MultiViewTracks

Release August 2019

Jul 13, 2020

Contents

mvt integrates the results from structure-from-motion (SfM) and video-based tracking.
mvt can be used to acquire highly-detailed animal trajectories for behavioral analyses.

CHAPTER 1

About

You can find can find our paper at movement ecology, where we used **mvt** in diverse aquatic environments.

CHAPTER 2

How to

Visit the GitHub repository for installation instructions.

For examples and reference of the python module, see the following pages.

2.1 Basic Usage Example

In this example, we show the basic usage on a calibration/ground-truth example, in which we tracked a calibration wand with four GoPro Hero7 cameras.

The cameras were aranged in a square with a side-length of 0.6 m. The 0.5 m calibration wand had two colored ends that we tracked throughout the synchronized footage, resulting in two four-view trajectories.

In this notebook, we the triangulation of these multiple-view trajectories using the structure-from-motion scene reconstruction output from COLMAP.

We demonstrate a RMSE of 1.07 cm for tracking the calibration wand in this example.

[1]:
```
import sys
sys.path.append('../..')

import MultiViewTracks as mvt
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

[2]:
```
scene = mvt.Scene(model_path='./data/sparse',
                  tracks_path='./data/tracks',
                  fisheye=False,
                  verbose=False)
scene.get_cameras()
```

First, interpolate the cameras to get complete camera paths, otherwise the tracks will be only triangulated at the reconstructed frames.

[3]:
```
scene.interpolate_cameras()
```

Then, we

- triangulate the trajectory points that are observed from multiple views (and calculate respective reprojection errors),
- project the trajectory points only observed from exactly one view, the depth for this projection is interpolated from the triangulated trajectories,
- and combine the triangulated and projected trajectories.

[4]:
```
scene.triangulate_multiview_tracks()
scene.get_reprojection_errors()
scene.project_singleview_tracks()
scene.get_tracks_3d()
```

We specify the measured real world distance and verify the selection of the correct camera ids.
Cameras 1 and 3 were on one side of the camera array, so their distance is 0.6m.

[5]:
```
camera_ids = [1, 3]
world_distance = 0.6

for camera_id in camera_ids:
    print(scene.cameras[camera_id])
```

```
Camera 1 | i_0_cut, 5821 views, with tracks
Camera 3 | m_0_cut, 5751 views, with tracks
```

Then, we scale the tracks and retrieve reconstruction errors.
The reconstruction error is the per-frame difference of the reconstructed camera positions and the known real world distance.

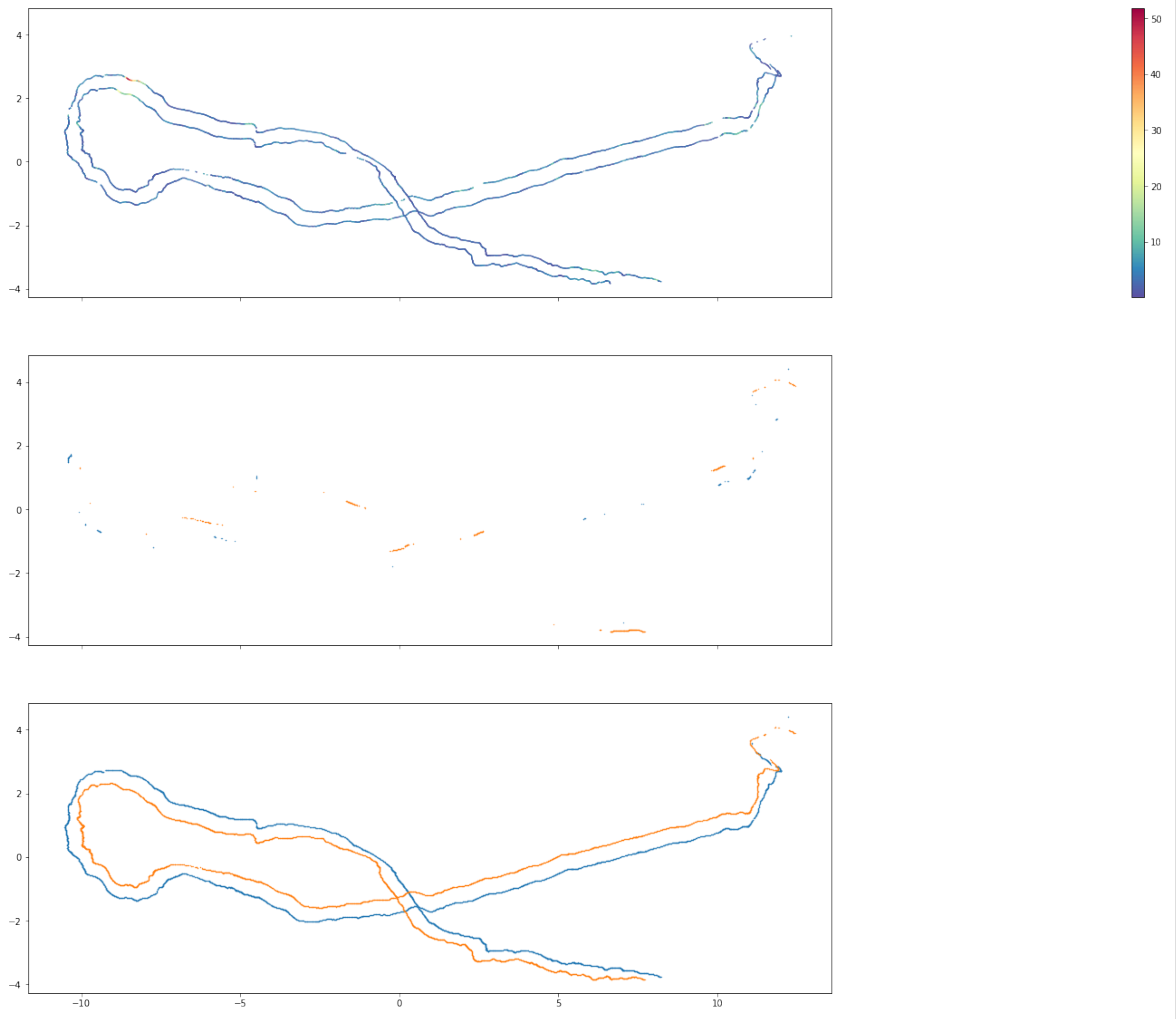
[6]:
```
reconstruction_errors = scene.scale(camera_ids, world_distance)
```

Finally, we rotate the tracks, so that x and y of the tracks match the first two principal components of the camera paths.
For reconstructions in a flat environment, this ensures that the z component of the tracks is actually the height (or depth if filmed top-down) component.

[7]:
```
scene.rotate()
```

Now we plot the multi-view triangulation (with reprojection errors), single-view projection and combined results for comparison.
Note that since we scaled with a known real world distance, the axis scales represent meters.

```
[8]: fig, axes = plt.subplots(3, 1, figsize=(20, 20), sharey=True, sharex=True)
axes[0] = mvt.utils.plot_tracks_2d(scene.tracks_triangulated, ax=axes[0], show=False,␣
↪size=0.1, style='errors')
axes[1] = mvt.utils.plot_tracks_2d(scene.tracks_projected, ax=axes[1], show=False,␣
↪size=0.1)
axes[2] = mvt.utils.plot_tracks_2d(scene.tracks_3d, ax=axes[2], show=True, size=0.1);
```



The triangulated multiple-view tracks were already quite complete in this example and mainly have low reprojection errors, but the single-view tracks add some additional trajectory points. We can calculate the reconstruction errors of the calibration wand and compare it to the reconstruction errors of the cameras retrieved above.

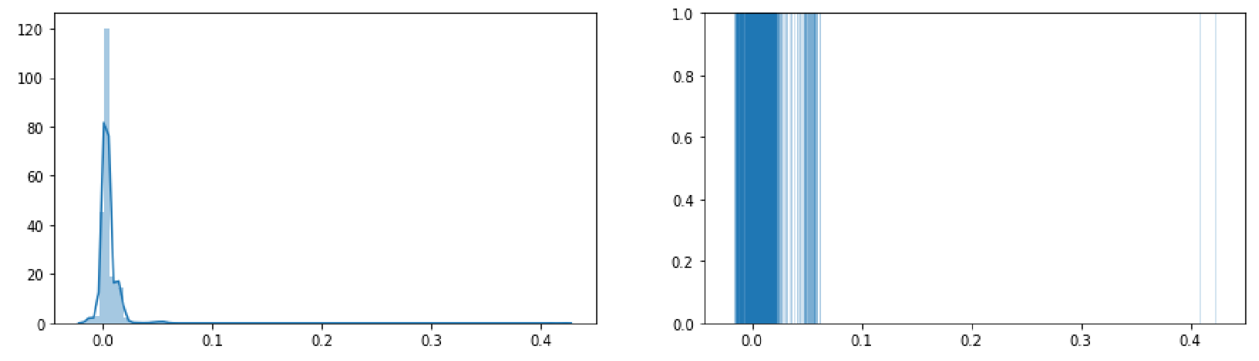
2.2 Calculating Reconstruction Errors

Now, we can calculate the calibration wand length and the corresponding reconstruction errors in each frame that has a 3d point for each of the ends.

```
[9]: import numpy as np

     # real world distance of the calibration wand ends
     world_distance = 0.5

     # generate masks of the shared frame indices for both individuals (calibration wand␣
     ↪ends)
     shared = []
     for i, j in zip(scene.tracks_3d['IDENTITIES'], scene.tracks_3d['IDENTITIES'][::-1]):
         shared.append(np.isin(scene.tracks_3d[str(i)]['FRAME_IDX'], scene.tracks_
     ↪3d[str(j)]['FRAME_IDX']))
     # retrieve the positions for each of the individuals, masked with the shared frames
     pts_3d = []
     for idx, i in enumerate(scene.tracks_3d['IDENTITIES']):
         pts_3d.append(np.transpose([scene.tracks_3d[str(i)]['X'][shared[idx]],
                                     scene.tracks_3d[str(i)]['Y'][shared[idx]],
                                     scene.tracks_3d[str(i)]['Z'][shared[idx]]]))
     # calculate the distances and the errors
     distances = np.sqrt(np.square(pts_3d[0] - pts_3d[1]).sum(axis=1))
     errors = distances - world_distance
```

First, let's plot the error distribution.

```
[10]: fig, axes = plt.subplots(1, 2, sharex=True, figsize=(15, 4))
      sns.distplot(errors, bins=100, ax=axes[0])
      sns.distplot(errors, bins=100, ax=axes[1], kde=False, hist=False, rug=True, rug_kws={
      ↪'height': 1, 'alpha': 0.2})
      axes[1].set_ylim((0, 1));
```



We can see that the majority of all errors are within a 5cm range around 0, which is quite accurate considering that the size of the reconstruction is > 20m.

However, we can also see that the distribution has a long tail on the right, with some errors forming a peak at 6 cm and with two errors > 40 cm.

For fine scale behavioral analyses, the larger errors should be generally avoided. So let's have a closer look at the reconstruction errors we retrieved above when scaling the scene. The scaling and rotating is only applied on the tracks, but not on the camera parameters, so we have to scale the camera paths separately using the same factor.

```
[11]: # specify the real world distance between the cameras
      world_distance = 0.6
```

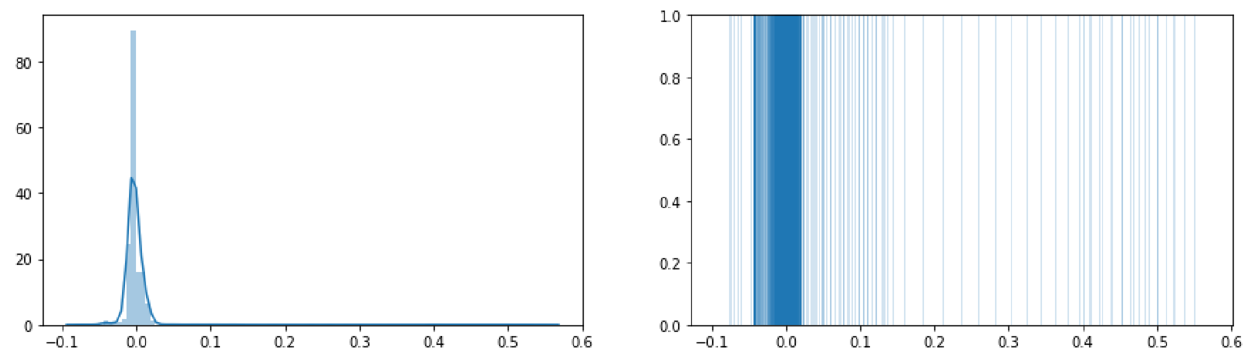
(continues on next page)

(continued from previous page)

```
# retrieve the two cameras which were used for scaling from the scene
cameras = [scene.cameras[camera_ids[0]], scene.cameras[camera_ids[1]]]
# generate masks for each camera view indices in which both cameras are reconstructed
reconstructed = [np.isin(cameras[0].view_idx, cameras[1].view_idx),
                 np.isin(cameras[1].view_idx, cameras[0].view_idx)]
# retrieve the camera center paths for both cameras with applied masks
pts_3d = [np.array([cameras[0].projection_center(idx) for idx in cameras[0].view_
↪idx])[reconstructed[0]],
          np.array([cameras[1].projection_center(idx) for idx in cameras[1].view_
↪idx])[reconstructed[1]]]
# calculate the scale factor and the reconstruction errors
distances = np.sqrt(np.square(pts_3d[0] - pts_3d[1]).sum(axis=1))
scale = world_distance / distances.mean()
distances = distances * scale
# now calculate the reconstruction errors.
# these are same as returned by scene.scale, but we needed the camera paths for later␣
↪visualization.
errors = distances - distances.mean()
```

Draw the same plot as for camera-to-camera distance error distribution.

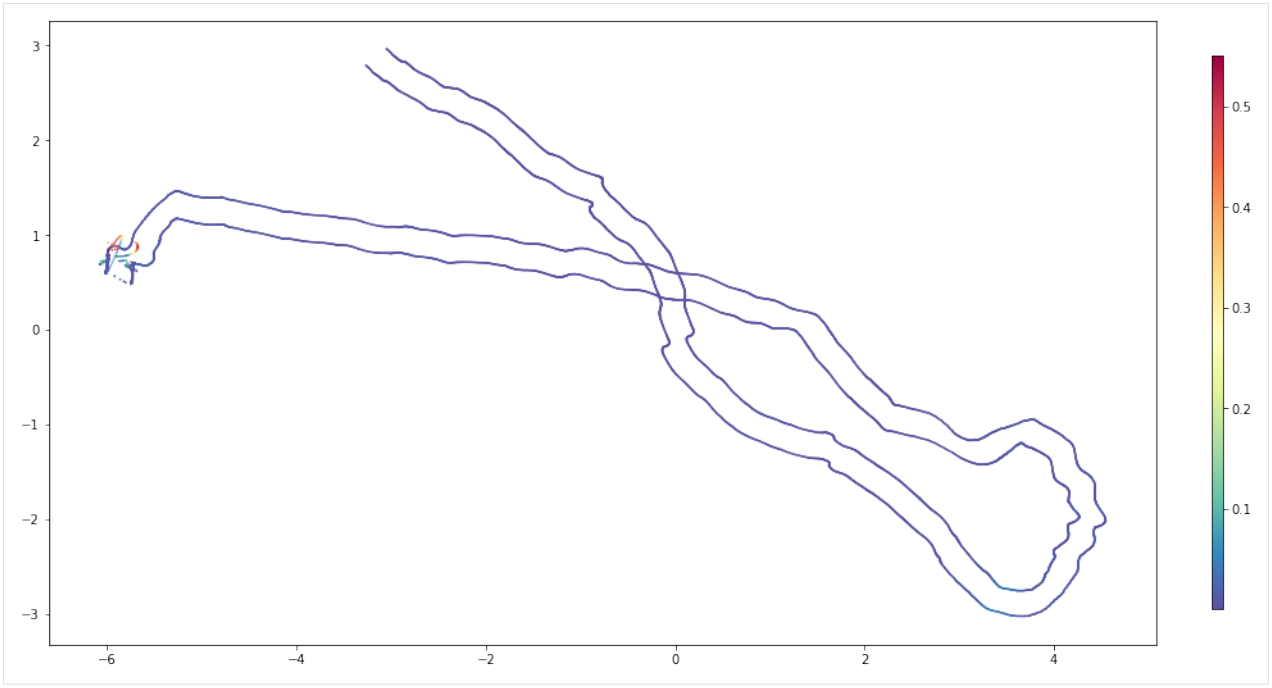
[12]:
```
fig, axes = plt.subplots(1, 2, sharex=True, figsize=(15, 4))
sns.distplot(errors, bins=100, ax=axes[0])
sns.distplot(errors, bins=100, ax=axes[1], kde=False, hist=False, rug=True, rug_kws={
↪'height': 1, 'alpha': 0.2})
axes[1].set_ylim((0, 1));
```



Again, we see that generally the camera-to-camera distances have a low error below 5 cm. The extended right tail of the error distribution is also visible here, with even more high errors. Let's plot the errors mapped onto the reconstructed camera paths.

Note that x and y are flipped for visualization (landscape plot), since the tracks rotation does not apply here. Although visually similar to the trajectory plots above, the plot below shows the camera paths of the two selected cameras and not the trajectories of the calibration wand ends.

[13]:
```
fig = plt.figure(figsize=(12, 10))
ax = fig.add_axes([0, 0, 1, 1])
for camera_path in pts_3d:
    mappable = ax.scatter(camera_path[:, 1], camera_path[:, 0], cmap=plt.get_cmap(
↪'Spectral_r'), c=np.absolute(errors), s=0.25)
ax.set_aspect('equal')
cax = fig.add_axes([1.05, 0.2, 0.01, 0.6])
fig.colorbar(mappable, cax=cax);
```



We can observe that the reconstrion errors are very low throughout the reconstruction, except at the end. So we have a look at the end of the videos, and can confirm that all cameras capture a diver, a lot of water and not so much of the static environment.

Basically, we forgot cut the synchronized videos to an appropriate length, and thus tried to reconstruct things that are not suitable for sfm reconstruction.

This is something we can easily fix by applying a conditional filter to the extrinsics before we initialize the cameras in the scene.

```
[14]: scene = mvt.Scene(model_path='./data/sparse',
                        tracks_path='./data/tracks',
                        fisheye=False,
                        verbose=False)
      scene.get_cameras()
```

Create and apply a filter that removes the last 15 seconds (or 450 frames) of the reconstruction. This is where the diver shows up and the trial is finished.

```
[15]: condition = scene.extrinsics['FRAME_IDX'] < scene.extrinsics['FRAME_IDX'].max() - 450
      scene.extrinsics = {key: scene.extrinsics[key][condition] for key in scene.extrinsics}
```

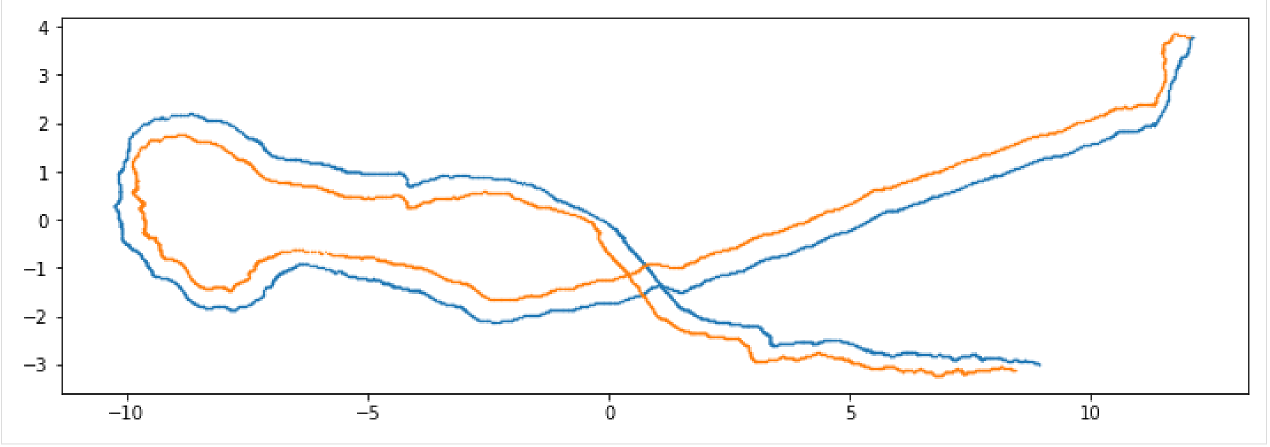
And continue as above.

```
[16]: scene.get_cameras()
      scene.interpolate_cameras()
      scene.triangulate_multiview_tracks()
      scene.get_reprojection_errors()
      scene.project_singleview_tracks()
      scene.get_tracks_3d()
      camera_ids = [1, 3]
      world_distance = 0.6
      reconstruction_errors = scene.scale(camera_ids, world_distance)
```

(continues on next page)

(continued from previous page)

```
scene.rotate()
mvt.utils.plot_tracks_2d(scene.tracks_3d, figsize=(12, 12), size=0.1, show=True);
```



We can see that in comparison to the tracks above, the weird-looking, final trajectory points are missing.

So let's plot the error distributions again and calculate root mean squared errors (RMSEs) for tracking the calibration wand and reconstructing the camera paths.

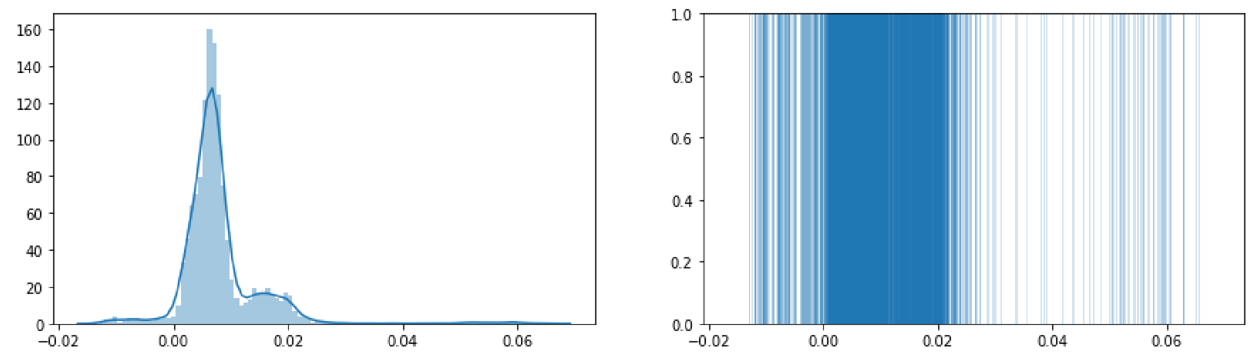
```
[17]: # real world distance of the calibration wand ends
world_distance = 0.5

# generate masks of the shared frame indices for both individuals (calibration wand␣
↪ends)
shared = []
for i, j in zip(scene.tracks_3d['IDENTITIES'], scene.tracks_3d['IDENTITIES'][::-1]):
    shared.append(np.isin(scene.tracks_3d[str(i)]['FRAME_IDX'], scene.tracks_
↪3d[str(j)]['FRAME_IDX']))
# retrieve the positions for each of the individuals, masked with the shared frames
pts_3d = []
for idx, i in enumerate(scene.tracks_3d['IDENTITIES']):
    pts_3d.append(np.transpose([scene.tracks_3d[str(i)]['X'][shared[idx]],
                                scene.tracks_3d[str(i)]['Y'][shared[idx]],
                                scene.tracks_3d[str(i)]['Z'][shared[idx]]]))
# calculate the distances and the errors
distances = np.sqrt(np.square(pts_3d[0] - pts_3d[1]).sum(axis=1))
errors_wand = distances - world_distance

# calculate the root mean squared error
rmse_wand = np.sqrt(np.square(errors_wand).mean())

print('Calibration wand length reconstruction errors')
fig, axes = plt.subplots(1, 2, sharex=True, figsize=(15, 4))
sns.distplot(errors_wand, bins=100, ax=axes[0])
sns.distplot(errors_wand, bins=100, ax=axes[1], kde=False, hist=False, rug=True, rug_
↪kws={'height': 1, 'alpha': 0.2})
axes[1].set_ylim((0, 1));
plt.show()

# specify the real world distance between the cameras
world_distance = 0.6

# retrieve the two cameras which were used for scaling from the scene
```

(continues on next page)

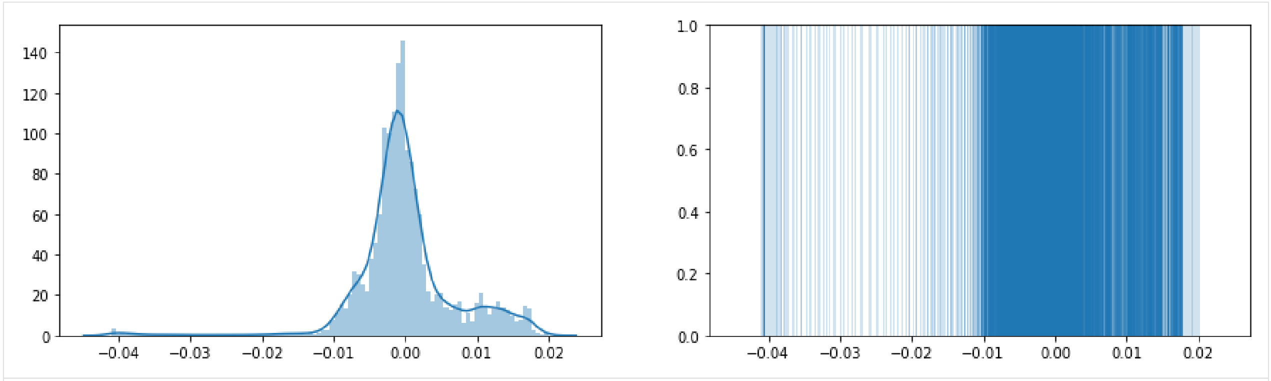
(continued from previous page)

```
cameras = [scene.cameras[camera_ids[0]], scene.cameras[camera_ids[1]]]
# generate masks for each camera view indices in which both cameras are reconstructed
reconstructed = [np.isin(cameras[0].view_idx, cameras[1].view_idx),
                 np.isin(cameras[1].view_idx, cameras[0].view_idx)]
# retrieve the camera center paths for both cameras with applied masks
pts_3d = [np.array([cameras[0].projection_center(idx) for idx in cameras[0].view_
↪idx])[reconstructed[0]],
          np.array([cameras[1].projection_center(idx) for idx in cameras[1].view_
↪idx])[reconstructed[1]]]
# calculate the scale factor and the reconstruction errors
distances = np.sqrt(np.square(pts_3d[0] - pts_3d[1]).sum(axis=1))
scale = world_distance / distances.mean()
distances = distances * scale
# now calculate the reconstruction errors.
# these are same as returned by scene.scale, but we needed the camera paths for later␣
↪visualization.
errors_cameras = distances - distances.mean()

# calculate the root mean squared error
rmse_cameras = np.sqrt(np.square(errors_cameras).mean())

print('Camera-to-camera reconstruction errors')
fig, axes = plt.subplots(1, 2, sharex=True, figsize=(15, 4))
sns.distplot(errors_cameras, bins=100, ax=axes[0])
sns.distplot(errors_cameras, bins=100, ax=axes[1], kde=False, hist=False, rug=True,␣
↪rug_kws={'height': 1, 'alpha': 0.2})
axes[1].set_ylim((0, 1));
plt.show()

print('Visualizing the errors on top of the camera paths')
fig = plt.figure(figsize=(12, 10))
ax = fig.add_axes([0, 0, 1, 1])
for camera_path in pts_3d:
    mappable = ax.scatter(camera_path[:, 1], camera_path[:, 0], cmap=plt.get_cmap(
↪'Spectral_r'), c=np.absolute(errors_cameras), s=0.25)
ax.set_aspect('equal')
cax = fig.add_axes([1.05, 0.2, 0.01, 0.6])
fig.colorbar(mappable, cax=cax);
```

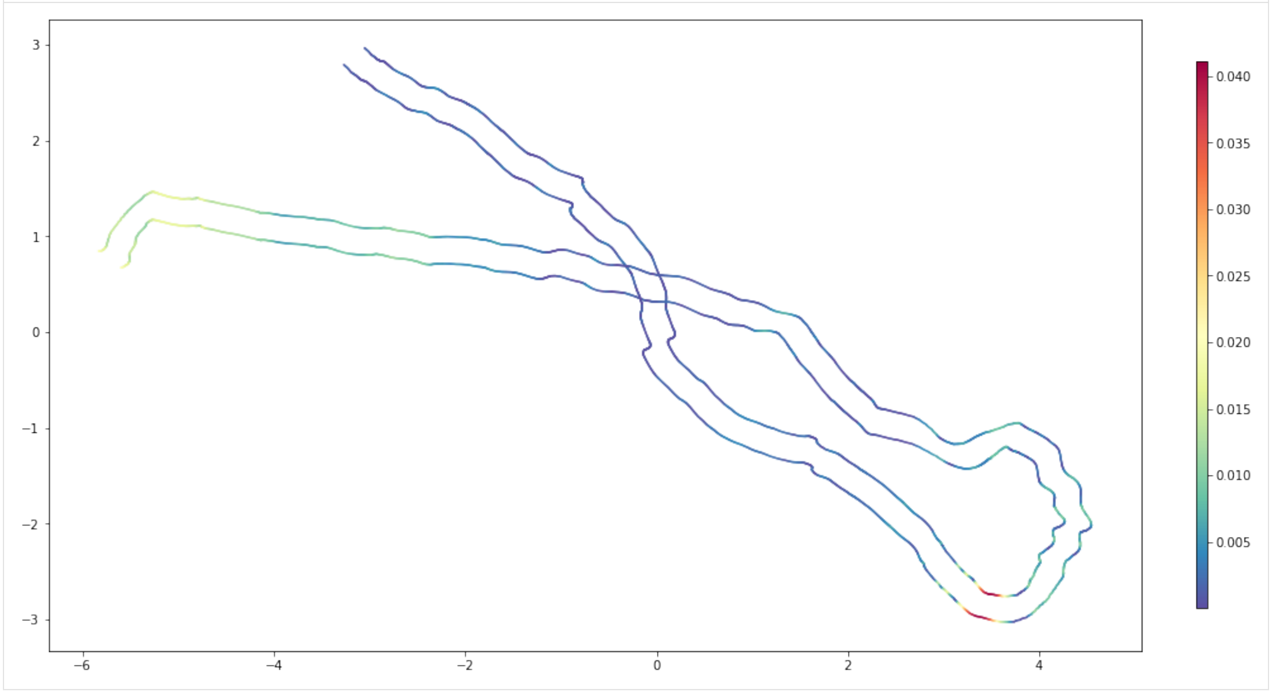
```
Calibration wand length reconstruction errors
```



```
Camera-to-camera reconstruction errors
```



```
Visualizing the errors on top of the camera paths
```



Note that again, x and y are flipped for visualization purposes and the plotted camera paths are not scaled to real-world scale.
Also note the different scale of the colormap.

The RMSEs calculated above:

```
[18]: rmse_wand, rmse_cameras
```

```
[18]: (0.010682141792378272, 0.007042060594613508)
```

Removing the last frames reduced the error ranges to 2 cm for the majority calibration wand legth errors (with a maximum error <7 cm) and to 3 cm for the majority of camera-to-camera distance reconstruction errors (with a maximum error <5 cm).

In conclusion, incorrect pre-processing of the structure-from-motion input (not cutting the videos correctly) was the main reason of tracking errors in this example. When corrected, we have achieved a RMSE of 1.07 cm for tracking the calibration wand throughout approx. 25 x 10 m of a rocky, marine underwater environment.

2.3 Visualization Example

Here, we follow the steps explained in *scene* to visualize the sparse or dense point clouds of the COLMAP reconstruction and generate a 3D visualization of the calibration wand tracks.

First, we initialize everything as already shown, but additionally, also load the COLMAP dense scene for plotting the dense point cloud.

[1]:
```
import sys
sys.path.append('../..')

import MultiViewTracks as mvt
import matplotlib.pyplot as plt

import os
from glob import glob
import numpy as np
```

[2]:
```
# copy dense scene to model path (otherwise sparse point cloud will be loaded)
!cp ./data/dense/fused_cleanded.ply ./data/sparse
```

[3]:
```
scene = mvt.Scene(model_path='/media/paul/Samsung_T5/revision/20190624-stick-0/sparse
↪',
                  tracks_path='/media/paul/Samsung_T5/revision/20190624-stick-0/tracks
↪',
                  fisheye=False,
                  verbose=False)
scene.get_pointcloud()
condition = scene.extrinsics['FRAME_IDX'] < scene.extrinsics['FRAME_IDX'].max() - 450
scene.extrinsics = {key: scene.extrinsics[key][condition] for key in scene.extrinsics}
scene.get_cameras()
scene.interpolate_cameras()
scene.triangulate_multiview_tracks()
scene.project_singleview_tracks()
scene.get_tracks_3d()

camera_ids = [1, 3]
world_distance = 0.6

reconstruction_errors = scene.scale(camera_ids, world_distance)
scene.rotate()
```

Linearly interploate the 3D tracks for visualization purposes, so that we ensure all frames have both ends of the calibration wand tracked.

[4]:
```
tracks_interpolated = mvt.tracks.interpolate_tracks(scene.tracks_3d)
```

Create lists of points for later visualization. Here we extract both ends of the wand every 60 frames, starting from frame 10.

[5]:
```
pts_2d = []
pts = []
for idx in tracks_interpolated['FRAME_IDX'][10::60]:
    if not np.all([np.isin(idx, tracks_interpolated[str(i)]['FRAME_IDX']) for i in␣
↪tracks_interpolated['IDENTITIES']]):
        continue
```

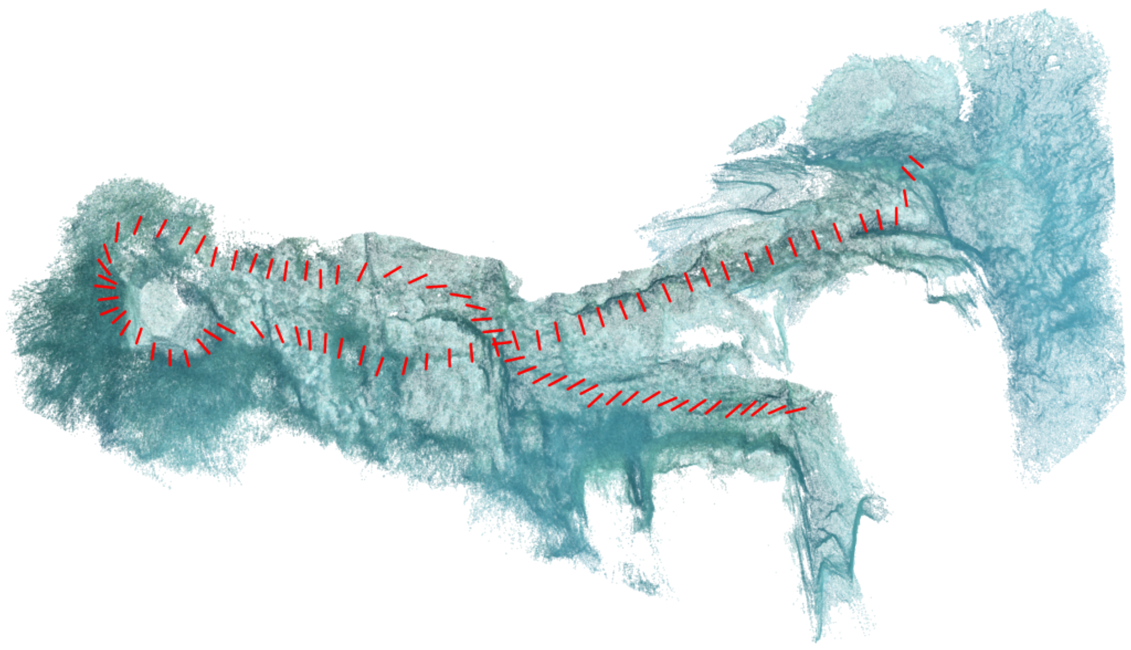
(continues on next page)

(continued from previous page)

```
    pts_3d = [np.transpose([tracks_interpolated[str(i)]['X'][tracks_
↪interpolated[str(i)]['FRAME_IDX'] == idx],
                            tracks_interpolated[str(i)]['Y'][tracks_
↪interpolated[str(i)]['FRAME_IDX'] == idx],
                            tracks_interpolated[str(i)]['Z'][tracks_
↪interpolated[str(i)]['FRAME_IDX'] == idx]]) \
              for i in tracks_interpolated['IDENTITIES']]
    pts_2d.append(np.array(pts_3d)[:, :, :2].reshape(2, 2))
    pts.append(np.array(pts_3d))
```

Let's plot it on top of the dense COLMAP reconstruction (the scatter plot takes some time).

```
[6]: fig, ax = plt.subplots(figsize=(20, 20))
ax.axes.axis('off')
ax.scatter(scene.point_cloud[:, 0], scene.point_cloud[:, 1], s=0.02, c=scene.point_
↪cloud[:, 3:] / 255)
lc = ax.plot(*np.transpose(pts_2d), c='r', lw=2.5, solid_capstyle='round')
ax.set_aspect('equal');
```



In order to visualize the wand in a 3D point cloud, we need to interpolate the 3D points between both ends of the wand for every frame we want to visualize.

```
[7]: pts = np.array(pts).reshape(-1, 2, 3)
x = pts[:, :, 0]
y = pts[:, :, 1]
z = pts[:, :, 2]
x_interpolated = []
y_interpolated = []
z_interpolated = []
for component, component_interpolated in zip([x, y, z], [x_interpolated, y_
↪interpolated, z_interpolated]):
    for idx in range(component.shape[0]):
```

(continues on next page)

(continued from previous page)

```
        component_interpolated.append(np.interp(np.linspace(0, 1, 100), np.array((0,␣
↪1)), component[idx, :]))
x_interpolated = np.array(x_interpolated)
y_interpolated = np.array(y_interpolated)
z_interpolated = np.array(z_interpolated)
```

Finally, we create a numpy array in the same format as the COLMAP reconstruction we loaded above, and write it to a .ply file. The same applies to the reconstruction, since we rotated and scaled the pointcloud.

[8]:
```
pts_interpolated = np.transpose([x_interpolated, y_interpolated, z_interpolated]).
↪reshape(-1, 3)
rgb = np.zeros_like(pts_interpolated)
rgb[:, 0] = 255
point_cloud = np.append(pts_interpolated, rgb, axis=1)
mvt.utils.write_ply(mvt.utils.pointcloud_to_ply(point_cloud),
                    file_name='./data/dense/visualization_calibration_wand.ply')
mvt.utils.write_ply(mvt.utils.pointcloud_to_ply(scene.point_cloud),
                    file_name='./data/dense/visualization_reconstruction.ply')
```

We use meshlab to load both the above created .ply files and flatten the mesh layers. The result of this example can be found on scetchfab.

2.4 Scene Class Reference

class MultiViewTracks.**Scene**(*model_path*, *tracks_path*, *fisheye*, *verbose=True*)

This is a class for triangulating tracks using the camera parameters of a COLMAP reconstruction.

model_path
Path to the COLMAP model .bin files

Type str

tracks_path
Path to the tracks .pkl files

Type str

fisheye
Did you use OPENCV_FISHEYE in COLMAP reconstruction?

Type bool

verbose
Do you want a bit of verbosity?

Type bool

extrinsics
Stores COLMAP extrinsic camera parameters

Type dict

intrinsics
Stores COLMAP intrinsic camera parameters

Type dict

cameras
Stores Camera class instance for each reconstructed camera

Type dict

`tracks`
Stores tracks for each camera

Type dict

`tracks_triangulated`
Stores the triangulated multiple-view tracks, otherwise None

Type dict

`tracks_projected`
Stores the projected single-view tracks, otherwise None

Type dict

`tracks_3d`
Stores the combined 3d tracks, otherwise None

Type dict

`point_cloud`
Stores the sparse or dense COLMAP point cloud otherwise None

Type np.ndarray

`get_cameras``()`
Creates Camera objects for each unique image prefix with COLMAP reconstruction parameters.

`get_extrinsics``()`
Read the COLMAP extxrinsic camera parameters.

See https://github.com/colmap/colmap/blob/dev/scripts/python/read_model.py for reference.

`get_intrinsics``()`
Read the COLMAP intrinsic camera parameters. Camera model should be OPENCV or OPENCV_FISHEYE.

See https://github.com/colmap/colmap/blob/dev/scripts/python/read_model.py for reference.

`get_pointcloud``()`
Tries to read a dense point cloud (.ply file) from the model path. Otherwise reads the sparse point cloud from the COLMAP reconstruction.

See https://github.com/colmap/colmap/blob/dev/scripts/python/read_model.py for reference.

`get_reprojection_errors``()`
Computes the minimum reprojection error for each triangulated 3D point.

Reprojection errors are stored in tracks_triangulated attribute.

`get_tracks``()`
Read the tracks .pkl files. The file names should match the camera name, but can have a prefix (i.e. prefix[camera_name].pkl)

`get_tracks_3d``()`
Combine triangulated multiple-view trajectories and projected single-view tracks.

`interpolate_cameras``()`
Interpolates the camera paths of the Scene using Camera.interpolate.

`project_singleview_tracks``()`
Project all trajectory points that are observed in only one view to an interpolated detph. Use this if the tracks are mostly planar and uncomplete.

project_tracks()
Projects the tracks of all cameras using Camera.project_tracks.

rotate(*camera_ids=[]*)
Rotates the tracks and 3d point cloud using PCA, so that the first two principal components of the camera paths are x and y.

If reprojection errors were computed for the triangulated trajectories, they remain stored in tracks_triangulated.

Parameters **camera_ids** (*list, optional*) – The ids of the cameras used to calculated the two main axes of view point positions. Defaults to all cameras

scale(*camera_ids*, *world_distance*)
Scales the tracks and 3d point cloud according to a known camera-to-camera distance.

If reprojection errors were computed for the triangulated trajectories, they remain stored in tracks_triangulated.

Parameters

- **camera_ids** (*(int, int)*) – The camera ids used to calculated the distance for scaling
- **world_distance** (*float*) – The known real-world distance between the two specified cameras

Returns The reconstruction errors calculated as the difference between reconstruted and measured distance

Return type np.ndarray

triangulate_multiview_tracks()
Triangulate all trajectory points that are observed in more than one view.

undistort_tracks()
Undistorts the tracks of all cameras using Camera.undistort_tracks.

2.5 Camera Class Reference

class MultiViewTracks.**Camera**(*id*, *name*, *fisheye*, *extrinsics*, *intrinsics*, *tracks*, *verbose=True*)
This is a class containing per camera preprocessing methods for triangulating tracks using the Scene class.

id
The camera id within a COLMAP reconstruction

Type int

name
A common prefix used for all images of this camera in a COLMAP reconstruction

Type str

fisheye
Was OPENCV_FISHEYE used as COLMAP camera model?

Type bool

image_names
Contains the image names of the reconstructed views of this camera

Type np.ndarray

`view_idx`
Contains the frame indices of all reconstructed views of this camera

Type np.ndarray

`n_views`
Number of reconstruecued views of this camera

Type int

`R`
A Rotation instance holding extrinsic parameters (rotations) for this camera

Type scipy.spatial.transform.Rotation

`r`
Stores the rotation matrices retrieved from R

Type np.ndarray

`t`
Stores COLMAP extrinsic camera parameters for each view of this camera

Type np.ndarray

`k`
The camera matrix of this camera

Type np.ndarray

`d`
The distortion parameters of this camera

Type np.ndarray

`tracks`
Contains the tracks visible from this camera or None

Type dict

`tracks_undistorted`
Contains undistorted tracks from Camera.undistort_tracks or None

Type dict

`tracks_projected`
Contains transformed tracks from Camera.project_tracks or None

Type dict

`tracks_reprojected`
Contains reprojected tracks from Camera.reproject_tracks or None

Type dict

`verbose`
Do you want some verbosity? Defaults to true

Type bool

`frames_in_view`(*i*)
Returns the frame indices in which individual i is observed in the camera views.

`get_rotations`()
Returns a np.ndarray of rotation matrices transformed from R.

`interpolate`()
Interpolates the camera path by linearly interpolating t and using SLERP for R.

`position`(*idx*, *i*, *kind=”*)
Returns the position of individual i at the specified frame index.

Parameters

- **`idx`** (*`int`*) – The frame index
- **`i`** (*`int`*) – The individual’s identity
- **`kind`** (*`str, optional`*) – One of “”, “undistorted”, “projected”. Defaults to “”

Returns The position of individual i at frame index idx

Return type np.ndarray

`project_tracks`()
Projects the tracks to world coordinates with unknown depth using r and t, Camera.undistort_tracks first if necessary.

`projection_center`(*idx*)
Return the 3d coordinates of the idx-th view projection center.

`reproject_tracks`(*tracks_3d*)
Projects given tracks from world coordinates to image coordinates using r and t, and distortion model. The tracks must be in the original coordinate system, i.e. they should not be rotated or scaled.

Parameters **`tracks_3d`** (*`dict`*) – The 3D tracks in the world coordiate system to be reprojected.

`undistort_tracks`()
Undistorts the tracks in image coordinates to normalized coordinates.

`view`(*idx*)
Returns the projection matrix of the camera at the specified frame index.

2.6 Tracks Reference

`MultiViewTracks.tracks.`**`interpolate_subtracks`**(*sub_tracks*)
Linearly interpolate X, Y and Z components of sub-track dictionary.

`MultiViewTracks.tracks.`**`interpolate_tracks`**(*tracks*)
Linearly interpolate X, Y and Z components of track dictionary.

`MultiViewTracks.tracks.`**`interpolate_trajectory`**(*trajectory*)
Linearly interpolate X, Y and Z of one trajectory or sub-trajectory.

`MultiViewTracks.tracks.`**`rotate_tracks`**(*tracks*, *pca*)
Rotate tracks with a given pca transform.

Parameters

- **`tracks`** (*`dict`*) – A dictionary containing the tracks
- **`pca`** (*`sklearn.decomposition.PCA`*) – A fitted PCA instance used for transformation

Returns The rotated tracks

Return type dict

`MultiViewTracks.tracks.`**`scale_tracks`**(*tracks*, *scale*)
Scale tracks with a given scale.

Parameters

- **`tracks`** (*`dict`*) – A dictionary containing the tracks
- **`scale`** (*`float`*) – The scale used for transformation

Returns The scales tracks

Return type dict

`MultiViewTracks.tracks.`**`tracks_from_pooled`**(*pooled*)
Convert tracks dictionary from a dictionary with wide table format.

`MultiViewTracks.tracks.`**`tracks_from_subtracks`**(*sub_tracks*)
Returns tracks joined from given sub-tracks

`MultiViewTracks.tracks.`**`tracks_to_pooled`**(*tracks*)
Convert the tracks dictionary into a dictionary with wide table format.

`MultiViewTracks.tracks.`**`tracks_to_subtracks`**(*tracks*, *max_dist*)
Split tracks into sub-trajectories.

Parameters

- **`tracks`** (*`dict`*) – The tracks dictionary
- **`max_dist`** (*`float`*) – Maximum distance between consequtive frames that is allowed in a sub-trajectory

Returns Tracks dictionary with sub-trajectories

Return type dict

`MultiViewTracks.tracks.`**`trajectory_from_subtrajectories`**(*sub_trajectories*)
Returns a trajectory joined from sub-trajectories

`MultiViewTracks.tracks.`**`trajectory_to_subtrajectories`**(*trajectory*, *max_dist*)
Split one trajectory into sub-trajectories with specified maximum distance.

2.7 Utils Reference

`MultiViewTracks.utils.`**`compute_reprojection_errors`**(*tracks*, *tracks_reprojected*, *identities=[]*)
Computes point-wise reprojection errors (distances) between tracks and their reprojections.

Parameters

- **`tracks`** (*`dict`*) – A dictionary containing the original tracks
- **`tracks_reprojected`** (*`dict`*) – A dictionary containing the reprojected tracks
- **`identities`** (*`list, optional`*) – A list of trajectory identities for which the reprojection errors should be computed

Returns

- *list* – A list of arrays containing the reprojection errors for each identity
- *array* – An array containing the respective identities

`MultiViewTracks.utils.`**`load`**(*file_name*)

Loads a python object from a .pkl file

Parameters **`file_name`** (*`str`*) – File path of saved object

Returns Loaded python object

Return type object

`MultiViewTracks.utils.`**`plot_tracks_2d`**(*tracks*, *ax=None*, *figsize=(30, 30)*, *show=True*, *style='scatter'*, *size=1.0*)

Plots the x and y components of tracks. Can be used to visualize reprojection errors.

Parameters

- **`tracks`** (*`dict`*) – A dictionary containing the tracks
- **`ax`** (*`matplotlib.pyplot.Axes, optional`*) – Axes for plotting
- **`figsize`** (*`(int, int), optional`*) – Size of the matplotlib output if ax is not specified. Defaults to (30, 30)
- **`show`** (*`bool, optional`*) – Show plot calling plt.show. Defaults to True.
- **`style`** (*`str, optional`*) – Plot style, one of "line", "scatter" or "errors". Defaults to "scatter".
- **`size`** (*`float, optional`*) – Line width or marker size. Defaults to 1.0.

Returns

Return type matplotlib.pyplot.Axes

`MultiViewTracks.utils.`**`pointcloud_to_ply`**(*point_cloud*)

Returns pre-formatted ply points from input points, use Scene.get_pointcloud

`MultiViewTracks.utils.`**`read_next_bytes`**(*fid*, *num_bytes*, *format_char_sequence*, *endian_character='<'*)

Read next bytes of a COLMAP .bin file.

See https://github.com/colmap/colmap/blob/dev/scripts/python/read_model.py for reference.

`MultiViewTracks.utils.`**`save`**(*dump*, *file_name*)

Save to a .pkl file

Parameters

- **`dump`** (*`object`*) – Python object to save
- **`file_name`** (*`str`*) – File path of saved object

Returns Successful save?

Return type bool

`MultiViewTracks.utils.`**`tracks_to_ply`**(*tracks*, *uniform_color=None*)

Prepare tracks for ply file save.

Parameters

- **`tracks`** (*`dict`*) – A tracks dictionary, not a pooled dictionary, must contain z component
- **`uniform_color`** (*`(int, int, int), optional`*) – A shared color used for all tracks, otherwise random RGB generation

Returns A list of lists of per individual pre-formatted ply points (x y z r g b a)

Return type list

`MultiViewTracks.utils.`**`triangulate_point`**(*pts_2d*, *views*)

Triangulate points from multiple views using either OpenCV.triangulatePoints or DLT.

See https://github.com/opencv/opencv_contrib/blob/master/modules/sfm/src/triangulation.cpp for reference.

`MultiViewTracks.utils.`**`write_ply`**(*pts_ply*, *file_name*)

Write points to a .ply file for visualization

Parameters

- **`pts_ply`** (*`list`*) – The prepared points in ply format, for example using tracks_to_ply
- **`file_name`** (*`str`*) – The file name of the saved file

Returns Successful save?

Return type bool

genindex

CHAPTER 3

References

We use COLMAP [2016sfm], [2016mvs], [2016vote], a general-purpose Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline to reconstruct camera paths and orientations from videos. This is necessary when using a moving camera setup for triangulating animal positions in 3D from multiple-view trajectories. We found COLMAP to be fit for this task, as it is well-documented, open-source and easily-accessible.

Bibliography

[2016sfm] Schönberger, J. L., & Frahm, J. M. (2016). Structure-from-motion revisited. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4104-4113).

[2016mvs] Schönberger, J. L., Zheng, E., Frahm, J. M., & Pollefeys, M. (2016, October). Pixelwise view selection for unstructured multi-view stereo. In European Conference on Computer Vision (pp. 501-518). Springer, Cham.

[2016vote] Schönberger, J. L., Price, T., Sattler, T., Frahm, J. M., & Pollefeys, M. (2016, November). A vote-and-verify strategy for fast spatial verification in image retrieval. In Asian Conference on Computer Vision (pp. 321-337). Springer, Cham.

Python Module Index

Index