OpenSfM Documentation

Release 0.1.0

Mapillary

Mar 01, 2018

Contents

1 Building 1

2 Using 3

3 Dataset Structure 11

4 Geometric Models 13

5 Camera Coordinate System and Conventions 15

6 Incremental reconstruction algorithm 21

7 Splitting a large dataset into smaller submodels 23

8 Reporting 27

9 Code Documentation 31

10 Python 2 and 3 compatibility 33

11 Indices and tables 35

CHAPTER 1

Building

OpenSfM code is available at Github.

OpenSfM depends on the following libraries that need to be installed before building it.

- OpenCV
- OpenGV
- Ceres Solver
- Boost Python
- NumPy, SciPy, Networkx, PyYAML, exifread

Once the dependencies have been installed, you can build OpenSfM by running the following command from the main folder:

```
python setup.py build
```

1.1 Installing dependencies on Ubuntu

See this Dockerfile for the commands to install all dependencies on Ubuntu 14.04. The steps are

1. Install OpenCV, Boost Python, NumPy, SciPy using apt-get
2. Install python requirements using pip
3. Clone, build and install OpenGV following the receipt in the Dockerfile
4. Build and Install the Ceres solver from its source using the `-fPIC` compilation flag.

1.2 Installing dependencies on MacOSX

Install OpenCV, boost python and the Ceres solver using:

```
brew tap homebrew/science
brew install opencv
brew install homebrew/science/ceres-solver
brew install boost-python
sudo pip install -r requirements.txt
```

And install OpenGV using:

```
brew install eigen
git clone https://github.com/paulinus/opengv.git
cd opengv/build
cd opengv/build
cmake .. -DBUILD_TESTS=OFF -DBUILD_PYTHON=ON
make install
```

Be sure to update your `PYTHONPATH` to include `/usr/local/lib/python2.7/site-packages` where OpenCV and OpenGV have been installed. For example:

```
export PYTHONPATH=/usr/local/lib/python2.7/site-packages:$PYTHONPATH
```

1.3 Note on OpenCV 3

When running OpenSfM on top of OpenCV version 3.0 the OpenCV Contrib modules are required for extracting SIFT or SURF features.

1.4 Building the documentation

To build the documentation and browse it locally use:

```
cd doc
make livehtml
```

and browse http://localhost:8001/

CHAPTER 2

Using

2.1 Quickstart

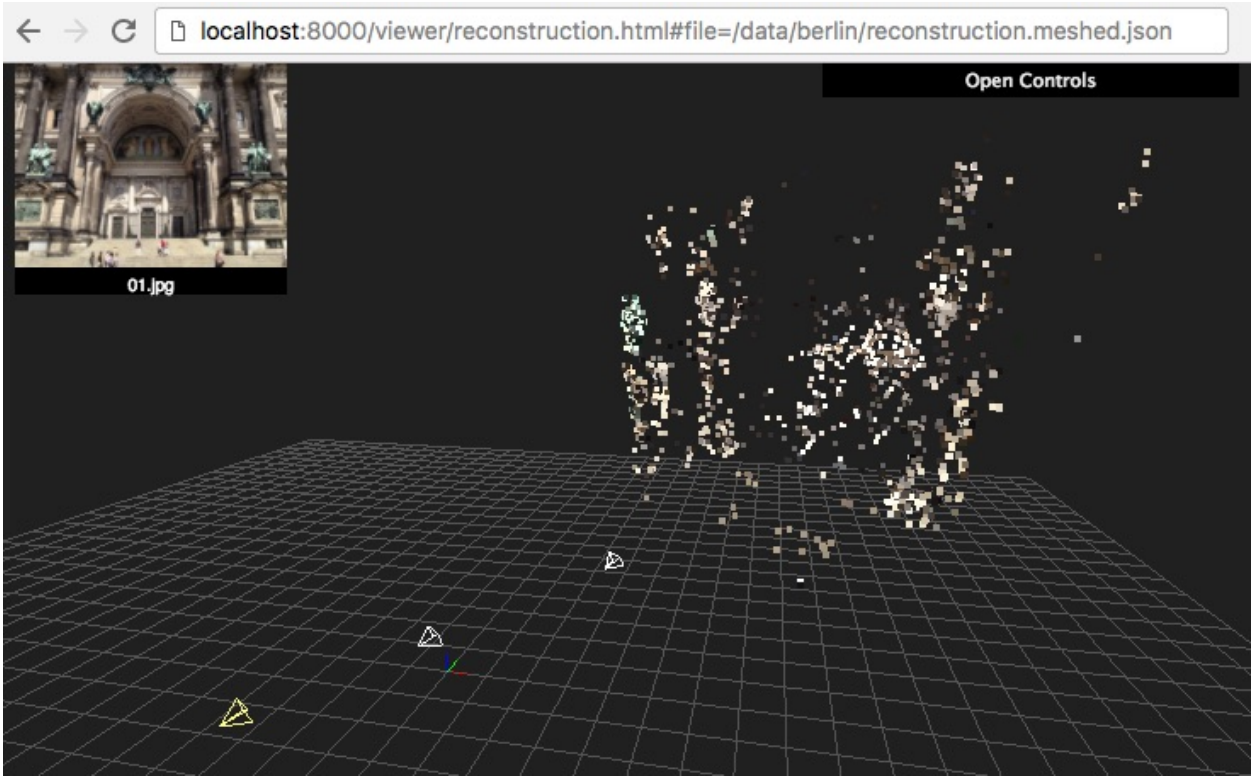
An example dataset is available at `data/berlin`. You can reconstruct it using by running:

```
bin/opensfm_run_all data/berlin
```

This will run the entire SfM pipeline and produce the file `data/berlin/reconstruction.meshed.json` as output. To visualize the result you can start a HTTP server running:

```
python -m SimpleHTTPServer
```

and then browse http://localhost:8000/viewer/reconstruction.html#file=/data/berlin/reconstruction.meshed.json You should see something like



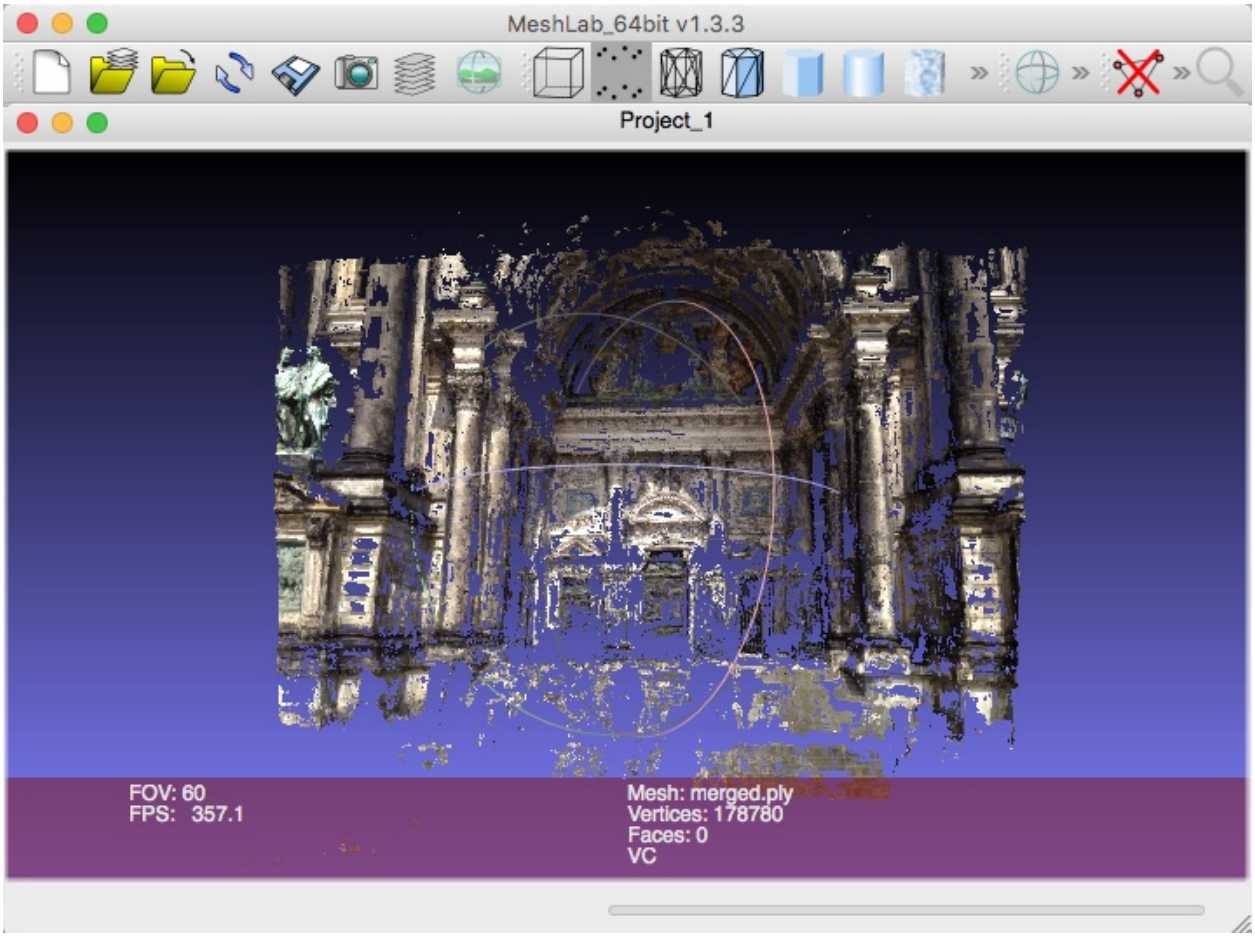
You can click twice on an image to see it. Then use arrows to move between images.

If you want to get a denser point cloud, you can run:

```
bin/opensfm undistort data/berlin
bin/opensfm compute_depthmaps data/berlin
```

This will run dense multiview stereo matching and produce a denser point cloud stored in `data/berlin/depthmaps/merged.ply`. You can visualize that point cloud using MeshLab or any other viewer that supports PLY files.

For the Berlin dataset you should get something similar to this



To reconstruct your own images,

1. put some images in `data/DATASET_NAME/images/`, and
2. copy `data/berlin/config.yml` to ``data/DATASET_NAME/config.yaml`

2.2 Reconstruction Commands

There are several steps required to do a 3D reconstruction including feature detection, matching, SfM reconstruction and dense matching. OpenSfM performs these steps using different commands that store the results into files for other commands to use.

The single application `bin/opensfm` is used to run those commands. The first argument of the application is the command to run and the second one is the dataset to run the commands on.

Here is the usage page of `bin/opensfm`, which lists the available commands:

```
usage: opensfm [-h] command ...

positional arguments:
  command            Command to run
    extract_metadata
                     Extract metadata form images' EXIF tag
    detect_features  Compute features for all images
    match_features   Match features between image pairs
```

```
    create_tracks     Link matches pair-wise matches into tracks
    reconstruct       Compute the reconstruction
    mesh              Add delaunay meshes to the reconstruction
    undistort         Save radially undistorted images
    compute_depthmaps
                      Compute depthmap
    export_ply        Export reconstruction to PLY format
    export_openmvs    Export reconstruction to openMVS format
    export_visualsfm
                      Export reconstruction to NVM_V3 format from VisualSfM

optional arguments:
  -h, --help          show this help message and exit
```

2.2.1 extract_metadata

This commands extracts EXIF metadata from the images an stores them in the `exif` folder and the `camera_models.json` file.

The following data is extracted for each image:

- `width` and `height`: image size in pixels
- `gps` `latitude`, `longitude`, `altitude` and `dop`: The GPS coordinates of the camera at capture time and the corresponding Degree Of Precission). This is used to geolocate the reconstruction.
- `capture_time`: The capture time. Used to choose candidate matching images when the option `matching_time_neighbors` is set.
- `camera orientation`: The EXIF orientation tag (see this exif orientation documentation). Used to orient the reconstruction straigh up.
- `projection_type`: The camera projection type. It is extracted from the GPano metadata and used to determine which projection to use for each camera. Supported types are *perspective*, *equirectangular* and *fisheye*.
- `focal_ratio`: The focal length provided by the EXIF metadata divided by the sensor width. This is used as initialization and prior for the camera focal length parameter.
- `make` and `model`: The camera make and model. Used to build the camera ID.
- `camera`: The camera ID string. Used to identify a camera. When multiple images have the same camera ID string, they will be assumed to be taken with the same camera and will share its parameters.

Once the metadata for all images has been extracted, a list of camera models is created and stored in `camera_models.json`. A camera is created for each diferent camera ID string found on the images.

For each camera the following data is stored:

- `width` and `height`: image size in pixels
- `projection_type`: the camera projection type
- `focal`: The initial estimation of the focal length (as a multiple of the sensor width).
- `k1` and `k2`: The initial estimation of the radial distortion parameters. Only used for *perspective* and *fisheye* projection models.
- `focal_prior`: The focal length prior. The final estimated focal length will be forced to be similar to it.
- `k1_prior` and `k2_prior`: The radial distortion parameters prior.

Providing your own camera parameters

By default, the camera parameters are taken from the EXIF metadata but it is also possible to override the default parameters. To do so, place a file named `camera_models_overrides.json` in the project folder. This file should have the same structure as `camera_models.json`. When running the `extract_metadata` command, the parameters of any camera present in the `camera_models_overrides.json` file will be copied to `camera_models.json` overriding the default ones.

Simplest way to create the `camera_models_overrides.json` file is to rename `camera_models.json` and modify the parameters. You will need to rerun the `extract_metadata` command after that.

Here is a spherical 360 images dataset example using `camera_models_overrides.json` to specify that the camera is taking 360 equirectangular images.

2.2.2 detect_features

This command detect feature points in the images and stores them in the *feature* folder.

2.2.3 match_features

This command matches feature points between images and stores them in the *matches* folder. It first determines the list of image pairs to run, and then run the matching process for each pair to find corresponding feature points.

Since there are a lot of possible image pairs, the process can be very slow. It can be speeded up by restricting the list of pairs to match. The pairs can be restricted by GPS distance, capture time or file name order.

2.2.4 create_tracks

This command links the matches between pairs of images to build feature point tracks. The tracks are stored in the *tracks.csv* file. A track is a set of feature points from different images that have been recognized to correspond to the same pysical point.

2.2.5 reconstruct

This command runs the incremental reconstruction process. The goal of the reconstruction process is to find the 3D position of tracks (the *structure*) together with the position of the cameras (the *motion*). The computed reconstruction is stored in the `reconstruction.json` file.

2.2.6 mesh

This process computes a rough triangular mesh of the scene seen by each images. Such mesh is used for simulating smooth motions between images in the web viewer. The reconstruction with the mesh added is stored in `reconstruction.meshed.json` file.

Note that the only difference between `reconstruction.json` and `reconstruction.meshed.json` is that the later contains the triangular meshes. If you don't need that, you only need the former file and there's no need to run this command.

2.2.7 undistort

This command creates undistorted version of the reconstruction, tracks and images. The undistorted version can later be used for computing depth maps.

2.2.8 compute_depthmaps

This commands computes a dense point cloud of the scene by computing and merging depthmaps. It requires an undistorted reconstructions. The resulting depthmaps are stored in the `depthmaps` folder and the merged point cloud is stored in `depthmaps/merged.ply`

2.3 Configuration

TODO explain config.yaml and the available parameters

2.4 Ground Control Points

When EXIF data contains GPS location, it is used by OpenSfM to georeference the reconstruction. Additionally, it is possible to use ground control points.

Ground control points (GCP) are landmarks visible on the images for which the geospatial position (latitude, longitude and altitude) is known. A single GCP can be observed in one or more images.

OpenSfM uses GCP in two steps of the reconstruction process: alignment and bundle adjustment. In the alignment step, points are used to globaly move the reconstruction so that the observed GCP align with their GPS position. Two or more observations for each GCP are required for it to be used during the aligment step.

In the bundle adjustment step, GCP observations are used as a constraint to refine the reconstruction. In this step, all ground control points are used. No minimum number of observation is required.

2.4.1 File format

GCPs can be specified by adding a text file named `gcp_list.txt` at the root folder of the dataset. The format of the file should be as follows.

- The first line should contain the name of the projection used for the geo coordinates.
- The following lines should con should contain the data for each ground control point observation. One per line and in the format:

```
<geo_x> <geo_y> <geo_z> <im_x> <im_y> <image_name>
```

  Where `<geo_x> <geo_y> <geo_z>` are the geospatial coordinates of the GCP and `<im_x> <im_y>` are the pixel coordinates where the GCP is observed.

2.4.2 Supported projections

The geospatial coordinates can be specified in one the following formats.

- WGS84: This is the standard latitude, longitude coordinates used by most GPS devices. In this case, `<geo_x> = longitude`, `<geo_y> = latitude` and `<geo_z> = altitude`

- UTM: UTM projections can be specified using a string projection string such as `WGS84 UTM 32N`, where 32 is the region and N is . In this case, `<geo_x> = E`, `<geo_y> = N` and `<geo_z> = altitude`
- proj4: Any valid proj4 format string can be used. For example, for UTM 32N we can use `+proj=utm +zone=32 +north +ellps=WGS84 +datum=WGS84 +units=m +no_defs`

2.4.3 Example

This file defines 2 GCP whose coordinates are specified in the WGS84 standard. The first one is observed in both `01.jpg` and `02.jpg`, while the second one is only observed in `01.jpg`

```
WGS84
13.400740745 52.519134104 12.0792090446 2335.0 1416.7 01.jpg
13.400740745 52.519134104 12.0792090446 2639.1 938.0 02.jpg
13.400502446 52.519251158 16.7021233002 766.0 1133.1 01.jpg
```

CHAPTER 3

Dataset Structure

```
project/
├── config.yaml
├── images/
├── masks/
├── gcp_list.txt
├── exif/
├── features/
├── matches/
├── tracks.csv
├── reconstruction.json
├── reconstruction.meshed.json
├── undistorted/
├── undistorted_tracks.json
├── undistorted_reconstruction.json
└── depthmaps/
    └── merged.ply
```

CHAPTER 4

Geometric Models

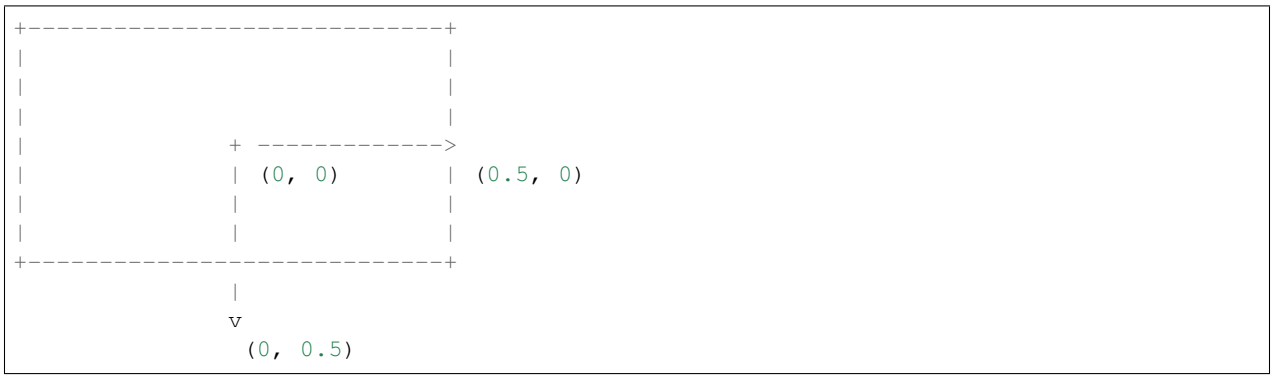
TODO

4.1 Coordinate Systems

4.1.1 Normalized Image Coordinates

The 2d position of a point in images is stored in what we will call *normalized image coordinates*. The origin is in the middle of the image. The x coordinate grows to the right and y grows downwards. The larger dimension of the image is 1.

This means, for example, that all the pixels in an image with aspect ratio 4:3 will be contained in the intervals `[-0.5, 0.5]` and `[3/4 * (-0.5), 3/4 * 0.5]` for the X and Y axis respectively.



Normalized coordinates are independent of the resolution of the image and give better numerical stability for some multi-view geometry algorithms than pixel coordinates.

4.1.2 Pixel Coordinates

Many OpenCV functions that work with images use *pixel coordinates*. In that reference frame, the origin is at the center of the top-left pixel, x grow by one for every pixel to the right and y grows by one for every pixel downwards. The bottom-right pixel is therefore at `(width - 1, height - 1)`.

The transformation from normalised image coordinates to pixel coordinates is

$$H = \begin{pmatrix} \max(w,h) & 0 & \frac{w-1}{2} \\ 0 & \max(w,h) & \frac{h-1}{2} \\ 0 & 0 & 1 \end{pmatrix},$$

and its inverse

$$H^{-1} = \begin{pmatrix} 1 & 0 & -\frac{w-1}{2} \\ 0 & 1 & -\frac{h-1}{2} \\ 0 & 0 & \max(w,h) \end{pmatrix},$$

where w and h being the width and height of the image.

4.1.3 World Coordinates

The position of the reconstructed 3D points is stored in *world coordinates*. In general, this is an arbitrary euclidean reference frame.

When GPS data is available, a topocentric reference frame is used for the world coordinates reference. This is a reference frame that with the origin somewhere near the ground, the X axis pointing to the east, the Y axis pointing to the north and the Z axis pointing to the zenith. The latitude, longitude, and altitude of the origin are stored in the `reference_lla.json` file.

When GPS data is not available, the reconstruction process makes its best to rotate the world reference frame so that the vertical direction is Z and the ground is near the *z = 0* plane. It does so by assuming that the images are taken from similar altitudes and that the up vector of the images corresponds to the up vector of the world.

4.1.4 Camera Coordinates

The *camera coordinate* reference frame has the origin at the camera's optical center, the X axis is pointing to the right of the camera the Y axis is pointing down and the Z axis is pointing to the front. A point in front of the camera has positive Z camera coordinate.

The pose of a camera is determined by the rotation and translation that converts world coordinates to camera coordinates.

4.2 Camera Models

TODO

CHAPTER 5

Camera Coordinate System and Conventions

5.1 Camera

The pose of a camera, conceptually, consists of two things:

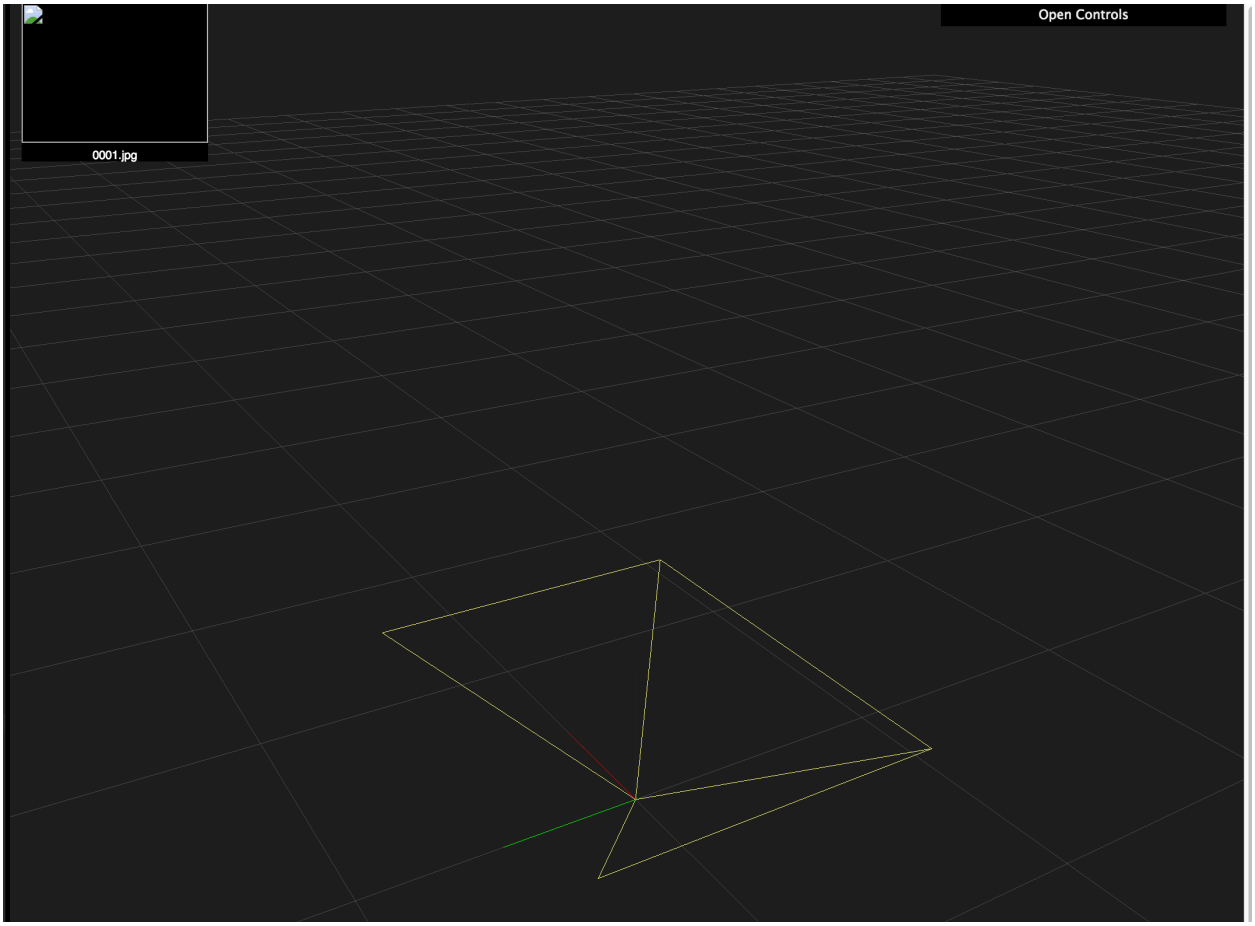
1. Which direction does it face in, i.e. its local coordinate axes
2. Where is it, i.e. the position of the camera origin

5.1.1 Local coordinate system of camera

These online docs say that, from the POV of a camera (a.k.a. a `Shot` object):

- The z-axis points **forward**
- The y-axis points **down**
- The x-axis points to the **right**

I can confirm, after experimentation, that this is accurate. (In the 3D reconstruction viewer, the axes go Red, Green, Blue: x, y, z.)



The OpenSfM `Pose` class contains a `rotation` field, representing the local coordinate system as an **axis-angle vector**.

- The **direction** of this 3D vector represents the **axis** around which to rotate.
- The **length** of this vector is the **angle** to rotate around said axis.

Sounds like it makes sense, but when it actually comes to working with code, all the hidden conventions crawl out of the shadows.

First is the obviously unstated **unit of angular measurement**. In computer maths libraries it's generally safe to assume everything's in **radians**, and it certainly looks like that here.

Next, a "rotation axis" is really just a *line*. But a vector, even a *unit* vector, defines a line $\lambda\mathbf{v}$ with *orientation*. One direction is "positive scalar", the opposite is "negative scalar". Could there be a difference between rotating around $\mathbf{v}$ and rotating around $-\mathbf{v}$?

Look at a clock face-on. Pick the axis from its face to your face. It's rotating clockwise around this axis. Now turn the clock around to face the opposite direction. Looking through the back of the clock, the hands rotate *anticlockwise* around the *negative* of the original axis. So rotating by θ around $-\mathbf{v}$ is the **same** as rotating $-\theta$ around $\mathbf{v}$.

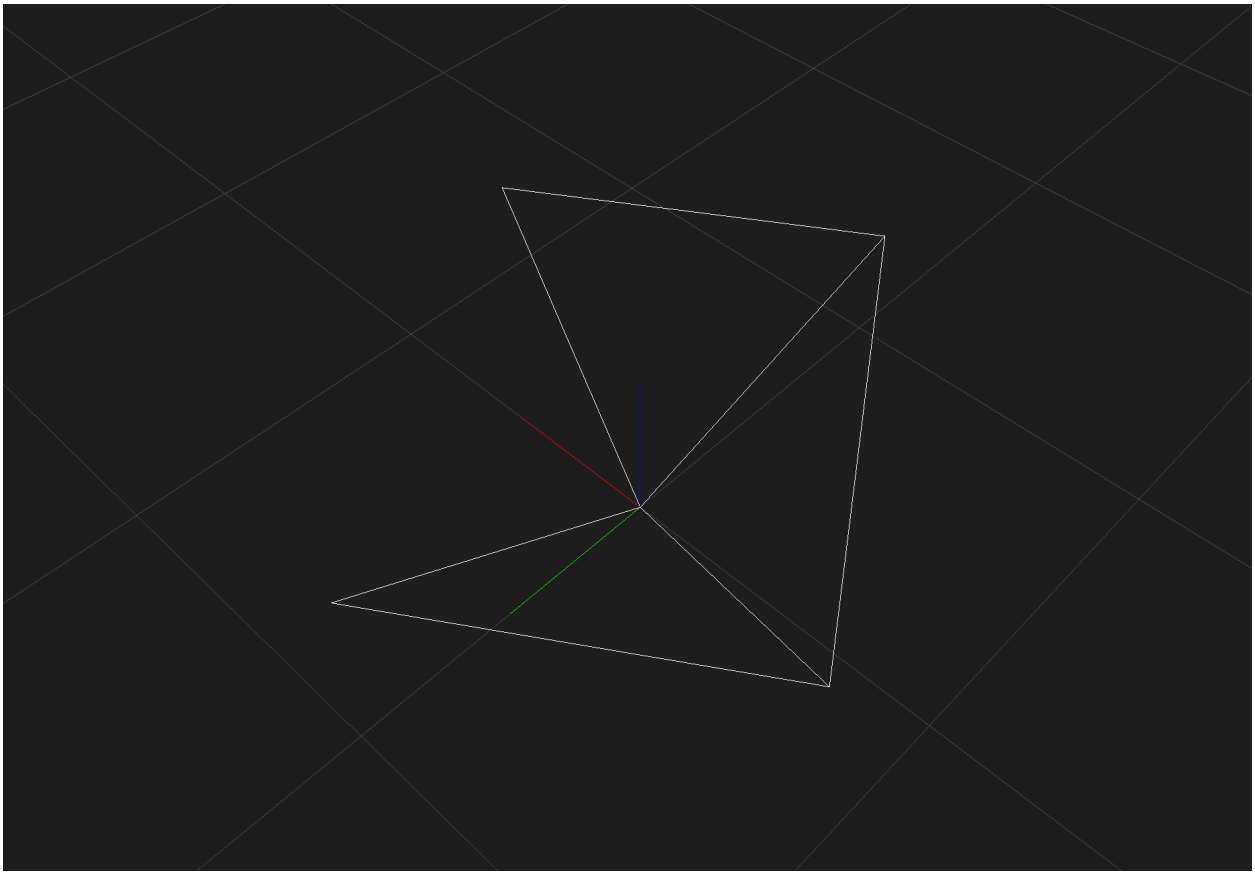
Even if we know that two representations are opposite, this still doesn't tell us which is which. What is a "rotation around the z-axis"? This sounds like asking whether each axis is clockwise or anticlockwise, but even this depends on which way you're looking…

Instead, the real question being asked is: Does a rotation by a small positive angle, around the positive z axis, rotate in the *x-to-y* direction, or the *y-to-x* direction? And likewise for the other axes.

To find out, I set the rotation vectors to rotate 1 radian around each of the axes. Results are:

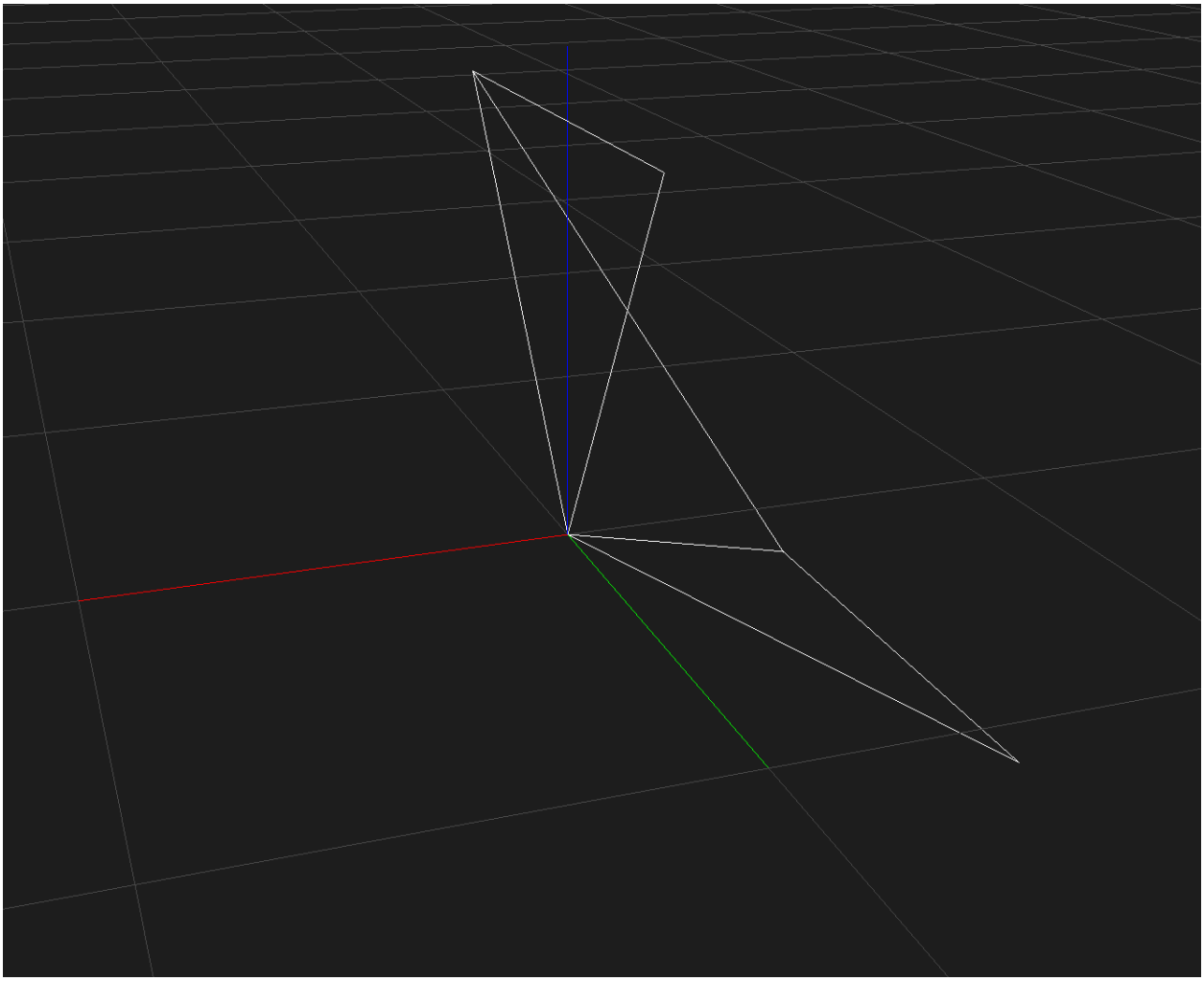
“Rotate around Z” is Y-to-X

With `pose.rotation = [0, 0, 1]`:



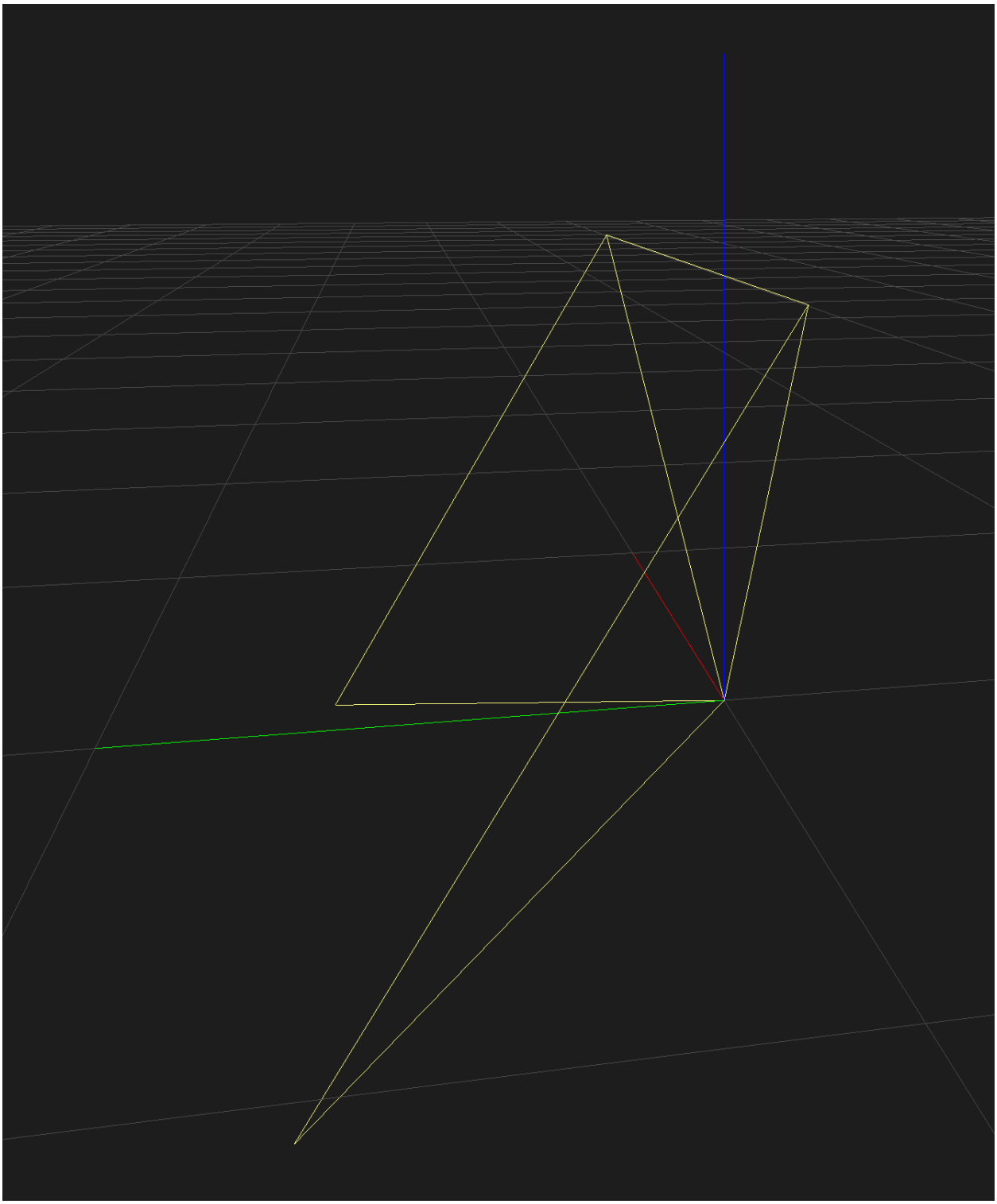
“Rotate around Y” is X-to-Z

With `pose.rotation = [0, 1, 0]`:



“Rotate around X” is Z-to-Y

With `pose.rotation = [1, 0, 0]`:



It basically works like this (apologies for ASCII art):

```
      Z ----+
    /   |   |
  /     |   V
^       |____Y
```

```
|   /     /
| /      /
X <-----+
```

5.1.2 Where is the camera?

Conceptually, this is a simple translation, in world coordinates, to the camera's origin position.

OpenSfM, however, chooses **not** to store the "camera origin" in `Pose` objects. Instead, it stores the **camera coordinates of the world origin** in the `translation` field.

These obviously depend on the position and, in particular, **rotation** of the camera. They are automatically calculated by the `pose.set_origin(origin)` method, using the **current** `pose.rotation`.

Because of this dependency, if the camera turns around, the `translation` will need updating. But `pose.set_rotation_matrix()` **does not do this.** So you should never call `set_origin()` **before** `set_rotation_matrix()`. Only set the origin afterwards.

The case where you only want to change the rotation, while keeping the position the same, is a bit subtle. You will have to manually update `pose.translation` after setting the rotation, but to what? You can't call `get_origin()` *after* updating the rotation, because this will calculate the origin from `translation` using the *new* rotation instead of the old one. The `translation` value only makes sense the coordinate system that set it. It must be kept in sync with `rotation`, something that seems to have been overlooked in the version at the time of writing.

Solution to safely set pose rotation:

```
org = pose.get_origin()       # save where the camera actually is
pose.set_rotation_matrix(R)   # set the rotation
pose.set_origin(org)          # update the translation property accordingly...
```

CHAPTER 6

Incremental reconstruction algorithm

OpenSfM implements an incremental structure from motion algorithm. This is reconstruction algorithm that starts building a reconstruction of a single image pair and then iteratively add the other images to the reconstruction one at a time.

The algorithm is implemented in the `reconstruction.py` module and the main entry point is the `incremental_reconstruction()` function.

The algorithm has three main steps:

1. Find good initial pairs
2. Bootstrap the reconstruction with two images
3. Grow the reconstruction by adding images one at a time

If after step 3 there are images that have not yet been reconstructed, steps 2 and 3 are repeated to generate more reconstructions.

6.1 1. Finding good initial pairs

To compute the initial reconstruction using two images, there needs to be enough parallax between them. That is, the camera should have been displaced between the two shots, and the displacement needs to be large enough compared to the distance to the scene.

To compute whether there is enough parallax, we start by trying to fit a rotation only camera model to the two images. We only consider image pairs that have a significant portion of the correspondences that can not be explained by the rotation model. We compute the number of outliers of the model and accept it only if the portion of outliers is larger than 30%.

The accepted image pairs are sorted by the number of outliers of the rotation only model.

This step is done by the `compute_image_pairs()` function.

6.2 2. Boostraping the reconstruction

To bootstrap the reconstruction, we use the first image pair. If initialization fails we try with the next on the list. If the initialization works, we pass it to the next step to grow it with more images.

The reconstruction from two views can be done by two algorithms depending on the geometry of the scene. If the scene is flat, a plane-based initialization is used, if it is not flat, then the five-point algorithm is used. Since we do not know a priori if the scene is flat, both initializations are computed and the one that produces more points is retained (see the `two_view_reconstruction_general()` function).

If the pair gives enough inliers we initialize a reconstruction with the corresponding poses, triangulate the matches and bundle adjust it.

6.3 3. Growing the reconstruction

Given the initial reconstruction with two images, more images are added one by one starting with the one that sees more of the reconstructed points.

To add an image it needs first needs to be aligned to the reconstruction. This is done by finding the camera position that makes the reconstructed 3D points project to the corresponding position in the new image. The process is called resectioning and is done by the `resect()` function.

If resectioning works, the image is added to the reconstruction. After adding it, all features of the new image that are also seen in other reconstructed images are triangulated. If needed, the reconstruction is then bundle adjusted and eventually all features are re-triangulated. The parameters `bundle_interval`, `bundle_new_points_ratio`, `retriangulation` and `retriangulation_ratio` control when bundle and re-triangulation are needed.

Finally, if the GPS positions of the shots or Ground Control Points (GPS) are available, the reconstruction is rigidly moved to best align to those.

CHAPTER 7

Splitting a large dataset into smaller submodels

Large datasets can be slow to process. An option to speed up the reconstruction process is to split them into smaller datasets. We will call each of the small datasets a *submodel*. Smaller datasets run faster because they involve fewer images on each bundle adjustment iteration. Additionally, the reconstruction of the different submodels can be done in parallel.

Since the reconstructions of the submodels are done independently, they will not be necessarily aligned with each other. Only the GPS positions of the images and the ground control points will determine the alignment. When the neighboring reconstructions share cameras or points, it is possible to enforce the alignment of common cameras and points between the different reconstructions.

In the following, we describe the commands that help to split a large dataset and aligning the resulting submodels.

7.1 Creating submodels

The command `create_submodels` splits a dataset into submodels. The splitting is done based on the GPS position of the images. Therefore, it is required to run `extract_metadata` before so that the GPS positions are read from the image metadata.

Additionally, the feature extraction and matching can also be done before creating the submodels. This makes it possible for each submodel to reuse the features and matches of the common images.

The process to split a dataset into submodels is then:

```
bin/opensfm extract_metadata path/to/dataset
bin/opensfm detect_features path/to/dataset
bin/opensfm match_features path/to/dataset
bin/opensfm create_submodels path/to/dataset
```

7.1.1 Submodels dataset structure

The submodels are created inside the `submodels` folder. Each submodel folder is a valid OpenSfM dataset. The images, EXIF metadata, features, and matches are shared with the global dataset by using symbolic links.

```
project/
├── images/
├── opensfm/
├── image_list.txt
├── image_list_with_gps.csv    # list of original images with GPS position
├── exif
├── features                   # eventually
├── matches                    # eventually
└── submodels/
    ├── clusters_with_neighbors.geojson  # geojson file with all images as features␣
↪with corresponding submodel as a property
    ├── clusters_with_neighbors.npz
    ├── clusters.npz
    ├── image_list_with_gps.tsv
    ├── submodel_0000/
    │   ├── image_list.txt        # images of submodel_0000
    │   ├── config.yaml           # copy from global equivalent
    │   ├── images/               # link to global equivalent
    │   ├── exif/                 # link to global equivalent
    │   ├── features/             # link to global equivalent
    │   ├── matches/              # link to global equivalent
    │   ├── camera_models.json    # link to global equivalent
    │   └── reference_lla.json    # link to global equivalent
    │   submodel_0001/
    └── ...
```

7.1.2 Config parameters

The creation of the submodels can be tuned by different parameters.

There are two parameters controlling the size and overlap of the submodels. The parameters need to be adjusted to the size.

- `submodel_size` Average number of images per submodel. The splitting of the dataset is done by clustering image locations into groups. K-means clustering is used and `k` is set to be the number of images divided by `submodel_size`.
- `submodel_overlap` Radius of the overlapping region between submodels in meters. To be able to align the different submodels, there needs to be some common images between the neighboring submodels. Any image that is closer to a cluster than `submodel_overlap` it is added to that cluster.

The folder structure of the submodels can also be controlled using the following parameters. You shouldn't need to do change them.

- `submodels_relpath` Relative path to the submodels directory. Cluster information will be stored in this directory.
- `submodel_relpath_template` Template to generate the relative path to a submodel directory.
- `submodel_images_relpath_template` Template to generate the relative path to a submodel images directory.
- `submodel_use_symlinks` When true, global features and matches will be symlinked in each submodel so that they can be reused. When false, features and matches will need to be run for each submodel.

7.1.3 Providing the image groups

The `create_submodels` command clusters images into groups to decide the partition into submodels. If you already know how you want to split the dataset, you can provide that information and it will be used instead of the clustering algorithm.

The grouping can be provided by adding a file named `image_groups.txt` in the main dataset folder. The file should have one line per image. Each line should have two words: first the name of the image and second the name of the group it belongs to. For example:

```
01.jpg A
02.jpg A
03.jpg B
04.jpg B
05.jpg C
```

will create 3 submodels.

Starting from this groups, `create_submodels` will add to each submodel the images in the overlap area based on the `submodels_overlap` parameter.

7.2 Running the reconstruction for each submodel

Since each submodel is a valid OpenSfM dataset, the reconstruction can be run using the standard commands. Assuming features and matches have already been computed, we will need to run:

```
bin/opensfm create_tracks path/to/dataset/submodels/submodel_XXXX
bin/opensfm reconstruct path/to/dataset/submodels/submodel_XXXX
```

for each submodel. This can be run in parallel since the submodels are independent.

7.3 Aligning submodels

Once every submodel has a reconstruction, they can be aligned by using the command:

```
bin/opensfm align_submodels path/to/dataset
```

This command will load all the reconstructions, look for cameras and points shared between the reconstructions, and move each reconstruction rigidly in order best align the corresponding cameras and points.

CHAPTER 8

Reporting

OpenSfM commands write reports on the work done. Reports are stored in the `reports` folder in json format so that they can be loaded by programatically. Here is the list of reports produced and the data included.

8.1 Feature detection

The report on feature detection is stored in the file `features.json`. Its structure is as follow:

```
{
    "wall_time": {{ total time compting features }},
    "image_reports": [    # For each image
        {
            "wall_time": {{ feature extraction time }},
            "image": {{ image name }},
            "num_features": {{ number of features }}
        },
        ...
    ]
}
```

8.2 Matching

The report on matching is stored in the file `matches.json`. Its structure is as follow:

```
{
    "wall_time": {{ total time compting matches }},
    "pairs": {{ list of candidate image pairs }}
    "num_pairs": {{ number of candidate image pairs }},
    "num_pairs_distance": {{ number of pairs selected based on distance }},
    "num_pairs_time": {{ number of pairs selected based on time }},
```

```
    "num_pairs_order": {{ number of pairs selected based on order }},
}
```

8.3 Create tracks

The report on tracks creation is stored in the file `tracks.json`. Its structure is as follow:

```
{
    "wall_time": {{ total time computing tracks }}
    "wall_times": {
        "load_features": {{ time loading features }},
        "load_matches": {{ time loading matches }},
        "compute_tracks": {{ time computing tracks }},
    },
    "num_images": {{ number of images with tracks }},
    "num_tracks": {{ number of tracks }},
    "view_graph": {{ number of image tracks for each image pair }}
}
```

8.4 Reconstruction

The report on the reconstruction process is stored in the file `reconstruction.json`. Its structure is as follow:

```
{
    "wall_times": {
        "compute_reconstructions": {{ time computing the reconstruction }},
        "compute_image_pairs": {{ time computing the candidate initial pairs }},
        "load_tracks_graph": {{ time loading tracks }}
    },
    "num_candidate_image_pairs": {{ number of candidate image pairs for initializing␣
↪reconstructions }},
    "reconstructions": [  # For each reconstruction build
        {
            "bootstrap": {  # Initialization information
                "memory_usage": {{ memory usage at the end of the process }},
                "image_pair": {{ initial image pair }},
                "common_tracks": {{ number of common tracks of the image pair }},
                "two_view_reconstruction": {
                    "5_point_inliers": {{ number of inliers for the 5-point algorithm␣
↪}},
                    "plane_based_inliers": {{ number of inliers for the plane based␣
↪initialization }},
                    "method": {{ method used for initialization "5_point" or "plane_
↪based" }}
                },
                "triangulated_points": {{ number of triangulated points }},
                "decision": {{ either "Success" or the reason for failure }},
            },
            "grow": {  # Incremental growth information
                "steps": [  # For every growth step
                    {
                        "image": {{ image name }},
                        "resection": {
```

```
                            "num_inliers": {{ number of inliers }},
                            "num_common_points": {{ number of reconstructed points␣
↪visible on the new image }}
                        },
                        "triangulated_points": {{ number of newly triangulated points␣
↪}},
                        "memory_usage": {{ memory usage after adding the image }},
                        "bundle": {
                            "wall_times": {
                                "setup": {{ time setting up bundle }},
                                "run": {{ time running bundle }},
                                "teardown": {{ time updating the values after bundle }
↪},
                            },
                            "brief_report": {{ Ceres brief report }}
                        },
                    }
                ]
            }
        }
    ],
    "not_reconstructed_images": {{ images that could not be reconstructed }},
}
```

CHAPTER 9

Code Documentation

9.1 Dataset I/O

9.2 Reconstruction Types

9.3 Features

9.4 Matching

9.5 Incremental Reconstruction

CHAPTER 10

Python 2 and 3 compatibility

The code must be compatible Python versions 2.7 and 3.6+.

Here are the basic rules to follow for all new code. Existing code needs to be revised to follow these rules. See the official guide for general rules.

10.1 Absolute imports

Always use absolute imports. Import absolute imports from future to disable relative imports.

```
from __future__ import absolute_import
```

10.2 Print

Use loggers instead of print when possible. When using print, use it as a function with parenthesis. Include print from future to disable Python 2 style print.

```
from __future__ import print_function
```

10.3 Division

Always add

```
from __future__ import division
```

to make sure that division acts the Python 3 way. Use `//` when you need integer division.

10.4 Text

All text should be Unicode. Encode Unicode text from and to bytes using UTF-8 encoding when doing I/O operations. Encoding and decoding is done as close as possible to the I/O operations. Some people refer to that as the Unicode sandwich.

By default, string literals are byte strings in Python 2 and Unicode strings in Python 3. Import Unicode literals from future to make all string literals Unicode in any Python version.

```
from __future__ import unicode_literals
```

When you really need a byte string literal create it with `b""`.

Use `opensfm.io.open_rt` and `opensfm.io.open_wt` to open text files for reading and writing. This functions take care of decoding and encoding UTF-8 files from and into Unicode.

Use `opensfm.io.json_load`, `opensfm.io.json_loads`, `opensfm.io.json_dump`, and `opensfm.io.json_dumps` to encode and decode JSON documents. This functions make sure that the JSON representation is Unicode text and that, when written in a file, it is written using UTF-8 encoding.

CHAPTER 11

Indices and tables

- genindex