 0.97 -0.35]
```

or the rays of multiple points in the image:

```
>>> offset, ray, cam.getRay([[1968, 2291], [1650, 2189]])
>>> offset
[0.00 0.00 15.40]
>>> ray
[[-0.09 0.97 -0.35]
 [-0.18 0.98 -0.33]]
```

getTopViewOfImage(*image*, *extent=None*, *scaling=None*, *do_plot=False*, *alpha=None*, *Z=0.0*, *skip_size_check=False*)

Project an image to a top view projection. This will be done using a grid with the dimensions of the extent ([x_min, x_max, y_min, y_max]) in meters and the scaling, giving a resolution. For convenience, the image can be plotted directly. The projected grid is cached, so if the function is called a second time with the same parameters, the second call will be faster.

Parameters

- **image** (*ndarray*) – the image as a numpy array.
- **extent** (*list, optional*) – the extent of the resulting top view in meters: [x_min, x_max, y_min, y_max]. If no extent is given a suitable extent is guessed. If a horizon is visible in the image, the guessed extent will in most cases be too streched.
- **scaling** (*number, optional*) – the scaling factor, how many meters is the side length of each pixel in the top view. If no scaling factor is given, a good scaling factor is guessed, trying to get about the same number of pixels in the top view as in the original image.
- **do_plot** (*bool, optional*) – whether to directly plot the resulting image in a matplotlib figure.
- **alpha** (*number, optional*) – an alpha value used when plotting the image. Useful if multiple images should be overlaid.
- **Z** (*number, optional*) – the "height" of the plane on which to project.
- **skip_size_check** (*bool, optional*) – if true, the size of the image is not checked to match the size of the cameras image.

Returns **image** – the top view projected image

Return type ndarray

gpsFromImage(*points*, *X=None*, *Y=None*, *Z=0*, *D=None*)

Convert points (Nx2) from the **image** coordinate system to the **gps** coordinate system.

Parameters **points** (*ndarray*) – the points in **image** coordinates to transform, dimensions (2), (Nx2)

Returns **points** – the points in the **gps** coordinate system, dimensions (3), (Nx3)

Return type ndarray

gpsFromSpace(*points*)

Convert points (Nx3) from the **space** coordinate system to the **gps** coordinate system.

Parameters **points** (*ndarray*) – the points in **space** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **gps** coordinate system, dimensions (3), (Nx3)

Return type ndarray

imageFromGPS (*points*)
Convert points (Nx3) from the **gps** coordinate system to the **image** coordinate system.

Parameters **points** (*ndarray*) – the points in **gps** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **image** coordinate system, dimensions (2), (Nx2)

Return type ndarray

imageFromSpace (*points*, *hide_backpoints=True*)
Convert points (Nx3) from the **space** coordinate system to the **image** coordinate system.

Parameters **points** (*ndarray*) – the points in **space** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **image** coordinate system, dimensions (2), (Nx2)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,
↪ 2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

transform a single point from the space to the image:

```
>>> cam.imageFromSpace([-4.17, 45.32, 0.])
[1969.52 2209.73]
```

or multiple points in one go:

```
>>> cam.imageFromSpace([[-4.03, 43.96,  0.], [-8.57, 47.91, 0.]]))
[[1971.05 2246.95]
 [1652.73 2144.53]]
```

load (*filename*)
Load the camera parameters from a json file.

Parameters **filename** (*str*) – the filename of the file to load.

rotateSpace (*delta_heading*)
Rotates the whole camera setup, this will turn the heading and rotate the camera position (pos_x_m, pos_y_m) around the origin.

Parameters **delta_heading** (*number*) – the number of degrees to rotate the camera clockwise.

save (*filename*)
Saves the camera parameters to a json file.

Parameters **filename** (*str*) – the filename where to store the parameters.

setGPSpos (*lat*, *lon=None*, *elevation=None*)
Provide the earth position for the camera.

Parameters

- **lat** (*number, string*) – the latitude of the camera or the string representing the gps position.
- **lon** (*number, optional*) – the longitude of the camera.
- **elevation** (*number, optional*) – the elevation of the camera.

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera()
```

Supply the gps position of the camera as floats:

```
>>> cam.setGPSpos(-66.66, 140.00, 19)
```

or as a string:

```
>>> cam.setGPSpos("66°39'53.4"S 140°00'34.8"")
```

spaceFromGPS (*points*)

Convert points (Nx3) from the **gps** coordinate system to the **space** coordinate system.

Parameters **points** (*ndarray*) – the points in **gps** coordinates to transform, dimensions (3), (Nx3)

Returns **points** – the points in the **space** coordinate system, dimensions (3), (Nx3)

Return type ndarray

spaceFromImage (*points*, *X=None*, *Y=None*, *Z=0*, *D=None*, *mesh=None*)

Convert points (Nx2) from the **image** coordinate system to the **space** coordinate system. This is not a unique transformation, therefore an additional constraint has to be provided. The X, Y, or Z coordinate(s) of the target points can be provided or the distance D from the camera.

Parameters

- **points** (*ndarray*) – the points in **image** coordinates to transform, dimensions (2), (Nx2)
- **X** (*number, ndarray, optional*) – the X coordinate in **space** coordinates of the target points, dimensions scalar, (N)
- **Y** (*number, ndarray, optional*) – the Y coordinate in **space** coordinates of the target points, dimensions scalar, (N)
- **Z** (*number, ndarray, optional*) – the Z coordinate in **space** coordinates of the target points, dimensions scalar, (N), default 0
- **D** (*number, ndarray, optional*) – the distance in **space** coordinates of the target points from the camera, dimensions scalar, (N)
- **mesh** (*ndarray, optional*) – project the image coordinates onto the mesh in **space** coordinates. The mesh is a list of M triangles, consisting of three 3D points each. Dimensions, (3x3), (Mx3x3)

Returns **points** – the points in the **space** coordinate system, dimensions (3), (Nx3)

Return type ndarray

Examples

```
>>> import CameraTransform as ct
>>> cam = ct.Camera(ct.RectilinearProjection(focallength_px=3729, image=(4608,
↪ 2592)),
>>>                 ct.SpatialOrientation(elevation_m=15.4, tilt_deg=85))
```

transform a single point (impliying the condition Z=0):

```
>>> cam.spaceFromImage([1968 , 2291])
[-3.93 42.45 0.00]
```

transform multiple points:

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]])
[[-3.93 42.45 0.00]
 [-8.29 46.11 -0.00]]
```

points that cannot be projected on the image, because they are behind the camera (for the RectilinearProjection) are returned with nan entries:

```
>>> cam.imageFromSpace([-4.17, -10.1, 0.])
[nan nan]
```

specify a y coordinate as for the back projection.

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]], Y=45)
[[-4.17 45.00 -0.93]
 [-8.09 45.00 0.37]]
```

or different y coordinates for each point:

```
>>> cam.spaceFromImage([[1968 , 2291], [1650, 2189]], Y=[43, 45])
[[-3.98 43.00 -0.20]
 [-8.09 45.00 0.37]]
```

undistortImage(*image*, *extent=None*, *scaling=None*, *do_plot=False*, *alpha=None*, *skip_size_check=False*)

Applies the undistortion of the lens model to the image. The purpose of this function is mainly to check the sanity of a lens transformation. As CameraTransform includes the lens transformation in any calculations, it is not necessary to undistort images before using them.

Parameters

- **image** (*ndarray*) – the image to undistort.
- **extent** (*list, optional*) – the extent in pixels of the resulting image. This can be used to crop the resulting undistort image.
- **scaling** (*number, optional*) – the number of old pixels that are used to calculate a new pixel. A higher value results in a smaller target image.
- **do_plot** (*bool, optional*) – whether to plot the resulting image directly in a matplotlib plot.
- **alpha** (*number, optional*) – when plotting an alpha value can be specified, useful when comparing multiple images.
- **skip_size_check** (*bool, optional*) – if true, the size of the image is not checked to match the size of the cameras image.

Returns **image** – the undistorted image

Return type ndarray

CHAPTER 10

Note

If you encounter any bugs or unexpected behaviour, you are encouraged to report a bug in our Bitbucket bugtracker.

Index

R

S

U