CFTM User Manual

Version 1.0

Writen by Xinlong Li, Mingtao Ding

Update 2024.06.24

Big Data Center for Geosciences and Satellites

Chang'an University

Journal Paper: Xinlong Li, Mingtao Ding*, Zhenhong Li*, Peng Cui, 2024, Common-Feature-Track-Matching approach for multi epoch UAV photogrammetry co-registration, ISPRS Journal of Photogrammetry and Remote Sensing (under review).

Patent Number: ZL 202311672166.3

Technical Support: Xinlong Li, Tel: +86 15583308860, Email: xinlong.li@chd.edu.cn

Introduction

CFTM is a tool for high-precision co-registration of multi-epoch UAV photogrammetry.

CFTM is implemented through the Agisoft Metashape Python API and optionally COLMAP.

CFTM is a Co-Structure-from-Motion (CoSfM) approach that allows for the simultaneous photogrammetry and co-registration of multi-epoch aerial imagery. Utilizing our novel matching strategy—"Common-Feature-Track-Matching"—and a specialized optimization algorithm, it can generate dense and evenly distributed Common Tie Points (CTPs), enabling high-precision co-registration of aerial images. The output of CFTM is the optimized image poses, which means that the subsequently generated geospatial products (such as Digital Elevation Models (DEM), Digital Orthophoto Maps (DOM), and 3D mesh models) are directly aligned in the same coordinate system, facilitating subsequent high-precision spatiotemporal analysis.

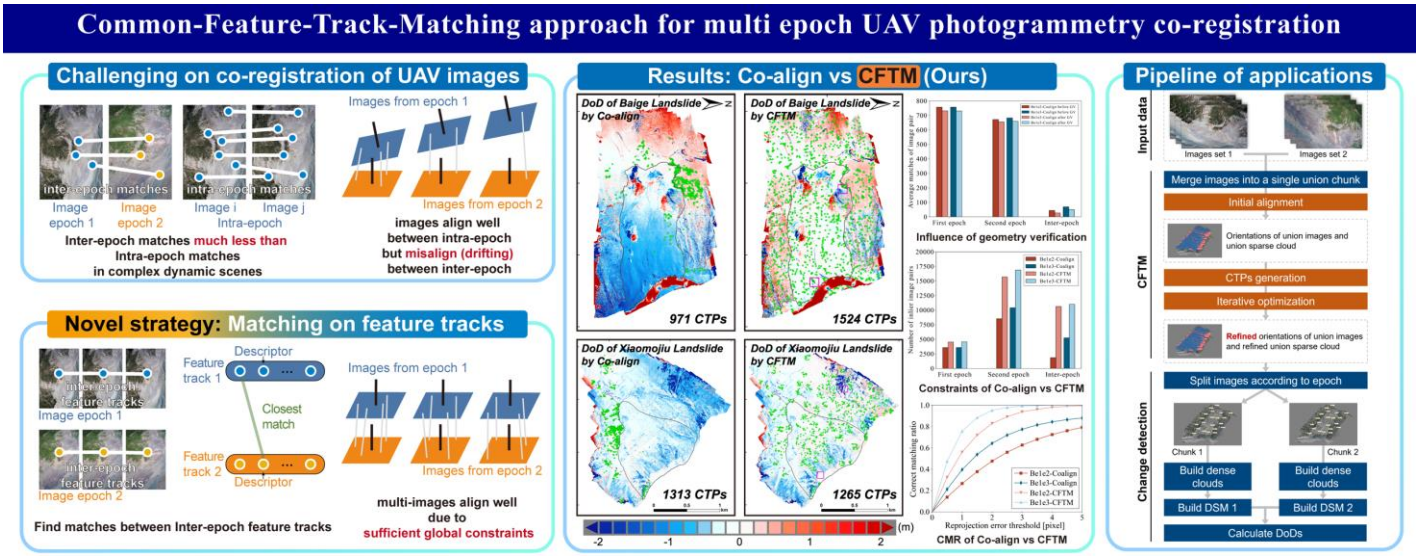
If you use this tool in your research, please cite the following paper:

```
@article{LI2024392,
title = {Common-feature-track-matching approach for multi-epoch UAV photogrammetry co-registration},
journal = {ISPRS Journal of Photogrammetry and Remote Sensing},
volume = {218},
pages = {392-407},
year = {2024},
issn = {0924-2716},
doi = {https://doi.org/10.1016/j.isprsjprs.2024.10.025},
url = {https://www.sciencedirect.com/science/article/pii/S0924271624004027},
author = {Xinlong Li and Mingtao Ding and Zhenhong Li and Peng Cui},
}
```

Abstract

Automatic co-registration of multi-epoch Unmanned Aerial Vehicle (UAV) image sets remains challenging due to the radiometric differences caused by complex dynamic scenes. Specifically, illumination changes and vegetation variations usually lead to insufficient and spatially unevenly distributed common tie points (CTPs), resulting in under-fitting of co-registration near the areas without CTPs. In this paper, we propose a novel Common-Feature-Track-Matching (CFTM) approach for UAV image co-registration, to alleviate the shortage of CTPs with complex dynamic scenes. Instead of matching features between multi-epoch images, we first search correspondences between multi-epoch feature tracks (i.e., groups of features corresponding to the same 3D points), which avoids the removal of matches due to unreliable estimation of the relative pose between inter-epoch image pairs. Then, the CTPs are triangulated from the successfully matched track pairs. Since the even distribution of CTPs is crucial for robust co-registration, a block-based strategy is designed, as well as for parallel computation. Finally, an iterative optimization algorithm is developed to gradually select the best CTPs to refine the poses of multi-epoch images. We assess the performance of our method on two challenging datasets. The results show that CFTM can acquire adequate and evenly distributed CTPs in complex dynamic scenes, thus achieving comparatively high co-registration accuracy approximately four times higher than the state-of-the-art in challenging scenario. Our code is available at https://github.com/lixinlong1998/CoSfM.

Graphic Abstract



Contents

1 Downloads

Source Code: https://github.com/lixinlong1998/CoSfM

Online Document (Chinese): https://blog.csdn.net/LXLng/article/details/136598055

Online Document (English): https://blog.csdn.net/LXLng/article/details/134613468

Example Data: https://pan.baidu.com/s/1s3BJk3cUs6cwj9PNI9s36g?pwd=cftm

2 Installation and Configuration

CFTM v1.0 is developed on Windows environment using the Python API of Agisoft Metashape Pro v1.8.5 software. Since Metashape Pro v1.8.5 does not provide interfaces for accessing feature points and raw feature matching results, feature point extraction is implemented using the feature extraction functionality of the open-source third-party library OpenCV (default) or COLMAP (optional), and feature matching analysis leverages COLMAP's feature matching functionality. In addition, some functionalities depend on third-party library functions. Although Metashape Pro v1.8.5 comes with a pre-installed Python 3.8 environment with some third-party libraries, additional installation and/or upgrading of certain third-party libraries are required.

2.1 Installing Agisoft Metashape Pro v1.8.5

If you are new to Metashape Pro, please note that it is a paid photogrammetry software. The software can be downloaded for free, but a license must be purchased for permanent use. Agisoft offers educational licenses which provide discounts; details can be found on the official website under purchasing instructions. Below are some relevant links to help you quickly understand and refer to Metashape:

1. Agisoft website: https://www.agisoft.com/downloads/installer/
2. Purchase license: https://www.agisoft.com/buy/licensing-options/
3. Beginner's tutorial: https://www.agisoft.com/support/tutorials/
4. Forum: https://www.agisoft.com/forum/
5. Knowledge base: https://agisoft.freshdesk.com/support/solutions

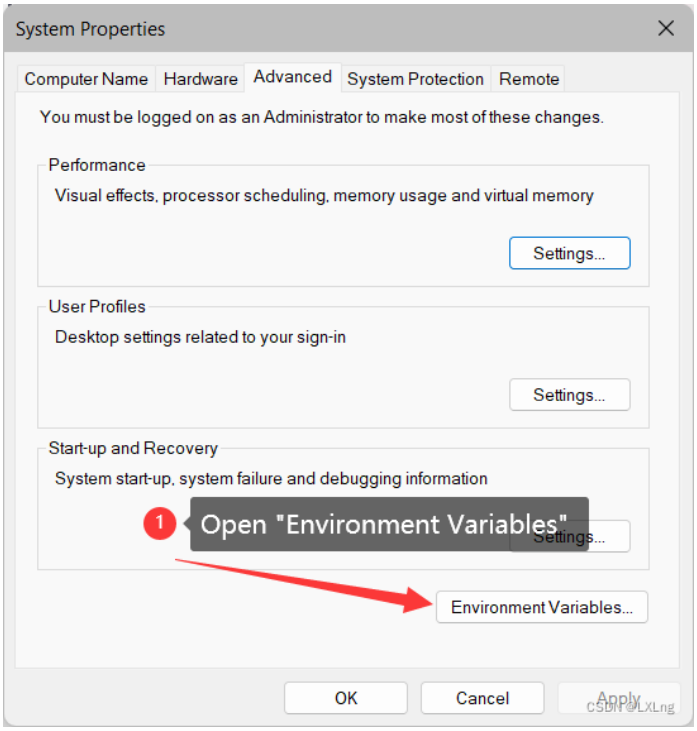
- Visit the Agisoft official website and download the installer for Metashape Pro v1.8.5. Please note that older versions may be removed from the official website due to updates. Users can search for historical versions via Google or contact official technical support after purchasing a license.
- Run the downloaded installer with administrator privileges and follow the

installation wizard's instructions.

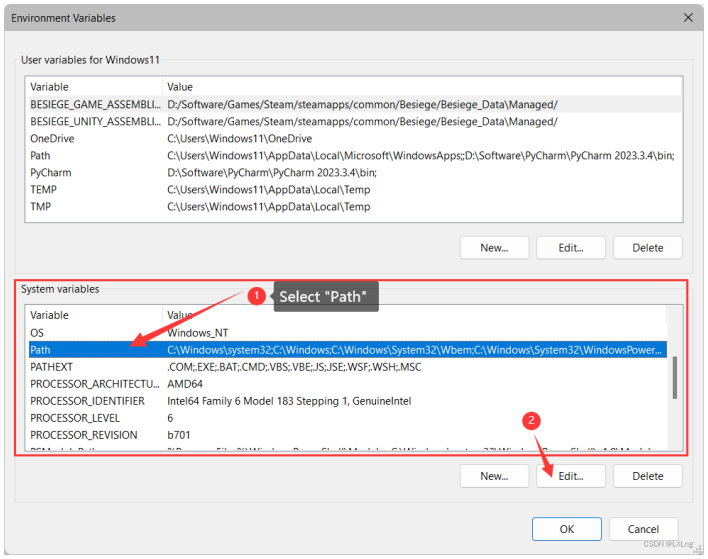
- During the installation process, you may need to accept the license agreement, choose the installation path, and select other options. We recommend keeping all options default except for the installation path.

2.2 Adding Metashape to the Environment Variables

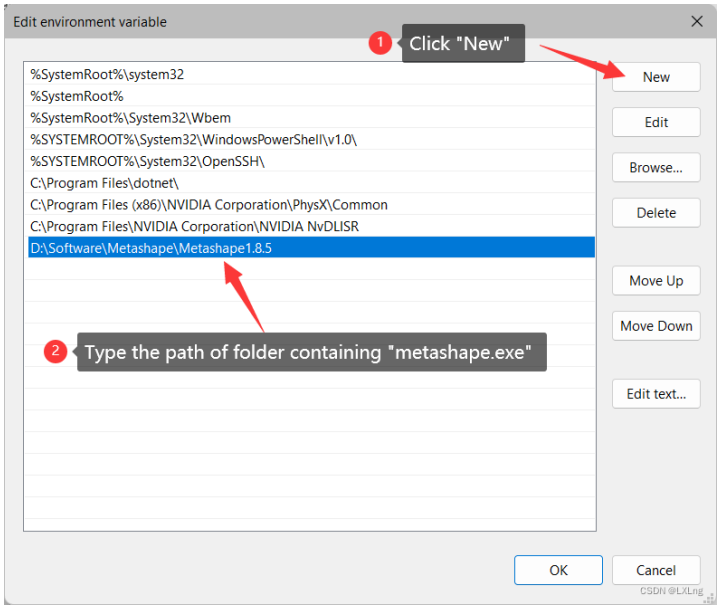
- Right-click on "This PC," select "Properties," and navigate to "Advanced system settings."

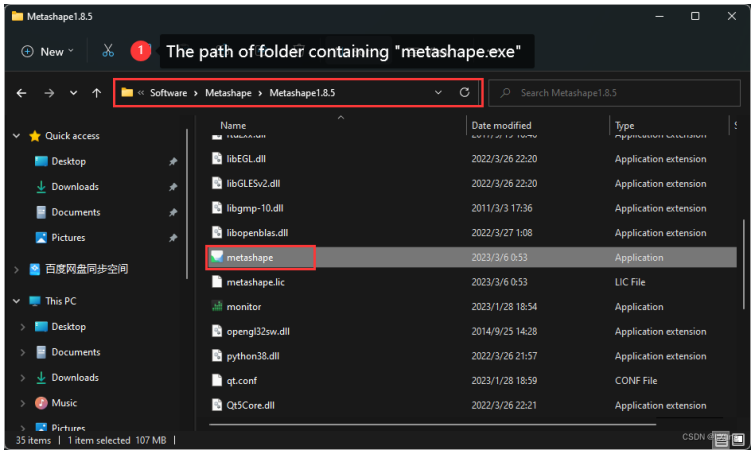


- Open the Environment Variables panel, in the "System variables" section, select "Path," and click "Edit."



- Click "New," then enter the installation path (the directory path containing Metashape.exe). Click "OK" in all panels to complete the environment variable setup.





2.3 Installing Third-Party Python Libraries

Official Instructions for Installing Python Libraries: How to install external Python module to Metashape Professional package. We provide two methods for installing third-party libraries in a specified Python environment on Windows 10 (using numpy as an example):

Method 1: Automatic Installation from Source:

1 Press Win+R, type cmd, and press Enter to open the command prompt (cmd.exe).

2 Type cd /d G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python (replace with your installation path).

3 Type python -m pip install numpy to install the numpy library.

4 If the library is not found, you can specify a source URL. For example, to use the Tsinghua source, type:

python -m pip install numpy -i https://pypi.tuna.tsinghua.edu.cn/simple/.

Method 2: Manual Download of whl Files:

1 Download the appropriate wheel file (e.g., numpy-1.24.2-cp38-cp38-win_amd64.whl) from the website.

2 Press Win+R, type cmd, and press Enter to open the command prompt (cmd.exe).

3 Type cd /d G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python (replace with your installation path).

4 Execute the command python -m pip install <path to wheel file>.

Here is a list of required third-party library functions. These library versions have been tested and verified to be suitable for Metashape Pro v1.8.5.

Libraries Name	Version	Method
numpy	1.23.5	Automatic
scipy	1.10.1	Automatic

pandas	2.0.0	Automatic
networkx	3.1	Automatic
GDAL	GDAL-3.4.3-cp38-cp38-win_amd64.whl	Manual
opencv-python	4.7	Automatic
opencv-contrib-python	4.7	Automatic
matplotlib	3.7.1	Automatic
python-dateutil	2.8.2	Automatic
pytz	2023.3	Automatic

- Begin installing third-party python libraries. GDAL needs to be installed using the Manual method. Download the GDAL whl file by searching on Google or from the 'libs' directory of our Github project. Once you have the GDAL installation package ready, implement these commands below step-by-step in the command prompt (pressing Win+R to open cmd). If you encounter issues related to a corrupted pip during the installation process, you can refer to this troubleshooting solution [1].

```
cd /d G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python
python -m pip install --upgrade pip
python -m pip install numpy==1.23.5
python -m pip install scipy==1.10.1
python -m pip install pandas==2.0.0
python -m pip install matplotlib==3.7.1
python -m pip install networkx==3.1
python -m pip install python-dateutil==2.8.2
python -m pip install pytz==2023.3
python -m pip install opencv-python==4.7.0.72 -i https://pypi.tuna.tsinghua.edu.cn/simple/
python -m pip install opencv-contrib-python==4.7.0.72 -i https://pypi.tuna.tsinghua.edu.cn/simple/
python -m pip install
```

1. Repairing Pip in Metashape's Built-in Python Environment
Press Win+R to open the command prompt. Run the following commands line by line, ensuring to replace <G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python> with your actual path:

```
cd /d <G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python>
python <…CFTM_v1.0\toolbox\get-pip.py>
python -m pip install --upgrade pip
```

After executing these commands, pip should function normally. However, when you reopen Metashape, it will automatically update and restore to its original pip version. If you prefer to prevent this, you can try disabling the automatic update option in Metashape (Tools → Preferences → Miscellaneous → Check for updates on program startup).

```
G:\UAV_SOFTWARE\Metashape\Python3Module\GDAL-3.4.3-cp38-cp38-win_amd64.whl
```

- After installation completion, open command prompt using Win+R. In the command prompt, enter the following commands line by line to view the installed libraries in the current Metashape Python environment. You can cross-reference this with your list of required libraries below, This command will display a list of all libraries installed in your Metashape Python environment.

```
1  cd /d G:\UAV_SOFTWARE\Metashape\Metashape1.8.5\python
2  python -m pip list
```

Libraries Name	Version	Libraries Name	Version
attrs	23.1.0	backcall	0.2.0
click	8.1.3	colorama	0.4.3
ConfigArgParse	1.5.3	contourpy	1.0.7
cycler	0.11.0	dash	2.9.3
dash-core-components	2.0.0	dash-html-components	2.0.0
dash-table	5.0.0	decorator	4.4.2
fastjsonschema	2.16.3	Flask	2.2.3
fonttools	4.39.3	GDAL	3.4.3
importlib-metadata	6.5.0	importlib-resources	5.12.0
ipykernel	5.3.4	ipython	7.16.1
ipython-genutils	0.2.0	ipywidgets	8.0.6
itsdangerous	2.1.2	jedi	0.17.2
Jinja2	3.1.2	jsonschema	4.17.3
jupyter-client	6.1.6	jupyter-core	4.6.3
jupyterlab-widgets	3.0.7	kiwisolver	1.4.4
llvmlite	0.39.1	MarkupSafe	2.1.2
matplotlib	3.7.1	nbformat	5.7.0
networkx	3.1	numba	0.56.4

numpy	1.23.5	open3d	0.17.0
opencv-contrib-python	4.7.0.72	opencv-python	4.7.0.72
packaging	23.1	pandas	2.0.0
parso	0.7.1	pexpect	4.8.0
pickleshare	0.7.5	Pillow	9.5.0
pip	24.0	pkgutil_resolve_name	1.3.10
plotly	5.14.1	prompt-toolkit	3.0.5
ptyprocess	0.6.0	Pygments	2.6.1
pyparsing	3.0.9	pyrsistent	0.19.3
PySide2	5.15.2.1	python-dateutil	2.8.1
pytz	2023.3	pywin32	228
pyzmq	19.0.1	qtconsole	4.7.5
QtPy	1.9.0	scipy	1.10.1
setuptools	49.2.0	shiboken2	5.15.2.1
shiboken2-generator	5.15.2.1	six	1.15.0
tenacity	8.2.2	tornado	6.0.4
traitlets	4.3.3	tzdata	2023.3
wcwidth	0.2.5	Werkzeug	2.2.3
wheel	0.29.0	widgetsnbextension	4.0.7
zipp	3.15.0		

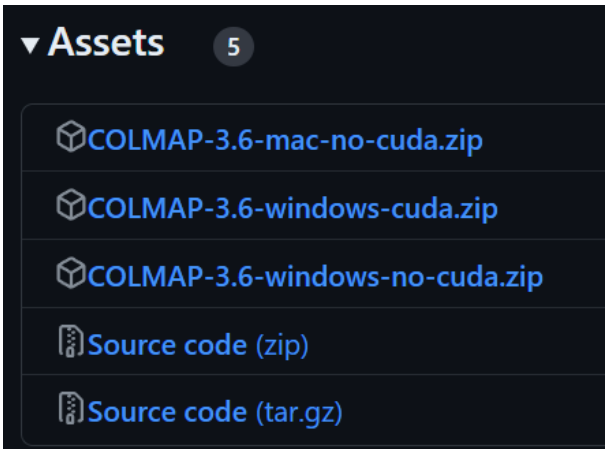
2.4 Installing COLMAP (Optional but Recommended)

Feature point extraction in CFTM v1.0 is implemented using the feature extraction functionality of the open-source third-party library OpenCV (default) or optionally COLMAP, with feature matching analysis leveraging COLMAP's capabilities. If you wish to utilize the faster SIFT-GPU functionality, it is recommended to install COLMAP version 3.6 (compatibility with other COLMAP versions has not been tested). Please ensure compliance with COLMAP's licensing agreement.

Official installation guides for COLMAP: Official website, GitHub Releases.

Other installation tutorials: [Installing and debugging COLMAP on Windows 10](#).

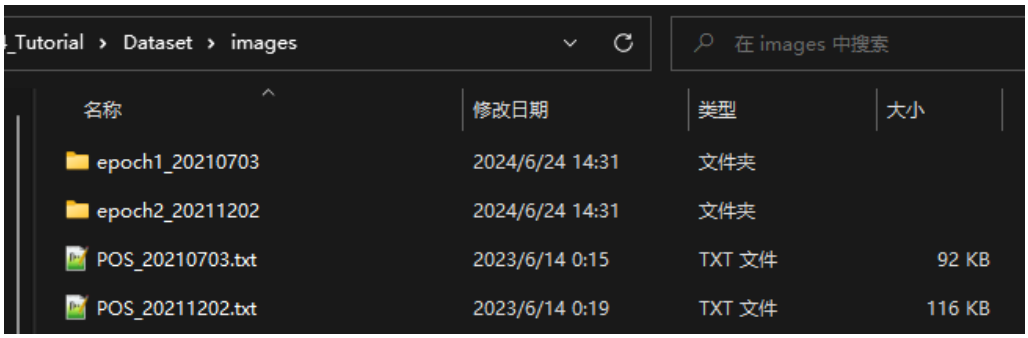
Direct download link for COLMAP 3.6 Release: [COLMAP 3.6 Release](#)



Click on COLMAP-3.6-windows-cuda.zip or COLMAP-3.6-windows-no-cuda.zip (depending on whether your device supports NVIDIA GPU) to download. After downloading, you will receive a compressed file which you can extract. Next, double-click "RUN_TUTORIALS" to test the environment. If there are no issues, COLMAP has been successfully installed. You can then directly double-click "COLMAP" to access the software's graphical interface. Make sure to note the path to COLMAP.bat, as it will be needed for subsequent configurations.

3 Example

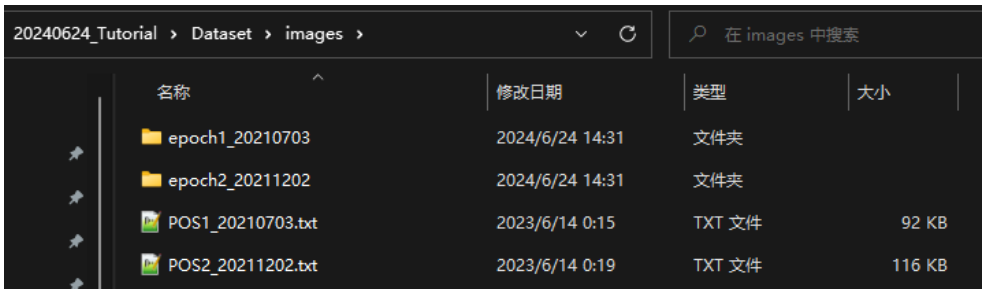
After extracting the Tutorial archive, you will see the files as shown in the image below. Among them, the Dataset folder contains example data, the Example folder contains example results obtained after running the tutorial, which readers can directly view, and the TryHere folder provides project files after co-alignment (equivalent to completing steps 1 to 3), making it easier to experience the CFTM algorithm directly.



Before running the examples in this chapter, ensure that you have completed the installation procedures as described earlier.

3.1 Dataset

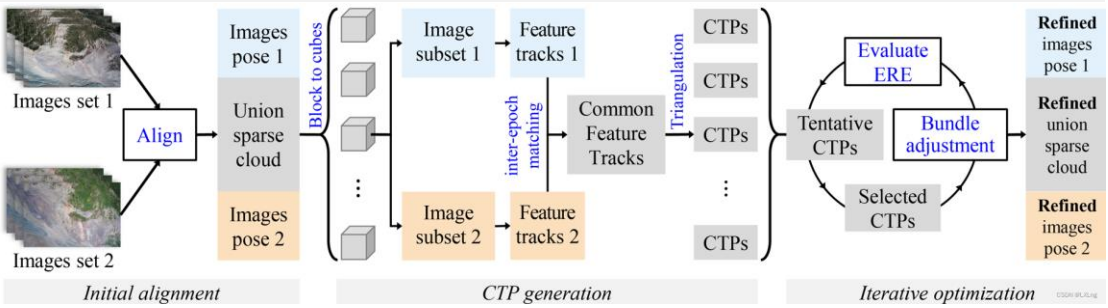
Prepare two sets of images captured at different times in the same surveyed area, labeled as epoch1 and epoch2. When optical aerial survey cameras capture images, the onboard GNSS/IMU positioning system simultaneously acquires the camera's optical center geographic position and camera attitude angles, collectively known as "POS data," labeled as POS1 and POS2. Note that some UAV manufacturers (e.g., DJI) directly embed POS information into image EXIF data, while others (e.g., Feima) store POS information in independent CSV files. Here, we use the latter as an example.



3.2 General Workflow

Here we briefly outline the workflow of the entire method. For detailed specifics, refer to "Section 3 Methodology" in the paper. The simplified workflow is as follows:

1. Acquire datasets of images taken at different times.
2. Roughly co-register images from multiple epochs to obtain approximate image poses and a unified sparse point cloud.
3. Divide the unified sparse point cloud into fixed-size grids.
4. Compute the bounding box for each grid and project it onto corresponding image datasets to extract a subset of feature points using masked polygons.
5. Construct feature trajectories for different epochs within each grid based on the subset of feature points.
6. Match feature trajectories between images from different epochs, identifying nearest matching features as common feature trajectories.
7. Use triangulation to construct common tie points from the identified common feature trajectories.
8. Calculate the Epoch Reprojection Error (ERE) of the common tie points and iterate until convergence to obtain refined image poses after registration.



3.3 Program Settings

To configure the code paths, open the CFTM code folder and locate "config.py". Set the paths in the SETUP section as follows:

1. PATH_CODE = Path to the CFTM code folder;

for example: PATH_CODE = r'D: \CoSfM\Release\CFTM_v1.0'

2 PATH_COLMAP_BAT = Path to COLMAP.bat;

for example: PATH_COLMAP_BAT = r'D:\Software\COLMAP\COLMAP-3.6-windows-cuda\COLMAP.bat'

The following paths are required only if your device has a GPU and you need to enable OpenCV's SIFT-GPU functionality:

3 PATH_CUDA = Path to CUDA installation;

for example: PATH_CUDA = r'C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.8\bin'

4 PATH_OPENCV = Path used for binding CUDA to OpenCV on Windows;

for example: PATH_OPENCV = r'G:\OpenCV\opencv_4_5_0_cuda_11_1_py38\install\x64\vc16\bin'

Next, open the script files in `..\CFTM_v1.0\toolbox`:

1 CheckPoints_Analyse.py

2 CheckPoints_Import.py

3 ICTPs_Analyse.py

4 ICTPs_Delete.py

5 Process_SplitChunk.py

Check the import section of script files list above. Set sys.path.append() to the path of the CFTM code folder. For example:

```
sys.path.append(r'D:\Research\20221223_CoSfM\Release\CFTM_v1.0')
```

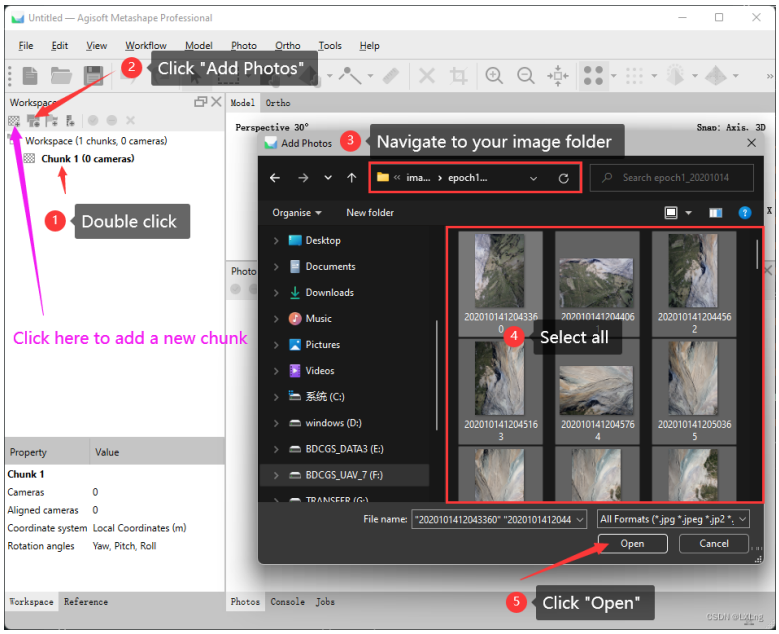
If you are using PyCharm to open the CFTM project, you can use the global replace feature to replace these paths. Additionally, ensure that the Python interpreter in the PyCharm project is set to python.exe under the Metashape installation directory, for example “D:\Software\Metashape\python\python.exe”。

3.4 Running Steps

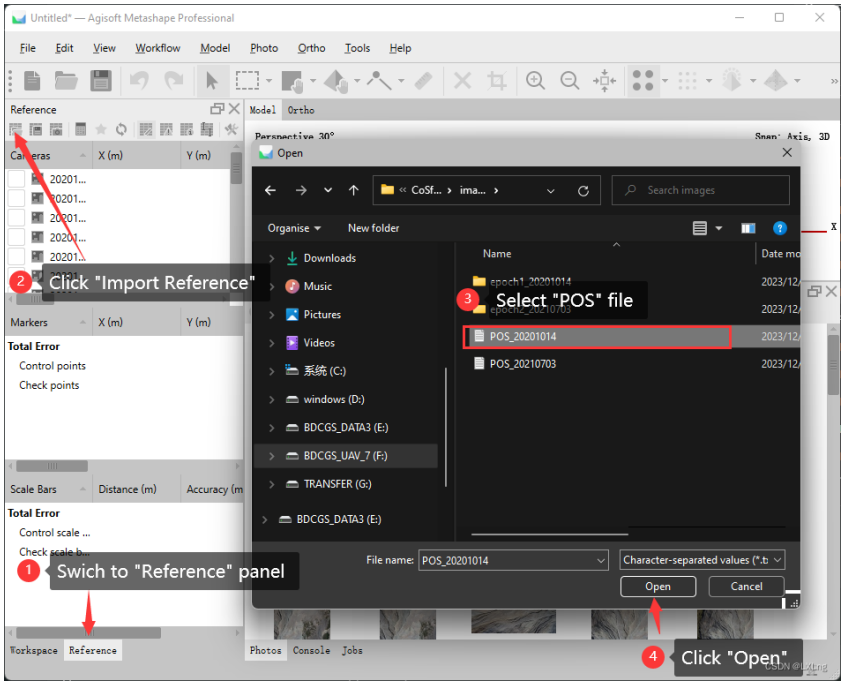
Step 1: Create Project File

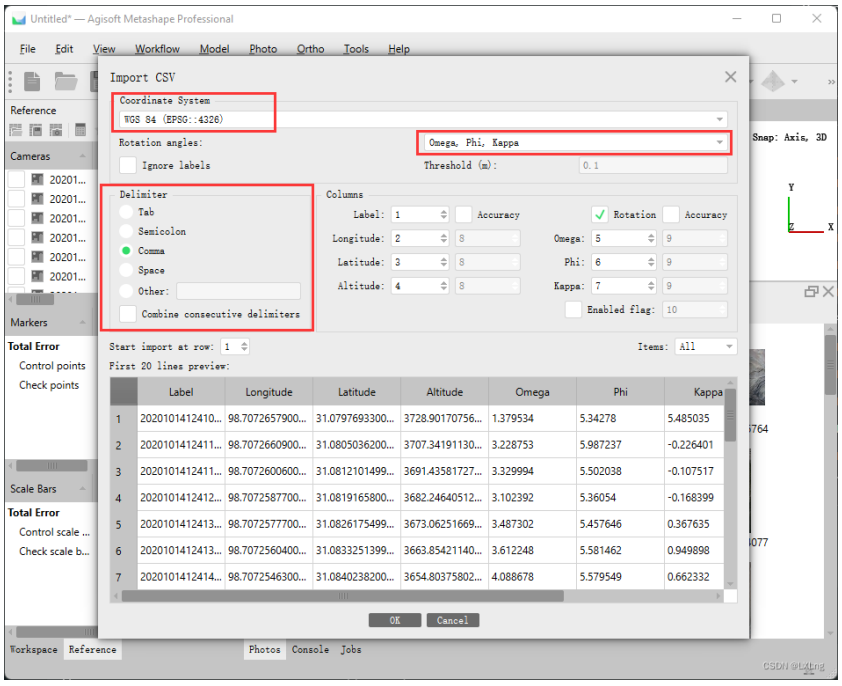
Open Metashape.exe. We need to create a project file and import data from two epochs into the project. Double-click in the workspace to select chunk1, then click the "Add Photos" button. In the file explorer that appears, navigate to ...\Tutorial\Dataset\images\epoch1_20201014, press Ctrl+A to select all photos, click Open, and wait for the photos to load.

For the second epoch data, click the "Add Chunk" button, then double-click chunk 2 to select it, and add the second epoch photos using the same method. The file path is ...\Tutorial\Dataset\images\epoch2_20210703.

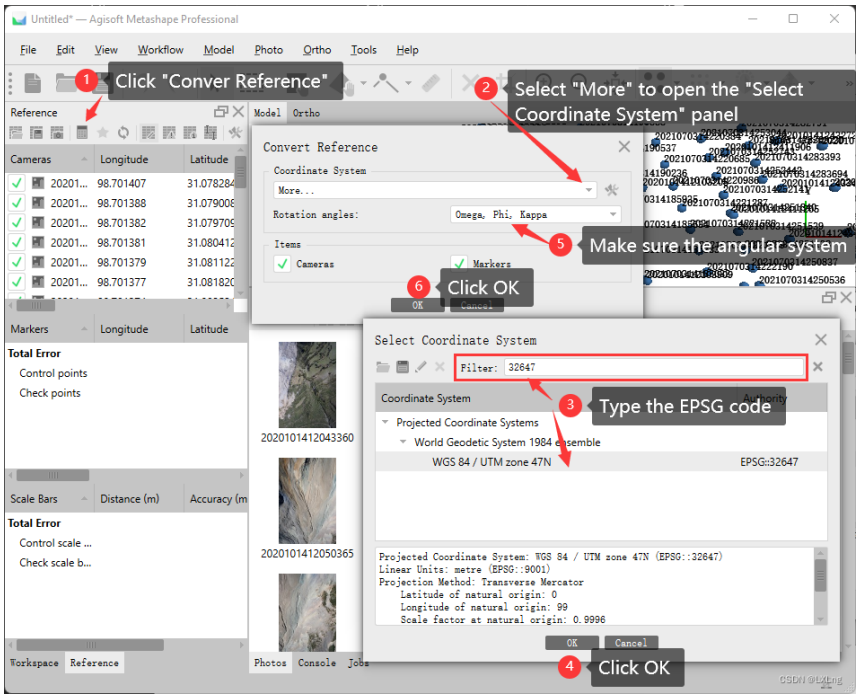


After importing photos, import POS data for each epoch. Using Chunk1 as an example, in the workspace, double-click to select Chunk1, then switch the workspace panel to the Reference pane. Click "Import Reference", select the POS data for the corresponding date, and follow the instructions in the dialog box to select coordinate system, orientation system, delimiter, tags, and precision.

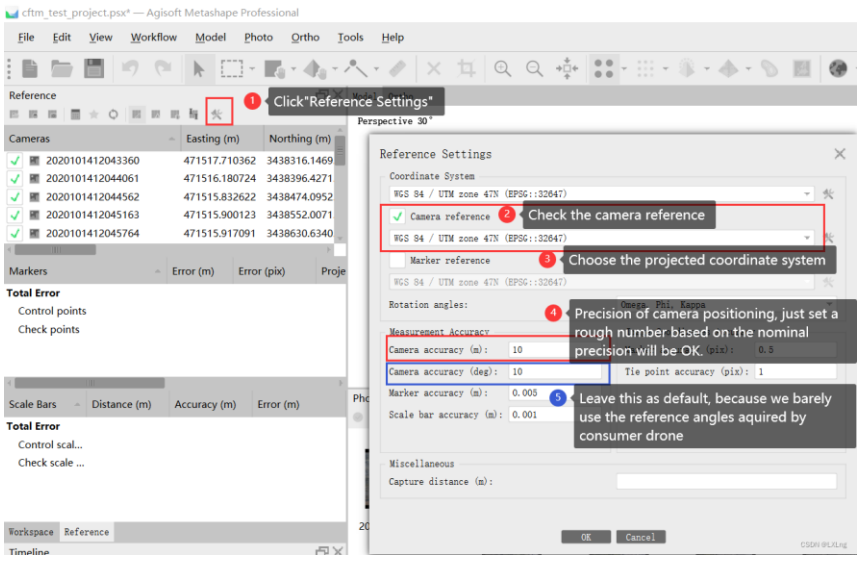




For accurate computation, set the chunk's coordinate system to a projection coordinate system. Click "Convert Reference Coordinate System" as shown, select the target coordinate system in the dialog box. Here, based on longitude conversion, select WGS 84/UTM zone 47N projection coordinate system. Keep other settings as default.

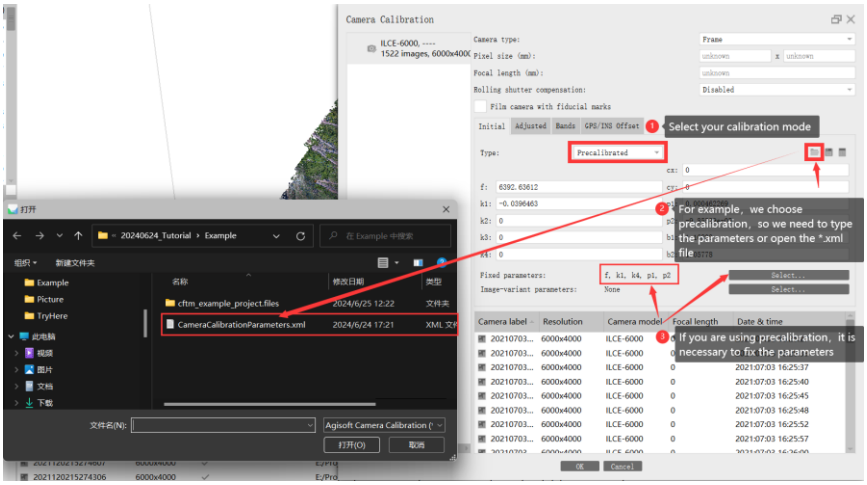


Set reference and accuracy parameters. As shown in the figure below, click on "Reference Settings", check "Camera Reference", and select WGS 84/UTM zone 47N as the reference coordinate system. Set the camera accuracy to 10, which is the nominal accuracy of UAV-mounted GPS. Keep other values as default.



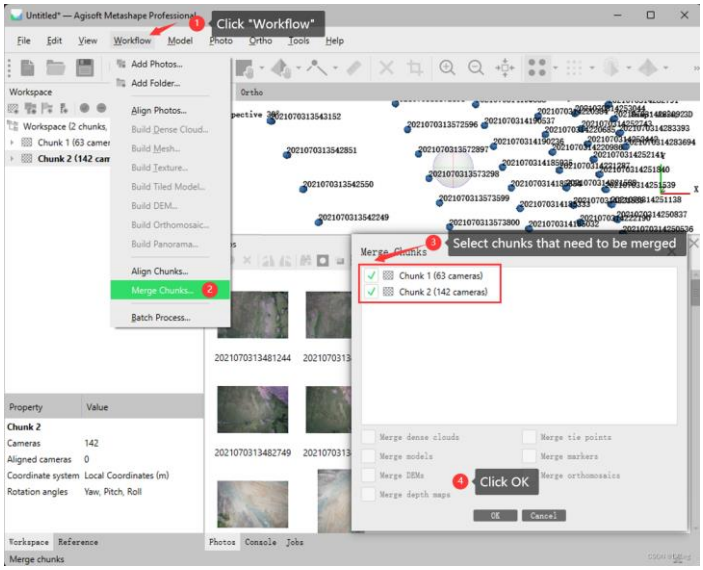
From the main menu, select Tools, then Camera Calibration.... In the dialog that appears, configure the camera calibration parameters. First, choose the calibration type. If selecting self-calibration, only fix the parameters that need to be fixed (i.e., keep parameters set to 0 to indicate they are not calibrated). If choosing pre-calibration (as in this example), manually enter parameters or open a parameter file (located

in ..\Tutorial\Example\), and fix the corresponding parameters to prevent overfitting.

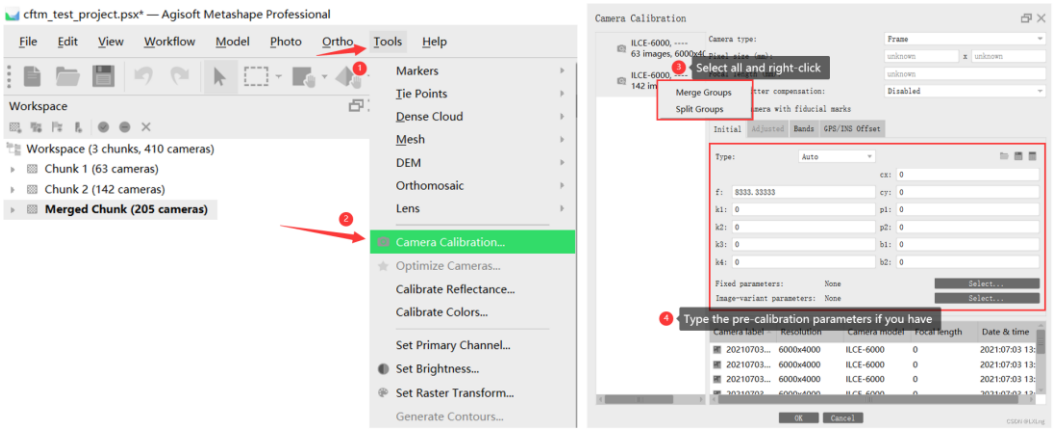


By now, Chunk1's data import is complete. Repeat these steps to complete Chunk2's data import. Once finished, save the project file to ...\Tutorial\Example (this is the workspace folder for this example; subsequent CFTM process files will be stored in a folder adjacent to the project file). Name the file "cftm_example_project.psx".

If your single epoch data includes multiple flights that need to be added separately, you can first add each flight's data to its own chunk, then merge them into one chunk. To do this, click on Workflow in the main menu, then select Merge Chunks.... In the dialog that appears, check the chunks to merge, then click OK.



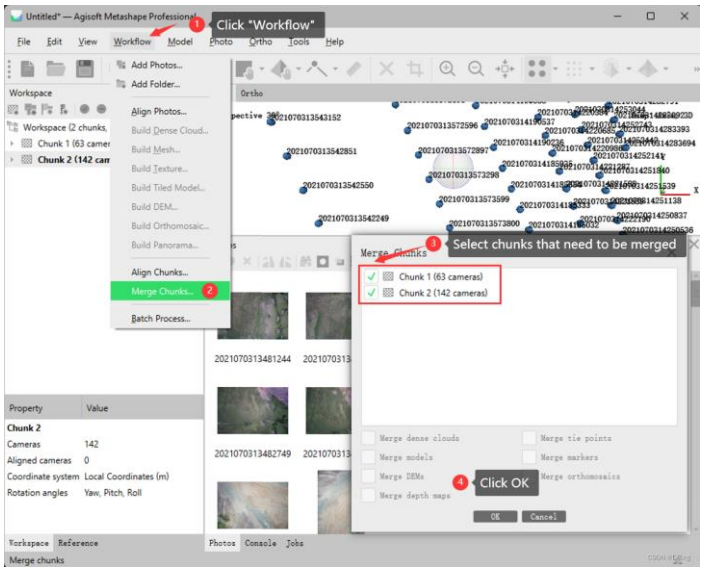
Note: When merging multiple chunks, there will be multiple camera models in Tools → Camera Calibration.... You can select them all, then right-click and choose Merge Groups to ensure that the merged chunk corresponds to a single camera model.



Step 2: Coarse Registration

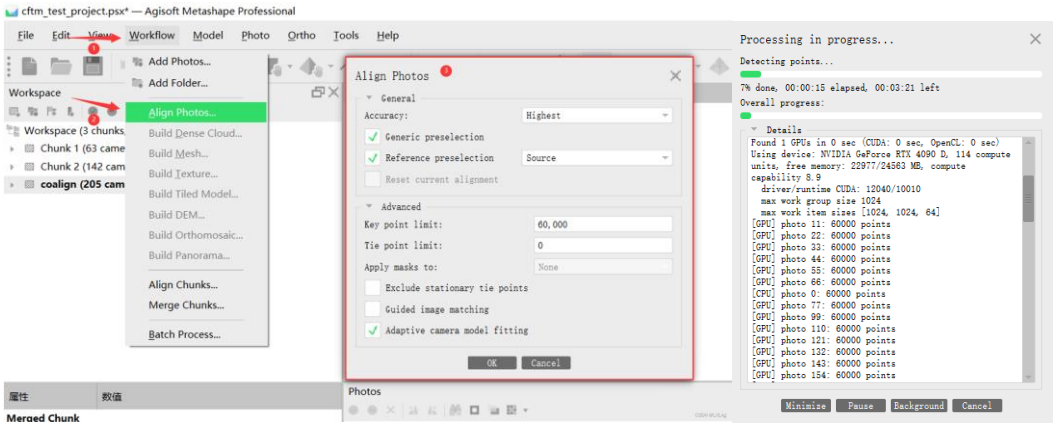
To perform registration using the Co-alignment algorithm, merge the two chunks into one before executing the SfM algorithm. Here are the specific steps:

- From the main menu, click on Workflow, then select Merge Chunks....
- In the dialog that appears, check Chunk1 and Chunk2, then click OK. This creates a merged chunk named "Merge" (which can be renamed to "coalign").



Next, as shown in the figure below:

- In the main menu under Workflow, select Align Photos....
- In the dialog that appears, configure the parameters as shown in the figure: set accuracy to highest, enable preselection, limit key points to 60,000, no tie points limit, and enable adaptive camera model fitting. Click OK to start aligning photos.



After completion, you will have successfully performed coarse registration using co-alignment, resulting in a merged sparse point cloud and photo poses.

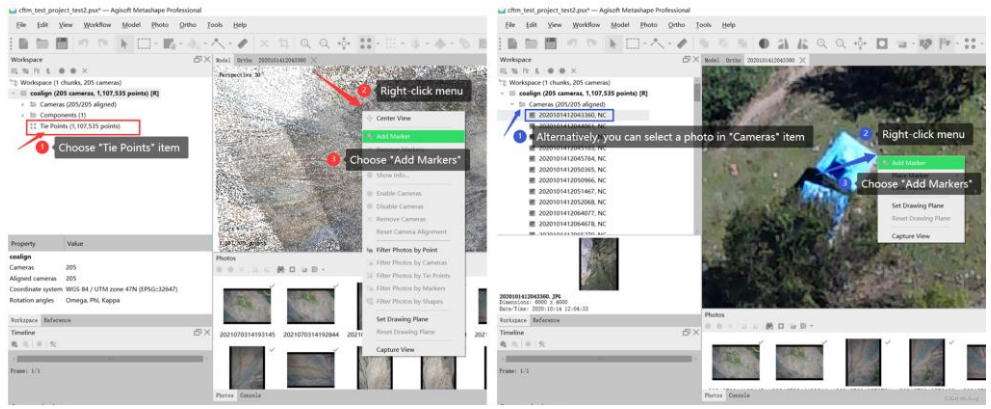
Step 3: Creating Checkpoints (Optional)

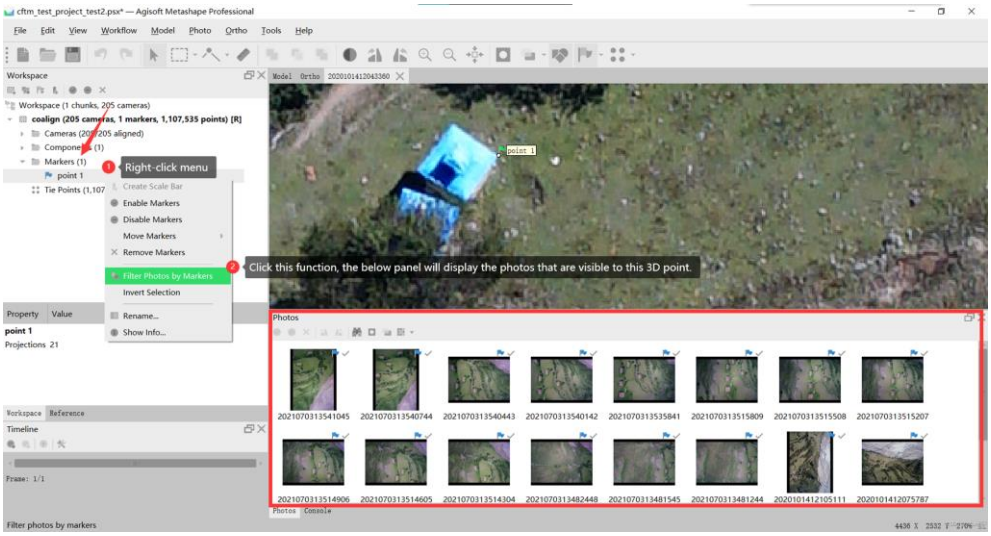
Identify stable ground features in the sparse point cloud or photo set, right-click on the feature's location, and select "Add Markers". The added markers can be found under the "Markers" group in the left workspace panel.

For precisely marking Checkpoints:

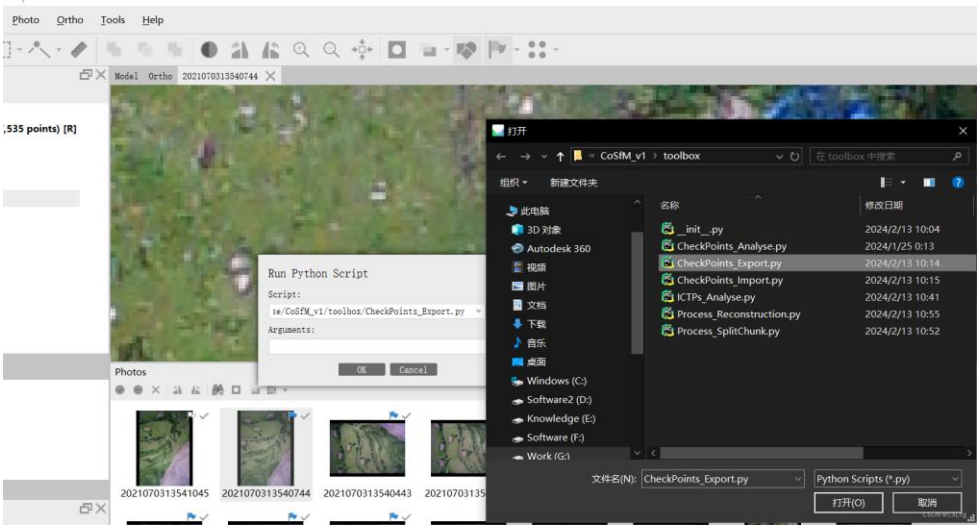
- Select the newly added marker (e.g., named point 1), right-click, and choose "Filter Photos by Markers". This action displays the corresponding visible photos in the Photos panel.
- Double-click on the first photo, then in the enlarged view, drag the tag to the accurate position. The flag color changes from blue to green (in Metashape, green and blue respectively indicate enabled and disabled image points during optimization when used as control points; however, as checkpoints, the image point remains active unless removed using the right-click menu "Remove Marker").
- Proceed to mark the next photo, aiming to evenly mark a roughly equal number of photos across both epochs.

Repeat these steps to create checkpoints, aiming for at least 5 checkpoints evenly distributed in stable areas.





Press Ctrl+R to open the "Run Python Script" dialog. Select the program located at ...\toolbox\CheckPoints_Export.py in the CFTM code package. Click OK. The checkpoint data file will be exported in the same directory as the project file (e.g., named cftm_test_project_CPsdatabase.txt).



After exporting, remove the newly created checkpoint markers. In Metashape, during bundle adjustment, markers are considered tie points for calculation. To prevent them from affecting automatic registration results, right-click on each checkpoint marker and select "Remove Marker". Don't worry, the checkpoint data file has been saved, and you can use the program ...\toolbox\CheckPoints_Import.py in the CFTM code package to import these points back into the project later.

Step 4: Configure Parameters

You need to configure parameters in the "SETUP" section of Run.py. Most parameters in Run.py can remain default, while the CFTM_args parameters need to be set based on your actual situation. Key parameters to set in CFTM_args include: **gird_size_1, gird_size_2, num_nominated, num_selected, num_max_iterations, pool_size , threshold_Distance.**

1. **grid_size_1:** This parameter divides the initial sparse point cloud into squares of this size. **grid_size_1** is set to constrain the matching range and accelerate matching efficiency, so it should not be too large and can be kept default. **grid_size_2** is set to ensure spatial uniformity but should prevent the algorithm from selecting low-quality points just to satisfy spatial uniformity. It's recommended to set **grid_size_2** larger than **grid_size_1**.
2. **num_nominated, num_selected:** The algorithm initially selects **num_nominated** × **m** × **n** CTPs with the smallest ERE as candidates, then tries to select **num_selected** CTPs with the smallest ERE from each grid from the candidate set. Therefore, the final CTPs obtained will be fewer than **num_selected** × **m** × **n**.
3. **num_max_iterations:** This is the maximum number of iterations. If the coarse registration is relatively good, the number of iterations may be in single digits. Generally, it's recommended to set it to 30.
4. **pool_size:** Set this according to the number of CPU cores. For example, if you have an Intel i7-10875H with 16 cores, you can set **pool_size** to a maximum of

16. If you encounter memory issues or block other programs, reduce the value of **pool_size**.

5 **threshold_Distance:** This parameter defines the maximum estimated initial offset for matching pairs of cross-temporal feature trajectories. If the distance between corresponding 3D points of a pair of cross-temporal feature trajectories exceeds this threshold, the algorithm will ignore them. This threshold should be greater than the accuracy of coarse registration based on our experience.

```
# Open a project file
workspace_path = r"...\Tutorial\Example"
project_name = "cftm_example_project.psx"
images_path = r'...\Tutorial\Dataset\images'
check_points_path =
r"...\Tutorial\Dataset\CPsDatabase.txt"


# please make sure your major version is:
compatible_major_version = "1.8"


# Process arguments
process_args = {
    # Choose the processing options for CFTM.
    "process_coalign": False,
    "process_feature_extraction": True,
    "process_CTPs_generation": True,
    "process_iterative_optimization": True,
    "process_analysis": True,
    "process_analysis_matches": False,
    "process_compare_matches": False,
```

```
    # Choose the processing options for reconstruction.
    "process_reconstruction": True,
    "process_build_DSM": True,
    "process_build_DOM": False,
    "process_build_TiledModel": False,
    "process_export_DSM": True,
    "process_export_DOM": False,
    "process_export_TiledModel": False,
    "process_white_balance": False,

    "save_project_each_step": True,
}

# Select the mode for dividing epochs.
# If it's "DATE", it will divide them by day based on
the dates of the photos.
# If it's "FOLDER", it will divide them based on the
folders where the photos are stored.
epoch_mode = 'DATE'  # or 'FOLDER'

# Which feature extraction mode do you want to use?
# 0 : use OpenCV to extract SIFT feature without cuda
(default but slow);
# 1 : use OpenCV to extract SIFT-GPU feature with cuda
(need additional manual compilation);
# 2 : use COLMAP to extract SIFT-GPU feature
(recommended).
```

```
feature_extraction_mode = 2


# Whether to continue the run from the last breakpoint?
begin_from_breakpoint = True


# Specify which chunk to process; leave empty will
automatically select the chunk which has maximum images.
coalign_chunk_name = ''


# Generate CTPs for all grids or only for those grid
without CTPs.
for_all_grids = True


# If CTPs are generated for all grids, do you what to
reuse the CTPs generated by co-alignment?
reuse_ICTPs = True


# Whether to use *.shp to mask the unstable areas?
unstable_areas_mask_path = ''


# Common Feature Track Matching
CFTM_args = {
    "gird_size_1": 32,
    "gird_size_2": 64,
    "scoring": [0, 0, 0, 0, 0, 0, 2, 4, 8, 16],
    "num_nominated": 10,
    "num_selected": 5,
    "num_max_iterations": 20,
```

```
    "pool_size": 16, # the  number should in line with
your CPU cores
    "criterias_1": [2, 3],
    "criterias_2": [1, 3],
    "ratio_inlier_init": 0.25,
    "confidence": 0.9,
    "threshold_RepError": 0.5,
    "threshold_Distance": 3,
    "skip_edge": False,
    "threshold_Termination": 0.001,
    "num_inertia": 3,
}

# Make sure the coordinates of the chunk are in the
projected coordinate system
reference_args = {
    "PJCS_EPSG": 32647,
    "camera_accuracy_m": 2.0,
    "camera_accuracy_deg": 10.0,
    "marker_accuracy_m": 20.0,
    "marker_accuracy_pix": 0.5,
    "tiepoint_accuracy_pix": 1,
    "camera_reference_enabled": True,
    "marker_reference_enabled": False,
    "rotation_system": 'opk'
}

# Bundle Adjustment Arguments
```

```
bundle_adjustment_args = {
    "fit_f": True,
    "fit_cx": True, "fit_cy": True,
    "fit_b1": False, "fit_b2": False,
    "fit_k1": True, "fit_k2": True,
    "fit_k3": True, "fit_k4": False,
    "fit_p1": True, "fit_p2": True,
    "fit_corrections": False,
    "adaptive_fitting": True,
    "tiepoint_covariance": True
}

# Photogrammetry arguments
photogrammetry_args = {
    # Align photos
    "quality_SPC": 1,  # 1(UltraHigh), 2(High), 4(Medium)
    "keypoint_limit": 60000,
    "keypoint_limit_Per_Megapixel": 1000,
    "tiepoint_limit": 0,

    # Error reduction
    "reprojection_error": 0.3,
    "reconstruction_uncertainty": 15,
    "projection_accuracy": 5,
    "max_ratio_remove_points": 0.3,

    # Dense matching
    "quality_DPC": 2,
```

```
    "dense_cloud_FileterMode": 'Mild',


    # Build tiled model
    "tiled_model_TiledModelFormat": 'Cesium',
    "tiled_model_pixel_size": 0.3,  # metres
    "tiled_model_face_count": 20000,
    "tiled_model_ModelFormat": 'None',
    "tiled_model_ImageFormat": 'JPEG',
}
```

Step 6: Running the Script

Press Win+R on your keyboard to open cmd.exe. Enter the following command to run the script. Make sure to replace the path with your actual Run.py path. Press Enter to start running CFTM.

```
1    metashape.exe -r D: \CoSfM\Release\CFTM_v1.0\Run.py
```

CFTM supports checkpoint recording. In case of unexpected situations, after correction, you can use the above command to continue running CFTM, which will resume from the last checkpoint. You can also manually control each stage's status by modifying the "NO" and "DONE" values in the ...Tutorial\Example\cftm\state.json file. If set to "NO", that part will continue to run; if set to "DONE", that part will be skipped.

Example of ...Tutorial\Example\cftm\state.json file:

```
{
    "coalign": "NO",
    "CFTM_creat_task": "DONE",
    "feature_extraction": "DONE",
    "CFTM_run_task": "DONE",
    "IO_mergeCTP": "DONE",
    "IO_iterater": "DONE",
    "IO_implementCTPs": "DONE",
```

```
    "analysis_CTPs": "DONE",
    "analysis_CPs": "DONE",
    "analysis_matches": "NO",
    "compare_matches": "NO",
    "reconstruction": "NO",
    "finished_grids": [],
    "finished_iterations": []
}
```


If you want to monitor the progress during the parallel generation of CTPs, you can keep an eye on the value printed as "task number" in the command line. Additionally, you can check the number of files in the `..\cftm\StateCFTM` folder, where each file contains the processing result information for the corresponding grid.

Step 7: Evaluation Report

After CFTM finishes running, you can find all process files and results in the "...\Tutorial\Example\cftm" folder located in the same directory as cftm_example_project.psx.

In the "...\cftm\Reports" folder, you will find generated GCTPs (Generalized Correspondence Tie Points) and their quality metrics. Files like "_GCTPsQua.txt" list the data of GCTPs, while "_GCTPsQuaSta.txt" provides statistical metrics based on GCTPs data.

If you created checkpoints earlier and configured the checkpoint file path in Run.py, you can also find CPs (Check Points) quality metrics in the "...\cftm\Reports" folder, formatted similarly to GCTPs.

4 Example Results

5 Analysis

5.1 Analysis Matches in CFTM

If you set process_analysis_matches to True in Run.py, CFTM will generate a file named MatchesReport_CFTM.txt in the ../cftm/Reports folder.

Below is the structure of the file along with explanations:

```
1 [Epoch1_PairNum, Epoch2_PairNum, Epoch1_MatchNum, Epoch2_MatchNum] # Number of image pairs in epoch e, number of matches in epoch e
2 [Epoch1_PairMatcNum_AVG, Epoch2_PairMatcNum_AVG] # Average number of matches per image pair in epoch e
3 [Epoch1_PairInlierMatcRatio_AVG, Epoch2_PairInlierMatcRatio_AVG] # Average number of effective matches per image pair in epoch e
4 [Common_PairNum, Common_MatchNum1, Common_MatchNum2, Common_MatchPortion, Common_PairMatcNum_AVG] # Number of image pairs across epochs, number of matches across epochs, proportion of matches across epochs, average number of correct matches per image pair across epochs
5 [Epoch1_TrackNum, Epoch2_TrackNum, Epoch1_InlierTrackNum, Epoch2_InlierTrackNum] # Number of feature tracks in epoch e, number of valid feature tracks in epoch e
6 [Common_TrackNum, Common_TrackCrosMatcNum_AVG] # Number of feature tracks across epochs, average number of matches per feature track across epochs
```

5.2 Compare Matches before and after CFTM

If you set process_compare_matches to True in Run.py, CFTM will generate a file named MatchesReport_Coalign.txt in the ../cftm/Reports folder. Below is the structure of the file along with explanations:

```
1 [Epoch1_PairNum, Epoch2_PairNum, Common_PairNum] # Number of image pairs
2 [Epoch1_MatchNum, Epoch2_MatchNum, Common_MatchNum] # Number of matches
3 [Epoch1_PairMatchNum_AVG, Epoch2_PairMatchNum_AVG, Common_PairMatchNum_AVG] # Average number of matches per image pair
4 [Epoch1_PairInlierMatchNum_AVG, Epoch2_PairInlierMatchNum_AVG, Common_PairInlierMatchNum_AVG] # Average number of effective matches per image pair
5 [Epoch1_PairInlierMatchRatio_WAVG, Epoch2_PairInlierMatchRatio_WAVG, Common_PairInlierMatchRatio_WAVG] # Weighted average number of effective matches per image pair
6 [Epoch1_PairInlierMatchRatio_AVG, Epoch2_PairInlierMatchRatio_AVG, Common_PairInlierMatchRatio_AVG] # Average effective matching ratio per image pair (ignoring pairs with a matching ratio of 0)
7 [Epoch1_MatchPortion, Epoch2_MatchPortion, Common_MatchPortion] # Proportion of matches in all matches for epoch 1, proportion of matches in all matches for epoch 2, proportion of matches in all matches across epochs
8 [Epoch1_InlierMatchPortion, Epoch2_InlierMatchPortion, Common_InlierMatchPortion] # Average effective matching ratio per image pair
9 [Epoch1_TrackNum, Epoch2_TrackNum, Common_TrackNum] # Number
```

```
of feature tracks
[Epoch1_ValidTrackNum, Epoch2_ValidTrackNum, Common_ValidTrackNum]  # Number of valid feature tracks
[Epoch1_ERValidTrackNum, Epoch2_ERValidTrackNum, Common_ERValidTrackNum]  # Number of valid feature tracks remaining after error removal
[Epoch1_ValidTrackPortion, Epoch2_ValidTrackPortion, Common_ValidTrackPortion]  # Effective rate of feature tracks
[Epoch1_ERValidTrackPortion, Epoch2_ERValidTrackPortion, Common_ERValidTrackPortion]  # Effective rate of feature tracks after error removal
```

6 Error Messages

ERROR: Database: {colmap_database_path} is not found!

SOLUTION: If the COLMAP database is not found, it means that during the "Compare Matches" step, the COLMAP files created during Feature Extraction were not found. It's possible that you deleted or moved these files during the CFTM process. You can fix this by modifying "…/cftm/state.json" and changing "feature_extraction": "DONE" to "feature_extraction": "NO", then rerunning CFTM. Alternatively, you can set "process_compare_matches" to False.

ERROR: this script could only deal with pairwise co-align!

SOLUTION: CFTM currently only supports co-registration between two epochs of data. Please ensure that the input data comes from two epochs. Regarding the epoch mode (variable "epoch_mode"), if you choose "DATE", check if the dates of the images in each epoch are from the same day. If you choose "FOLDER", ensure that there are exactly two subfolders (corresponding to each epoch) under the "images_path" folder.

ERROR: COLMAPindex should be 'identifier' or 'image_path'!

SOLUTION: Please confirm whether the image paths stored in Metashape are consistent with those stored in COLMAP. This usually happens when the drive letter changes after reconnecting the hard disk.

7 Varibles Structure

- Cameras = {camera_id:Camera,...}
  - camera_id = int
  - Camera = [transform, reference, covariance, sensors_id, state, path, identifier]
    - transform= [camera_transform, camera_loc, camera_rot, camera_transform_PJCS]
    - reference = [camera_loc_Ref,location_accuracy,camera_rot_Ref,rotation_accuracy]
    - covariance = [camera_location_covariance, camera_rotation_covariance]
    - states = [label,camera_loc_Ref,camera_rot_Ref, selected, camera_orientation]
- Points = {point_id:Point,...}
  - point_id = int
  - Point = [Coordinates, Covariance, StandardDeviation, Quality, selected, valid, Track_id]
    - Coordinates=[[X, Y, Z],[X, Y, Z]] # in CLCS & PJCS respectively
    - Covariance = matrix # with 3x3 covariances in mm
    - StandardDeviation = [stdX, stdY, stdZ] # in mm
    - Quality = [RE, RU, PA, IC]
- Tracks = [Track,...] # index=Track_id, length=len(chunk.point_cloud.Tracks)
  - Track = [view,...] # index=view_id, length=number of views
    - view = [camera_id, project_info]
      - project_info = [projection.coord[0], projection.coord[1], projection.size, projection.track_id]
- Sensors = {sensor_id:Sensor,...}
  - sensor_id = int
  - Sensor = [calibration_params, calibration_matrix]
    - calibration_params = [f, cx, cy, b1, b2, k1, k2, k3, k4, p1, p2, p3, p4, height, width]
    - calibration_matrix = matrix # nxn covariance matrix, where n is the number of estimating parameters.
- Projections = {identifier:Camera_Projections,...}
  - Camera_Projections= [projection_info,...]
    - projection_info = [u, v, size, Track_id]
- CoordinateTransform = [M, LSE, T, R, S]
  - M = ndarray # 4x4 ndarray
  - LSE = ndarray # 4x4 ndarray
  - T = ndarray # 4x4 ndarray
  - R = ndarray # 3x3 ndarray
  - S = float
- CoordinateAttribute = [crs_name, crs_PJCS_wkt, crs_GCCS_wkt, crs_GDCS_wkt]
  - crs_PJCS_wkt = string # Projection coordinate system definition in WKT format.
  - crs_GCCS_wkt = string # Geocentric coordinate system definition in WKT format.
  - crs_GDCS_wkt = string # Geodetic coordinate system definition in WKT format.
- identifier = int
  - unique number for image, consist of photographing time and image name (e.g. an image taking at '2021:04:04 13:40:13' with name 'DSC00009.JPG', then its identifier is 2021040413401309).
- cameraPaths = {camera.photo.path:identifier,...}
  - camera.photo.path = string
  - identifier = int
- camera_ids = {identifier:camera_id,...}
  - identifier = int
  - camera_id = int
- point_ids = {Track_id :point_id,...} # if the track faild in built a point, the point_id will be -1
  - Track_id = int
  - point_id = int
- validTracks = [Track,...] # index=validTrack_id
  - Track = [view,...] # index=view_id, length=number of views
    - view = [camera_id, project_info]
      - project_info = [projection.coord[0], projection.coord[1], projection.size, projection.track_id]
- validTracks_ids = {validTrack_id:Track_id,...}
  - validTrack_id= int
  - Track_id= int
- pair_ids = {pair_id:match_state,...}
  - pair_id = (int, int)

- match_state = [all, valid, invalid]
    - all = int # the total matches number
    - valid = int # the valid matches number
    - invalid = int # the invalid matches number
- match_ids = {pair_id:[valid_matches, invalid_matches],...}
    - pair_id = (int, int) # (camera_id1, camera_id2)
    - valid_matches = [Match,...]
    - invalid_matches = [Match,...]
        - Match = ([proj_u,proj_v,size,track_id],[proj_u,proj_v,size,track_id])
- chunkKeypoints = {identifier:Keypoints,...}
    - identifier = int
    - Keypoints = ndarray # dtype=float32, shape=(numKeypoints, 6)
        - Keypoint = [u,v,affinity1, affinity2, affinity3, affinity4]
- chunkDescriptors = {identifier:Descriptors,...}
    - identifier= int
    - Descriptors = ndarray # dtype=uint8, shape=(numKeypoints, 128)
- Boundary_CLCS = [boundary_X_MAX,boundary_X_MIN,boundary_Y_MAX,boundary_Y_MIN]
    - the boundary of sparse point cloud in XY plane, in meter
    - boundary_X_MAX = float
    - boundary_X_MIN = float
    - boundary_Y_MAX = float
    - boundary_Y_MIN = float
- Boundary_PJCS = [boundary_X_MAX,boundary_X_MIN,boundary_Y_MAX,boundary_Y_MIN]
    - the boundary of sparse point cloud in XY plane, in meter
    - boundary_X_MAX = float
    - boundary_X_MIN = float
    - boundary_Y_MAX = float
    - boundary_Y_MIN = float
- epochs = [datelist,...]
    - datelist = [date,...] # a datalist contains the date for an epoch.
    - date = 'YYYY:MM:DD'
    - for example: epochs = [['2020:10:14'], ['2021:07:03'], ['2021:12:03'], ['2021:12:06']]
- epochNum = [ei,...] # index=epoch_id, length=epochs number
    - ei = int # the number of images from the specified epoch
    - for example: epochNum= [e1, e2, e3, e4]
- signal = [si, ...]
    - si = int # Using 0 or 1 to represent whether there is image from the specified epoch
    - for example: signal = [0, 0, 1, 1]
- CamerasEpoch = [epoch_id,...] # index=camera_id, length=cameras number
    - epoch_id = int # assign which epochs dose camera belongs to
- TracksEpoch = [epochNum,...] # index=Track_id, length=Tracks number
    - epochNum = [e1, e2, e3, e4] # the counts for images of a certain epochs in a Track.
- EpochTracks = [Track,...] # index=EpochTracks_id (accompanied by EpochTracks_ids)
    - extract the Track that contains images from different epochs based on the TracksEpoch
    - Track = [view,...] # index=view_id, length=number of views
        - view = [camera_id, project_info]
            - project_info = [projection.coord[0], projection.coord[1], projection.size，projection.track_id]
- EpochTracks_ids = [Track_id,...] # index=EpochTracks_id
    - Since the EpochTracks extract from the Track_ids, the index of EpochTracks is different from Track_ids. So the original Track_id index of it is recorded in this associated varible.
- CommonTiePoint_IdList = [point_id, ...] # index=CommonTiePoint_id, length=CommonTiePoints number
    - analysis Common Tie Points (CTPs) based on TracksEpoch and extract the index of epoch tie points to this list.
    - point_id = int
- CommonTiePoint_Signal = [signal,...] # index=CommonTiePoint_id, length=CommonTiePoints number
    - analysis common tie points(CTPS) based on TracksEpoch and assign the signal for each common tie point.
    - signal = list
- CTPsCoord_CLCS = [Coordinates,...] # length=CommonTiePoints number
    - Coordinates = [X, Y, Z] # in CLCS
- CTPsCoord_PJCS = [Coordinates,...] # length=CommonTiePoints number
    - Coordinates = [X, Y, Z] # in PJCS
- EpochPairs
- EpochMatches

- PointsGrid = {grid_id: Point_IdList}
  - point_id list of each cell
  - grid_id = (r, c)
  - Point_IdList = [point_id,...]
- CTPsGrid = matrix # shape=(r, c), index=r, c
  - number of CTP in each cell
  - CTP number = int
- PointsGrid_NoCTPs = {grid_id: Point_IdList}
  - point_id list of the cell that contains no CTPs
  - grid_id = (r, c)
  - Point_IdList = [point_id,...]
- GridDots
- Cubes = { grid_id: conners}
  - grid_id = (r, c)
  - conners= [topLeft, topRight, leftBottom, rightBottom, upper, lower]
- CubesPolygon
- features_cube_ei = {grid_id: conners}
  - similar to Cubes, but only contains the camera from ei
  - grid_id = (r, c)
  - conners= [topLeft, topRight, leftBottom, rightBottom, upper, lower]
- features_cube_camera_id = [keypointList,...] # index=keypoints+1
  - keypointList = [identifier, keypoint_id, keypoint_id, ..., keypoint_id]
  - identifier = int
  - keypoint_id = int
- CameraMask_cube = {identifier: polygon,...}
  - identifier = int
  - polygon = ndarray # 2x2 the 2D coordinates of conners in the image plane
- matches_cube_e1 = [matches_cube_pair_id,...]
  - matches_cube_pair_id = list
- MatchesReport_cube_e1 = [pair,...]
  - pair = [(camera_id1, camera_id2), Matches, inlierMatches]
  - Matches = np.array([[keypoint_id, keypoint_id, distance], ...])
  - inlierMatches = np.array([[keypoint_id, keypoint_id, distance], ...])
- TracksReport_cube_e1 = [TrackReport,...]
  - TrackReport = [views, edges]
  - views = np.array([[camera_id, keypoint_id], ...])
  - edges = np.array([[view_id, view_id, distance], ...])
- OriginTracksReport_cube_e1 = [TrackReport,...]
  - TrackReport = [views, edges]
  - views = np.array([[camera_id, keypoint_id], ...])
  - edges = np.array([[view_id, view_id, distance], ...])
- OriginTracksReport_cube_e1 = [matches_cube_pair_id,...]
  - matches_cube_pair_id = list
- matches_cube_pair_id = [item,...] # length = matches+1
  - item = [(camera_id1, camera_id2),match,match,...,match]
    - match= [feature, feature, matchQuality]
      - matchQuality = [distance,point3D]
      - feature = [camera_id, proj_info, keypoint_id]
        - proj_info = [proj_u,proj_v]
- matches_cube_pair_verified = [item,...] # length = matches+1
  - item = [(camera_id1, camera_id2),match,match,...,match]
    - match= [feature, feature, matchQuality]
      - matchQuality = [distance,point3D]
      - feature = [camera_id, proj_info, keypoint_id]
        - proj_info = [proj_u,proj_v]
- epochMatches = [match,...] # length = matches
  - match= [feature, feature]
    - feature = [camera_id, proj_info, keypoint_id]
      - proj_info = [proj_u,proj_v]
- epochMatchesQuality = [distance,...] # index = by epochMatches
  - record the Euclidean distance between the matched two descriptors.
  - distance = float
- epochTracksMatches_cube = [epochTracksMatches,...]
  - the descriptors of common feature track.
  - epochTracksMatches = [view_id,...]

- view_id = (camera_id, keypoints_id)

- epochTracksMatches_cube_Quality = [epochMatchesQuality,...] # index = by epochTracksMatches
  - the matches and similarity measures of common feature track.
  - epochMatchesQuality= [distance ,...]
  - distance = float
- Tracks_ViewId = {view_id,...}
  - record the Euclidean distance between the matched two descriptors.
  - view_id = (camera_id, keypoints_id)
- Camera_IdList = [camera_id,...]
  - The visible camera list for a certain cube.
  - camera_id = int
- Camera_IdList_cube = {grid_id:Camera_IdList}
  - The visible camera list for each cube.
  - grid_id = int
  - Camera_IdList = [camera_id,...]
- Cameras = {camera_id:Camera,...}
  - camera_id = int
  - Camera = [transform, reference, covariance, sensors_id, state, path, identifier]
    - transform= [camera_transform, camera_loc, camera_rot, camera_transform_PJCS]
    - reference = [camera_loc_Ref,location_accuracy,camera_rot_Ref,rotation_accuracy]
    - covariance = [camera_location_covariance, camera_rotation_covariance]
    - states = [label,camera_loc_Ref,camera_rot_Ref, selected, camera_orientation]
- Points = {point_id:Point,...}
  - point_id = int
  - Point = [Coordinates, Covariance, StandardDeviation, Quality, selected, valid, Track_id]
    - Coordinates=[[X, Y, Z],[X, Y, Z]] # in CLCS & PJCS respectively
    - Covariance = matrix # with 3x3 covariances in mm
    - StandardDeviation = [stdX, stdY, stdZ] # in mm
    - Quality = [RE, RU, PA, IC]
- Tracks = [Track,...] # index=Track_id, length=len(chunk.point_cloud.Tracks)
  - Track = [view,...] # index=view_id, length=number of views
    - view = [camera_id, project_info]
      - project_info = [projection.coord[0], projection.coord[1], projection.size，projection.track_id]
- Sensors = {sensor_id:Sensor,...}
  - sensor_id = int
  - Sensor = [calibration_params, calibration_matrix]
    - calibration_params = [f, cx, cy, b1, b2, k1, k2, k3, k4, p1, p2, p3, p4, height, width]
    - calibration_matrix = matrix # nxn covariance matrix, where n is the number of estimating parameters.
- Projections = {identifier:Camera_Projections,...}
  - Camera_Projections= [projection_info,...]
    - projection_info = [u, v, size, Track_id]
- CoordinateTransform = [M, LSE, T, R, S]
  - M = ndarray # 4x4 ndarray
  - LSE = ndarray # 4x4 ndarray
  - T = ndarray # 4x4 ndarray
  - R = ndarray # 3x3 ndarray
  - S = float
- CoordinateAttribute = [crs_name, crs_PJCS_wkt, crs_GCCS_wkt, crs_GDCS_wkt]
  - crs_PJCS_wkt = string # Projection coordinate system definition in WKT format.
  - crs_GCCS_wkt = string # Geocentric coordinate system definition in WKT format.
  - crs_GDCS_wkt = string # Geodetic coordinate system definition in WKT format.
- identifier = int
  - unique number for image, consist of photographing time and image name (e.g. an image taking at '2021:04:04 13:40:13' with name 'DSC00009.JPG', then its identifier is 2021040413401309).
- cameraPaths = {camera.photo.path:identifier,...}
  - camera.photo.path = string
  - identifier = int
- camera_ids = {identifier:camera_id,...}
  - identifier = int
  - camera_id = int
- point_ids = {Track_id :point_id,...} # if the track faild in built a point, the point_id will be -1
  - Track_id = int
  - point_id = int

- validTracks = [Track,...] # index=validTrack_id
  - Track = [view,...] # index=view_id, length=number of views
    - view = [camera_id, project_info]
      - project_info = [projection.coord[0], projection.coord[1], projection.size, projection.track_id]
- validTracks_ids = {validTrack_id:Track_id,...}
  - validTrack_id= int
  - Track_id= int
- pair_ids = {pair_id:match_state,...}
  - pair_id = (int, int)
  - match_state = [all, valid, invalid]
    - all = int # the total matches number
    - valid = int # the valid matches number
    - invalid = int # the invalid matches number
- match_ids = {pair_id:[valid_matches, invalid_matches],...}
  - pair_id = (int, int) # (camera_id1, camera_id2)
  - valid_matches = [Match,...]
  - invalid_matches = [Match,...]
    - Match = ([proj_u,proj_v,size,track_id],[proj_u,proj_v,size,track_id])
- chunkKeypoints = {identifier:Keypoints,...}
  - identifier = int
  - Keypoints = ndarray # dtype=float32, shape=(numKeypoints, 6)
    - Keypoint = [u,v,affinity1, affinity2, affinity3, affinity4]
- chunkDescriptors = {identifier:Descriptors,...}
  - identifier= int
  - Descriptors = ndarray # dtype=uint8, shape=(numKeypoints, 128)
- Boundary_CLCS = [boundary_X_MAX,boundary_X_MIN,boundary_Y_MAX,boundary_Y_MIN]
  - the boundary of sparse point cloud in XY plane, in meter
  - boundary_X_MAX = float
  - boundary_X_MIN = float
  - boundary_Y_MAX = float
  - boundary_Y_MIN = float
- Boundary_PJCS = [boundary_X_MAX,boundary_X_MIN,boundary_Y_MAX,boundary_Y_MIN]
  - the boundary of sparse point cloud in XY plane, in meter
  - boundary_X_MAX = float
  - boundary_X_MIN = float
  - boundary_Y_MAX = float
  - boundary_Y_MIN = float
- epochs = [datelist,...]
  - datelist = [date,...] # a datalist contains the date for an epoch.
  - date = 'YYYY:MM:DD'
  - for example: epochs = [['2020:10:14'], ['2021:07:03'], ['2021:12:03'], ['2021:12:06']]
- epochNum = [ei,...] # index=epoch_id, length=epochs number
  - ei = int # the number of images from the specified epoch
  - for example: epochNum= [e1, e2, e3, e4]
- signal = [si, ...]
  - si = int # Using 0 or 1 to represent whether there is image from the specified epoch
  - for example: signal = [0, 0, 1, 1]
- CamerasEpoch = [epoch_id,...] # index=camera_id, length=cameras number
  - epoch_id = int # assign which epochs dose camera belongs to
- TracksEpoch = [epochNum,...] # index=Track_id, length=Tracks number
  - epochNum = [e1, e2, e3, e4] # the counts for images of a certain epochs in a Track.
- EpochTracks = [Track,...] # index=EpochTracks_id (accompanied by EpochTracks_ids)
  - extract the Track that contains images from different epochs based on the TracksEpoch
  - Track = [view,...] # index=view_id, length=number of views
    - view = [camera_id, project_info]
      - project_info = [projection.coord[0], projection.coord[1], projection.size, projection.track_id]
- EpochTracks_ids = [Track_id,...] # index=EpochTracks_id
  - Since the EpochTracks extract from the Track_ids, the index of EpochTracks is different from Track_ids. So the original Track_id index of it is recorded in this associated varible.
- CommonTiePoint_IdList = [point_id, ...] # index=CommonTiePoint_id, length=CommonTiePoints number
  - analysis Common Tie Points (CTPs) based on TracksEpoch and extract the index of epoch tie points to this list.
  - point_id = int
- CommonTiePoint_Signal = [signal,...] # index=CommonTiePoint_id, length=CommonTiePoints number

- analysis common tie points(CTPS) based on TracksEpoch and assign the signal for each common tie point.
- signal = list

- CTPsCoord_CLCS = [Coordinates,...] # length=CommonTiePoints number
  - Coordinates = [X, Y, Z] # in CLCS
- CTPsCoord_PJCS = [Coordinates,...] # length=CommonTiePoints number
  - Coordinates = [X, Y, Z] # in PJCS
- EpochPairs
- EpochMatches
- PointsGrid = {grid_id: Point_IdList}
  - point_id list of each cell
  - grid_id = (r, c)
  - Point_IdList = [point_id,...]
- CTPsGrid = matrix # shape=(r, c), index=r, c
  - number of CTP in each cell
  - CTP number = int
- PointsGrid_NoCTPs = {grid_id: Point_IdList}
  - point_id list of the cell that contains no CTPs
  - grid_id = (r, c)
  - Point_IdList = [point_id,...]
- GridDots
- Cubes = { grid_id: conners}
  - grid_id = (r, c)
  - conners= [topLeft, topRight, leftBottom, rightBottom, upper, lower]
- CubesPolygon
- features_cube_ei = {grid_id: conners}
  - similar to Cubes, but only contains the camera from ei
  - grid_id = (r, c)
  - conners= [topLeft, topRight, leftBottom, rightBottom, upper, lower]
- features_cube_camera_id = [keypointList,...] # index=keypoints+1
  - keypointList = [identifier, keypoint_id, keypoint_id, ..., keypoint_id]
  - identifier = int
  - keypoint_id = int
- CameraMask_cube = {identifier: polygon,...}
  - identifier = int
  - polygon = ndarray # 2x2 the 2D coordinates of conners in the image plane
- matches_cube_e1 = [matches_cube_pair_id,...]
  - matches_cube_pair_id = list
- MatchesReport_cube_e1 = [pair,...]
  - pair = [(camera_id1, camera_id2), Matches, inlierMatches]
  - Matches = np.array([[keypoint_id, keypoint_id, distance], ...])
  - inlierMatches = np.array([[keypoint_id, keypoint_id, distance], ...])
- TracksReport_cube_e1 = [TrackReport,...]
  - TrackReport = [views, edges]
  - views = np.array([[camera_id, keypoint_id], ...])
  - edges = np.array([[view_id, view_id, distance], ...])
- OriginTracksReport_cube_e1 = [TrackReport,...]
  - TrackReport = [views, edges]
  - views = np.array([[camera_id, keypoint_id], ...])
  - edges = np.array([[view_id, view_id, distance], ...])
- OriginTracksReport_cube_e1 = [matches_cube_pair_id,...]
  - matches_cube_pair_id = list
- matches_cube_pair_id = [item,...] # length = matches+1
  - item = [(camera_id1, camera_id2),match,match,...,match]
    - match= [feature, feature, matchQuality]
      - matchQuality = [distance,point3D]
      - feature = [camera_id, proj_info, keypoint_id]
        - proj_info = [proj_u,proj_v]
- matches_cube_pair_verified = [item,...] # length = matches+1
  - item = [(camera_id1, camera_id2),match,match,...,match]
    - match= [feature, feature, matchQuality]
      - matchQuality = [distance,point3D]
      - feature = [camera_id, proj_info, keypoint_id]
        - proj_info = [proj_u,proj_v]
- epochMatches = [match,...] # length = matches

        - match= [feature, feature]
            - feature = [camera_id, proj_info, keypoint_id]
                - proj_info = [proj_u,proj_v]
- epochMatchesQuality = [distance,...] # index = by epochMatches
    - record the Euclidean distance between the matched two descriptors.
    - distance = float
- epochTracksMatches_cube = [epochTracksMatches,...]
    - the descriptors of common feature track.
    - epochTracksMatches = [view_id,...]
    - view_id = (camera_id, keypoints_id)
- epochTracksMatches_cube_Quality = [epochMatchesQuality,...] # index = by epochTracksMatches
    - the matches and similarity measures of common feature track.
    - epochMatchesQuality= [distance ,...]
    - distance = float
- Tracks_ViewId = {view_id,...}
    - record the Euclidean distance between the matched two descriptors.
    - view_id = (camera_id, keypoints_id)
- Camera_IdList = [camera_id,...]
    - The visible camera list for a certain cube.
    - camera_id = int
- Camera_IdList_cube = {grid_id:Camera_IdList}
    - The visible camera list for each cube.
    - grid_id = int
    - Camera_IdList = [camera_id,...]