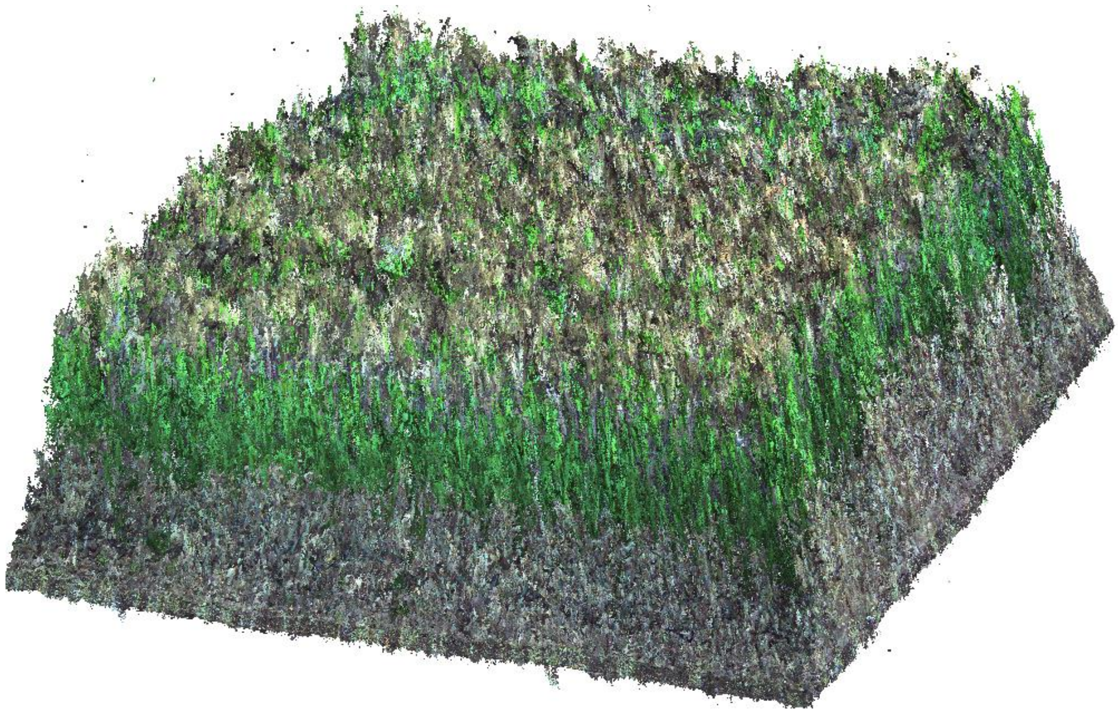
PyExpress Version 1.0

Photogrammetric Image Analysis for Vegetation Monitoring

- Based on Agisoft Metashape Python API –

https://github.com/Helmholtz-UFZ/PyExpress

Author: Martin Kobe, UFZ Leipzig, 02.2025
E-Mail: martin.kobe@ufz.de; martin.kobe@email.de



Inhaltsverzeichnis

1. Short description and general information

Agisoft Metashape is a stand-alone software product designed for photogrammetric processing of digital images and to generate 3D spatial data (Agisoft, 2024). This data can be used in Geopgraphic Information System (GIS) applications, cultural heritage documentation, visual effects production, and for indirect measurements of objects at various scales. The software supports processing of various types of imagery, including aerial, close-range, and satellite images, as well as the integration of LiDAR points. For more detailed information, please refer to one of the references listed below.

PyExpress version 1.0[1] builds on the functionalities of the Metashape Python API[2], offering a range of methods to support the development of tailored, fully automated workflows for photogrammetric image analysis. In the context of the EXPRESS[3] and MIRO[4] projects at the UFZ Leipzig, several exemplary workflows have been developed to automate the processing of UAV image data captured by DJI drones (DJI, 2024), specifically the Mavic 2 Enterprise Advanced (M2EA) and Mavic 3 Thermal (M3T). Additionally, Kobe et al. (2024) developed an automated workflow to evaluate fixed RGB stereo camera installations for continuous high-resolution vegetation monitoring using Metashape's 4D dynamic multiframe workflow, which is designed to handle time-lapse photogrammetric datasets. The outputs generated by these workflows can include a wide variety of data products such as point clouds, analysis reports, camera and reference parameters, digital elevation models (DEMs), orthomosaics, 3D models, tiled models, and precision maps, as calculated according to James et al. (2017). These outputs serve as valuable resources for a variety of downstream applications, such as Geographic Information Systems (GIS) or CloudCompare, facilitating advanced data analysis and, consequently, supporting decision-making processes. PyExpress also supports interaction with MinIO[5], a High Performance Object Storage system, enabling efficient handling of large datasets and seamless cloud storage integration (MinIO, Inc, 2024). For more details, please refer to one of the references below.

Software user information – PyExpress adapts Metshape versions higher than v.2.1.0 and is available as an open-source product and currently includes only the functionalities required for the drone and stereo camera workflows developed so far. Due to the extensive range of capabilities in Metashape, the implemented functions are divided into two categories: *processing core functions*, which mirror and specify all the core processing functionality from Metashape's GUI dropdown menu Workflow in the main task bar, and *processing optional functions*, which mirror xyz-processing functions and cover tasks such as reference management, calibration, tie point cloud filtering, bounding box adjustments, or result export specifications. Additionally, *utility and data management tools* are available to assist with project administration and image data management.

Example workflow – This tutorial also includes a sample workflow for drone data analysis, demonstrating the performance of PyExpress through a practical application example.

Contribution and request – For support or inquiries, including requests for additional, please contact martin.kobe@ufz.de or rikard.grass@ufz.de. As PyExpress is continuously evolving, contributions via merge requests are welcome. For details, see the CONTRIBUTING.md file on GitHub.

Funding – This research was funded by the Federal Ministry of Food and Agriculture, grant no. 28DE102B18.

[1] PyExpress git repository: https://github.com/Helmholtz-UFZ/PyExpress
[2] Metashape download center: https://www.agisoft.com/downloads
[3] Project EXPRESS: https://www.digitalisierung-landwirtschaft.de/
[4] Project MIRO: https://obstbau-digital.de/
[5] MinIO object storage Python API: https://github.com/minio/minio

2. PyExpress package: Installation, program structure, methods

Find general information about the software and its requirements, as well as instructions for installation from the git repository. Please note that the software is under constant development and optimization. Updates and improvements are made regularly, so it is recommended to periodically pull the latest changes from the repository.

2.1. Requirements

Metashape version:	2.1.0 or higher
Python version:	3.7 to 3.11
External packages:	astral, influxdb_client, minio, numpy, opencv-python, pandas, python-abc, pysftp, pyyaml, tifffile, scikit-learn, scikit-image, seaborn → required packages are installed by using pip package manager

2.2. Installation of PyExpress and Metashape

PyExpress v.1.0 is available in the Git repository https://github.com/Helmholtz-UFZ/PyExpress. It is recommended to clone the package into your working directory and install it from there into the project-specific Python environment. To do this, use the following commands in your terminal:

(a) Cloning PyExpress into your working directory using either

https:	git clone -b <branch_name> <repository_url>
ssh-key:	git clone -b <branch_name> git@<host>:<repository_path>.git

(b) Installing PyExpress by using pip and the file setup.py

user (recommended):	pip install <dirpath_to_setup.py>.
contributor:	pip install -e <dirpath_to_setup.py>.

Note, if installed as user: If an updated version of PyExpress is pulled from the Git repository the package must be reinstalled using the setup.py-file as described above.

Note, if installed as contributor: Any changes made to the package, as well as updates pulled from Git repository, will be applied immediately. Merge requests are welcome.

The **Metashape Python API** is available as a .whl-file on https://www.agisoft.com/downloads/installer/. The latest version, 2.2.0 (date: 12.02.2025), requires Python 3.7 to 3.11 and should be installed into the project-specific Python environment using the following command:

(c) pip install <source_path>\<package_name>*.whl

License file: Copy the Metashape license file (metashape.lic) to the directory where the Metashape Python API is installed or to the directory where your main control script is located (recommended).

2.3. Package structure version 1.0

Figure 1: Overview of the photogrammetric analysis package PyEpxress, version 1.0. The included packages and modules are under permanent version controll and are continuously being developed and expended.

2.4. PyExpress: Methods, Functions, Import

In the following section, a comprehensive overview of the functionalities in PyExpress is provided (cf. Fig. 1). Each function is explained in detail, with illustrative examples demonstrating how to create an individual project. Function arguments and input parameters are outlined, highlighting their roles, expected types, and usage to ensure clear understanding. It is recommended to import specific packages with an alias for better code readability and to ensure access to all necessary functionalities for your project data.

Important Note: Whenever ****kwargs** appears, it refers to the keyword arguments used in the corresponding adapted Metashape function. By using Metashape function arguments, you can customize the behavior of adapted Metashape functions with additional options tailored to your specific needs. The available function arguments are described in detail in the current Metashape Python API User Manual[6] and can be referenced from there.

2.4.1. Configuration file

The funtionalities of PyExpress are mainly controlled by settings defined within a configuration file. Both YAML (recommended) and JSON file formats can be used. If you prefer JSON, please ensure that keyword hierarchies are maintained. The package includes templates of project configuration files for drone and stereo projects, each thoroughly documented directly within the template. An appropriately adjusted configuration file is essential and should be customized for each execution of a script using PyExpress. To achieve this, dependencies, input types, and keyword hierarchies must be strictly followed and formatted in valid Python syntax (Fig. 2)! Corresponding templates for stereo or drone projects can be found u nder *PyExpress.WorkflowExample.UserSettings*!

```yaml
#
input:
  general:
    new_project: bool        # whether to create a new project structure
    ID_source: str           # source for project ID generation: ['campaign', 'string']
    ID_string: str           # specify projectID if 'string' was chosen above as ID source
#
```

Figure 2: Excerpt from the template for the configuration of a drone project. The complete template can be found in the PyExpress module: ./PyExpress/WorkflowExample/UserSettings/config_droneProject_template.yaml.

[6] Metashape Python API User Manual: https://www.agisoft.com/downloads/user-manuals/

2.4.2. ImageAnalysis: Project class instance

Class *TypeOfProject*(config_data, project_dir, image_dir, config_name)

→ Creates an instance of a Drone or Stereo project type to serve as a container for photogrammetric analysis, while also holding a collection of methods and metadata associated with the project type (cf. Example 1).

Class parameters

- ***TypeOfProject*** **[StereoProject, DroneProject]**
  StereoProject Photogrammetric analysis of data from rigid installed cameras
  DroneProject Photogrammetric analysis from campaign-based UAV data

- config_data (dict): contents of the photogrammetric project configuration file
- project_dir (str): absolute path to your Metashape project folder
- image_dir (str): absolute path to your image folder
- config_name (str): filename of the configuration file used

Public project class methods and their parameters

addPhotosToChunk(image_dir, file_format, save_project=(False, “”), **kwargs)

→ Adds photos to the active chunk in your project instance.

- image_dir (str): absolute path to folder of preprocessed images
- file_format (str): image format, e.g. ‘JPG’, ‘TIFF’, …
- save_project[7] (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs refers to the Python API class *Metashape.Chunk.addPhotos*

applyVegetationIndex(config_data, save_project=(False, “”),)

→ Sets vegetation-dependent raster transformation specifications from the configuration file for the photogrammetric project, including index, range, formula, and palette. Applies for drone projects only.

- config_data (dict): contents of the photogrammetric project configuration file
- save_project (tuple): (True/False, ‘label of the chunk to be activated’)

logging(msg)

→ Generates a terminal log and saves it with an additional timestamp in the project log file *(cf. Fig. 2).*

- msg (str): message to be logged in the protocol file

saveMetashapeProject(active_chunk)

→ Saves your project into the path *./metashape_prj/project_data/projectID.psx (cf. Fig. 2).*

- active_chunk (str): label of the active chunk in the MS project

List of accessible project class internal objects

- chunk (MSobj[8]): active chunk object

[7] After saving, a project reloads automatically due to Metashape default settings. Here, we recommend to select the chunk to activate by its label; otherwise, Metashape defaults to activating the chunk with ID number -1.

[8] MSobj – Metashape object type

- chunk_ID (int): ID of the active chunk; -1 indicates a newly created chunk is active
- config (PRJobj[9]): enables access to all config params, preserving keyword hierarchy
- config_name (str): filename of the configuration file used in the project
- doc (MSobj): document which is connected to your project instance
- drone_IR/RGB (bool): classifiers for drone IR and RGB data
- export_dir (str): directory for exported results
- extensive_logging (bool): defines if workflow logging is enabled/disabled; default: TRUE
- GCP_path (DP[10]) (str): csv-file with real-world coordinates of Ground Control Points (GCP)
- ignore_lock (bool): ignore lock state for project modifications; existing projects only
- image_dir (str): directory in which your preprocessed images are stored
- image_format_conv (str): format of preprocessed/converted images
- image_format_raw (str): format of raw images from image acquisition
- log_dir (str): directory in which the log-file is being created
- marker_proj_path (DP) (str): csv/xml-file with GCP image projections as x-y image coordinates
- project_dir (str): directory for export of metashape result and project files
- project_ID (str): biunique identification of your project
- project_status (str): classify if you work on a new or an existing project
- read_only (DP) (bool): open a Metashape document in read-only mode
- reflect_panel_path (str): csv-file with reflectance panel information
- save_dir (str): directory for saved project data
- sensor_type (str): spectral channel used for image acquisition
- stereo_IR/RGB (bool): classifiers for stereo IR and RGB data
- vegetation_index (str): raster band/formula used for raster (color space) tansformation

Example 1: Initializing a photogrammetric project and the first photogrammetric workflow steps.

```
# This example workflow demonstrates how to create a drone-based photogrammetric project instance
and add photos to the active chunk  based on input paramters configured in advance. It also includes
printouts to illustrate how to access internal class attributes within the project.

import Metashape as ms
import PyExpress.ImageAnalysis as ppp

MyProject = ppp.DroneProject(config_data    = configDataDictionary,
                             project_dir    =  absolute_project_directory
                             image_dir      =  absolute_image_directory
                             config_name  = '/…/config.yaml')

MyProject.addPhotosToChunk(image_dir     = absolute_image_directory ,
                           file_format   = MyProject.imageFormat,
                           layout        = ms.UndefinedLayout,              (Metashape kwarg)
                           save_project = (True, MyProject.chunk.label))

print(f'ID/name of my project: {MyProject.project_ID}')
print(f'Status of my project: {MyProject.project_status}')
print(f'Point cloud classification is using: {MyProject.config.metashape.point.cloud.classification.set})

…
```

[9] PRJobj – Object of the photogrammetric project instance

[10] DP – For Drone Projects only. This parameter is not available or used for Stereo Project Class.

2.4.3. ImageAnalysis: Main metashape workflow methods

This section describes the methods implemented in PyExpress so far, following the core photogrammetric processing steps as seen in the Metashape GUI workflow panel. The non-void functions are located within the *PyExpress.ImageAnalysis* module and should be imported with an alias for simplified access to the contained functions and classes (cf. example applications). Without exception, all of these functions return updated working project instances as they process the image data. The methods and functionalities are compatible with both Metashape versions 1 and 2.

For multi-frame chunks, such as those used in stereo project instances, the chunk object automatically includes a reference to the specific frame currently being processed. This reference is initialized to the current frame by default. Methods that operate on a per-frame basis specifically target this active frame within the chunk. The following functions apply to both single-frame (drone projects) and multi-frame chunks (stereo projects), ensuring operations are performed appropriately based on the chunk type. The processing functions are listed in the recommended order of application:

a. Methods

matchPhotos(project, save_project=(False, “”), **kwargs)

→ Performs image matching for each frame in the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.Chunk.matchPhotos*

alignCameras(project, save_project=(False, “”), **kwargs)

→ Performs image alignment for each frame in the the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.Chunk.alignCameras*

buildDepthMaps(project, save_project, **kwargs)

→ Builds a depth map from the matched photos for each frame in the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.Chunk.buildDepthMaps*

buildPointCloud(project, save_project=(False, “”), **kwargs)

→ Builds a dense point cloud for each frame in the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.Chunk.buildPointCloud*

buildModel(project, save_project=(False, “”), **kwargs)

→ Builds a 3D model for each frame in the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, 'label of the chunk to be activated')
- **kwargs — refers to the Python API class *Metashape.Chunk.buildModel*

buildDEM(project, save_project=(False, “”), **kwargs)

→ Builds a Digital Elevation Model (DEM) for each frame in the the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, 'label of the chunk to be activated')
- **kwargs — refers to the Python API class *Metashape.Chunk.buildDEM*

buildUV(project, save_project=(False, “”), **kwargs)

→ Maps a 3D model's surface onto a 2D plane for texture application (UV mapping).

- project — Metashape project instance; your drone/stereo project instance
- save_project — (tuple): (True/False, 'label of the chunk to be activated')
- **kwargs — refers to the Python API class *Metashape.Chunk.buildUV*

buildOrthoProjection(project, coord_system, save_project=(False, “”), **kwargs)

→ Creates an ortho projection (orthomosaic) for each frame in the active Metashape chunk.

- project — Metashape project instance; your drone/stereo project instance
- coord_system — refers to the Python API class *Metashape.CoordinateSystem*
- save_project — (tuple): (True/False, 'label of the chunk to be activated')
- **kwargs — refers to the Python API class *Metashape.Chunk.buildOrthomosaic*

Example 2: Application of main Metashape methods on a photogrammetric project.

```
# This example workflow demonstrates how non-void processing functions are applied to analyse your
digital image data. Metashape keyword arguments (**kwargs) are simply handed over and applied.

…

MyProject = ppp.matchPhotos(project              = MyProject,
                            downscale            = 1,
                            generic_preselection = False,
                            keypoint_limit       = 10000,
                            tiepoint_limit       = 2000,
                            save_project         = (True, MyProject.chunk.label))

MyProject = ppp.alignCameras(project          = MyProject,
                            adaptive_fitting = True,
                            save_project     = (True, MyProject.chunk.label))
…

MyProject = ppp.buildOrthoProjection(project      = MultiProject,
                                     coord_system = ms.CoordinateSystem('EPSG::25833'),
                                     fill_holes   = False,
                                     save_project = (True, active_chunk.label))
…
```

2.4.4. ImageAnalysis: Optional and supporting methods

The following section outlines optional functionalities implemented, based on requirements for automated, specific workflows in the photogrammetric analysis of image data from the following case studies: *(a) Single Frame Project* – campaign-based drone data analysis in fruit and viticulture, and *(b) 4D Multiframe Project* – time-lapse analysis of stereoscopic image data in biomass monitoring. The optional methods are located within the *PyExpress.ImageAnalysis* module and should be imported with an alias for simplified access to the contained functions and classes (cf. example applications). These functionalities leverage only a subset of Metashape's full capabilities and may be expanded as needed or when additional steps are introduced. To simplify navigation, all optional methods are organized within class structures, each focused on a specific topic.

b. Methods

metashape_initial_check(version_GUI: str, enable_GPU: boolen, enable_CPU: boolean)

→ Checks if the Metashape API and GUI versions are compatible, and if the Metashape API is active. Enables GPU and CPU acceleration as specified in the configuration file.

- version_GUI (str): version number of the locally installed Metashape GUI (e.g., 2.1.2)
- enable_GPU (bool): enable GPU acceleration
- enable_CPU (bool): enable CPU acceleration if GPU acceleration is enabled

c. Class BBox()

This class was created to manipulate the bounding box, which defines the spatial extent of a point cloud or 3D model. It is particularly useful for optimizing processing, focusing on relevant data, and adjusting the area for camera alignment and model generation.

return_center(project, return_type=*Metashape.Vector*)

→ Returns the center of the region in the active chunk as a vector in *Metashape.Vector* or tuple format.

- project Metashape project instance; your drone/stereo project instance
- return_type (type): [Metashape.Vector, tuple] - xyz vector return type

redefine_center(project, xyz_coord, xyz_type=*Metashape.Vector*, save_project=(False, “”))

→ Moves the region center in the active chunk. Returns the updated project.

- project Metashape project instance; your drone/stereo project instance
- xyz_coord (see below): xyz vector to set as the new region center
- xyz_type (type): [*Metashape.Vector*, tuple] - xyz vector format
- save_project (tuple): (True/False, ‘label of the chunk to be activated’)

return_xyz_extent(project, return_type=Metashape.Vector)

→ Returns the extent of the region in the active chunk as a vector in *Metashape.Vector* or tuple format.

- project Metashape project instance; your drone/stereo project instance
- return_type (type): [*Metashape.Vector*, tuple] - xyz vector return type

resize_xyz_extent(project, xyz_ extent, xyz_type=*Metashape.Vector*, save_project=(False, “”))

→ Moves the region center in the active chunk. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- xyz_extent — (see below): xyz vector to set as the new extent
- xyz_type — (type): [*Metashape.Vector*, tuple] - xyz vector format
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

return_rot_matrix(project, return_type=*Metashape.Matrix*)

→ Returns the rotation matrix of the region in the active chunk as a *Metashape.Matrix* or a list.

- project — Metashape project instance; your drone/stereo project instance
- return_type — (type): [*Metashape.Matrix*, tuple] - rot matrix return type

rotate_region(project, input_matrix, input_type=*Metashape.Matrix*, save_project=(False, “”))

→ Updates the rotation matrix values of the region in the active chunk. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- input_matrix — (see below): 3x3 matrix to set as new rotation matrix
- input_type — (type): [*Metashape.Vector*, tuple] – format of 3x3 input matrix
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

redefine_auto_multiframe(project, reference_ID=0, save_project=(False, “”))

→ Resizes the bounding box in a multiframe project based on the maximum extent in the x, y, or z direction from the different frame tie point clouds. The center of the region is moved to match the center of the image defined by the reference ID. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- reference_ID — (int): ID of the reference frame/image
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

d. Class Calibration()

The calibration class supports lens and sensor calibration in Metashape, setting attributes and distortion parameters from input configuration files, following standard photogrammetric practices.

import_from_file(project, save_project=(False, “”), **kwargs)

→ Imports lens calibration values from a specified calibration file. Returns the updated project.

- project — Metashape project instance; your drone/stereo project
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class method *Metashape.Calibration.load*

set_sensor_param_stereo(project, save_project=(False, “”), **kwargs)

→ Sets sensor attributes and precalibrated lens distorsion parameters for rigid cameras in a stereo vision arrangement based on configuration file specifications. Returns the updated project.

- project — Metashape project instance; your drone/stereo project
- config_data — (dict): contents of the photogrammetric project configuration file
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

e. Class Export()

This class enables convenient export of photogrammetric results, supporting user-defined formats and specifications. For multiframe projects, export filenames can be generated using regex patterns applied to image names via Python's *re* module. Refer to the documentation[11] for regex usage and syntax.

export_from_list(project, export_list)

→ Exports Metashape results from the active chunk specified by a list. This method can be used as shortcut and uses default parameters without any keyword argument specifications. For more specific export options, please use the corresponding methods in the Export class.

- project — Metashape project instance; your drone/stereo project instance
- export_list — (list of str): chose one or more results to be exported from the following list [camera, dem, dem_trafo, marker, model, ortho, ortho_trafo, point_cloud, precision_map, report, tiled_model]

Export formats by default

- Camera — XML
- DEM — TIFF
- DEM_trafo — basic raster transformed by value (_val.TIFF) and palette (_pal.TIFF)
- Marker — XML and/or CSV
- Model (3D) — OBJ
- Orthomosaic — TIFF
- Orthomosaic_trafo — basic raster transformed by value (_val.TIFF) and palette (_pal.TIFF)
- PointCloud — OBJ
- PrecisionMap — TXT
- Report — PDF
- TiledModel — OBJ

camera(project, **kwargs)

→ Exports the positions of the point cloud and/or cameras from the active chunk.

- project — Metashape project instance; your drone/stereo project instance
- **kwargs — refers to the Python API class *Metashape.Chunk.exportCameras*

marker(project, export_format="xml", **kwargs)

→ Exports marker projections from the active chunk in CSV, XML, or both formats to the active chunk's export path within the project, as well as to the predefined export path specified in the config file.

- project — Metashape project instance; your drone/stereo project instance
- export_format — (str): ['csv', 'xml', 'both'] - export format of marker projections
- **kwargs — refers to the Python API class *Metashape.Chunk.exportMarkes*

model(project, pattern="", string="", **kwargs)

→ Exports the generated 3D model for the active chunk. Please note that the parameters "pattern" and "string" are only required for multiframe projects, such as the StereoProject class.

[11] The documentation of the Python *re* module is avilable at https://docs.python.org/3/library/re.html.

- project Metashape project instance; your drone/stereo project instance
- pattern (str): pattern from Python *re* module to extract an ID from an image name
- string (str): designated export filename; will be combined with a numeric ID
- **kwargs refers to the Python API class *Metashape.Chunk.exportModel*

point_cloud(project, pattern="", string="", **kwargs)

→ Exports the point cloud from the active chunk. Please note that the parameters "pattern" and "string" are only required for multiframe projects, such as the StereoProject class.

- project Metashape project instance; your drone/stereo project instance
- pattern (str): pattern from Python *re* odule to extract an ID from an image name
- string (str): designated export filename; will be combined with a numeric ID
- **kwargs refers to the Python API class *Metashape.Chunk.exportPointCloud*

precision_map(project, pattern="", string="", export_format="txt", **kwargs)

→ Exports the presicion map as text file from the active chunk. This functionality is based on James et al. (2017)[12].

- project Metashape project instance; your drone/stereo project instance
- pattern (str): pattern from Python *re* module to extract an ID from an image name
- string (str): designated export filename; will be combined with a numeric ID
- export_format (str): text file format for export e.g. txt, log, dat, csv

raster(project, export_type, pattern="", string="", **kwargs)

→ Exports the raster DEM or raster orthomosaic from the active chunk. Please note that the parameters "pattern" and "string" are only required for multiframe projects, such as the StereoProject class.

- project Metashape project instance; your drone/stereo project instance
- export_type (str): used to classify the main type of exported raster in the filename
- pattern (str): pattern from Python *re* module to extract an ID from an image name
- string (str): designated export filename; will be combined with a numeric ID
- **kwargs refers to the Python API class *Metashape.Chunk.exportRaster*

report(project, **kwargs)

→ Exports the processing report in PDF-format.

- project Metashape project instance; your drone/stereo project instance
- **kwargs refers to the Python API class *Metashape.Chunk.exportReport*

tiled_model(project, pattern="", string="", **kwargs)

→ Exports the generated tiled model from the active chunk. Please note that the parameters "pattern" and "string" are only required for multiframe projects, such as the StereoProject class.

- project Metashape project instance; your drone/stereo project instance
- pattern (str): pattern from Python *re* module to extract an ID from an image name
- string (str): designated export filename; will be combined with a numeric ID
- **kwargs refers to the Python API class *Metashape.Chunk.exportTiledModel*

[12] For more information visit also visit the project page http://tinyurl.com/sfmgeoref.

f. Class PointCloud()

The point cloud class provides tools for classifying and filtering 3D point clouds in Metashape. It includes ground and non-ground classification based on optimal parameter settings for UAV imagery, following Klápště et al. (2020), as well as point class removal and filtering for noise, ground, and unclassified points. Configuration-based specifications are used, and results can be organized in new chunks.

Class Classification()

classify_ point_cloud(project, class_param, save_project=(False, “”), **kwargs)

→ Classifies the point cloud in the active chunk into ground and non-ground classes, following Axelsson (2000). Compatible with both Metashape versions 1 and 2. Returns the updated project.

Important remarks for users:

The classification parameters are stored in a class-like structure within a Python script available at *PyExpress.WorkflowExamples.supportData*. Here, users can create custom classification types/sets or adjust values by following the class structure and the syntax provided. It is recommended to adjust the values after testing different parameters and using the test mode, which can be specified in the *metashape / point cloud / classification* block of your project configuration file (keys: type, max_angle, max_distance, cell_size).

- project — Metashape project instance; your drone/stereo project instance
- class_param — (list): specfication in the format - [True/False, type, paramset]

  Specifications
  type: 'test', 'vine', 'crop'
  paramset if type is not 'test': 'set1', 'set2', ...
  paramset if type is 'test': {'max_angle': float, 'max_distance': float, 'cell_size': float}

  Examples:
  class_param = [true, ‘vine’, ‘set1’]
  class_param = [true, ‘test’, {'max_angle': 10, 'max_distance': 5, 'cell_size': 1.2}

- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.PointCloud.classifyGroundPoints*

prepare_classification(project, config_data)

→ Extracts classification specifications from the configuration file and, if specified, generates a new chunk dedicated to point cloud classification. Returns a tuple with specifications for point cloud classifiction and the updated Metashape project.

- project — Metashape project instance; your drone/stereo project instance
- config_data — refers to the Python API class *Metashape.PointClass*

Class Filter()

delete_point_class(project, point_class=*Metashape.PointClass.Ground*, render_preview=False, save_project=(False, “”))

→ Removes a given point class from classified point cloud. Specifically designed for drone projects. Compatible with both Metashape versions 1 and 2. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- point_class — refers to the Python API class *Metashape.PointClass*
- render_preview — (bool): export preview into your export directory
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

filter_from_list(project, filter_unclass=False, filter_high_noise=False, filter_ground=False, render_preview=False, save_project=(False, “”))

→ Function that simultaneously calls several filter options controlled by Boolean flags. So far implemented for filtering high noise, ground, and unclassified points. Specifically designed for drone projects. Compatible with both Metashape versions 1 and 2. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- filter_unclass — (bool): filter for unclassified points
- filter_high_noise — (bool): filter for high noise points
- filter_ground — (bool): filter for ground points
- render_preview — (bool): export preview into your export directory
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

g. Class Reference()

The reference class provides tools for defining geometrical objects and reference parameters, including markers, scalebars, GCP coordinates, camera positions, and accuracies, as well as operations like chunk transformation and camera optimization to enhance camera alignment and 3D point cloud precision.

add_scalebar(project, distance, accuracy, enable, image_IDs=(0,1), optimize_cam=False, save_project=(False, “”))

→ Defines a scale bar between two specified image IDs in the active chunk. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- distance — (float): distance between a given start- and an endpoint in meters
- accuracy — (float): accuracy of the distance/scalebar in meters
- enable — (bool): enable/disable the scalebar
- image_IDs — (list): IDs of the images between which a scalebar will be defined
- optimize_cam — (bool): apply camera optimization using default values; refers to the Python API class *Metashape.Chunk.optimizeCameras*
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

import_marker_proj(project, coord_system, optimize_cam=False, save_project=(False, “”))

→ Imports marker projections into an active chunk. Imported format can be either CSV or XML. Very useful if markers have already been flagged and saved for a specific set of images. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance

- coord_system (obj): coordinate system as either 'local' or EPSG code (e.g., "EPSG::4326")
- optimize_cam (bool): apply camera optimization using default values;
  refers to the Python API class *Metashape.Chunk.optimizeCameras*
- save_project (tuple): (True/False, 'label of the chunk to be activated')

import_marker_coord(project, coord_system, optimize_cam=False, save_project=(False, ""), **kwargs)

→ Imports a list of measured real-world coordinates for markers into the active chunk. Should be used directly following either the method *set_marker_manually()* or the method *import_marker_proj().* Returns the updated project.

- project Metashape project instance; your drone/stereo project instance
- coord_system (obj): coordinate system as either 'local' or EPSG code (e.g., "EPSG::4326")
- optimize_cam (bool): apply camera optimization using default values;
  refers to the Python API class *Metashape.Chunk.optimizeCameras*
- save_project (tuple): (True/False, 'label of the chunk to be activated')
- **kwargs refers to the Python API class *Metashape.PointCloud.classifyGroundPoints*

set_marker_manually(project, config_data, save_project=(False, ""))

→ Pauses execution to add marker flags manually in the Metashape GUI. Ensure all markers are set with green flags. Note: Follow the example shown in Figure 3. Returns the updated project.

- project Metashape project instance; your drone/stereo project instance
- config_data (dict): contents of the photogrammetric project configuration file
- save_project (tuple): (True/False, 'label of the chunk to be activated')

Note
Automatically exports marker flags to a predefined export path from the configuration file, as well as to the active chunk's export path in the project, in both CSV and XML formats.

set_camera_param(project, config_data, optimize_cam=False, save_project=(False, ""))

→ Sets camera position and accuracy parameters based on configuration file specifications. Designed for projects with fixed camera installations, e.g. stereo camera projects. Returns the updated project.

- project Metashape project instance; your drone/stereo project instance
- config_data (dict): contents of the photogrammetric project configuration file
- optimize_cam (bool): apply camera optimization using default values;
  refers to the Python API class *Metashape.Chunk.optimizeCameras*
- save_project (tuple): (True/False, 'label of the chunk to be activated')

set_reference_param(project, config_data, optimize_cam=False, save_project=(False, ""))

→ Sets reference parameters corresponding to the Reference Settings block in the Metashape GUI and based on configuration file specifications. Returns the updated project.

- project Metashape project instance; your drone/stereo project instance
- config_data (dict): contents of the photogrammetric project configuration file
- optimize_cam (bool): apply camera optimization using default values;
  refers to the Python API class *Metashape.Chunk.optimizeCameras*
- save_project (tuple): (True/False, 'label of the chunk to be activated')

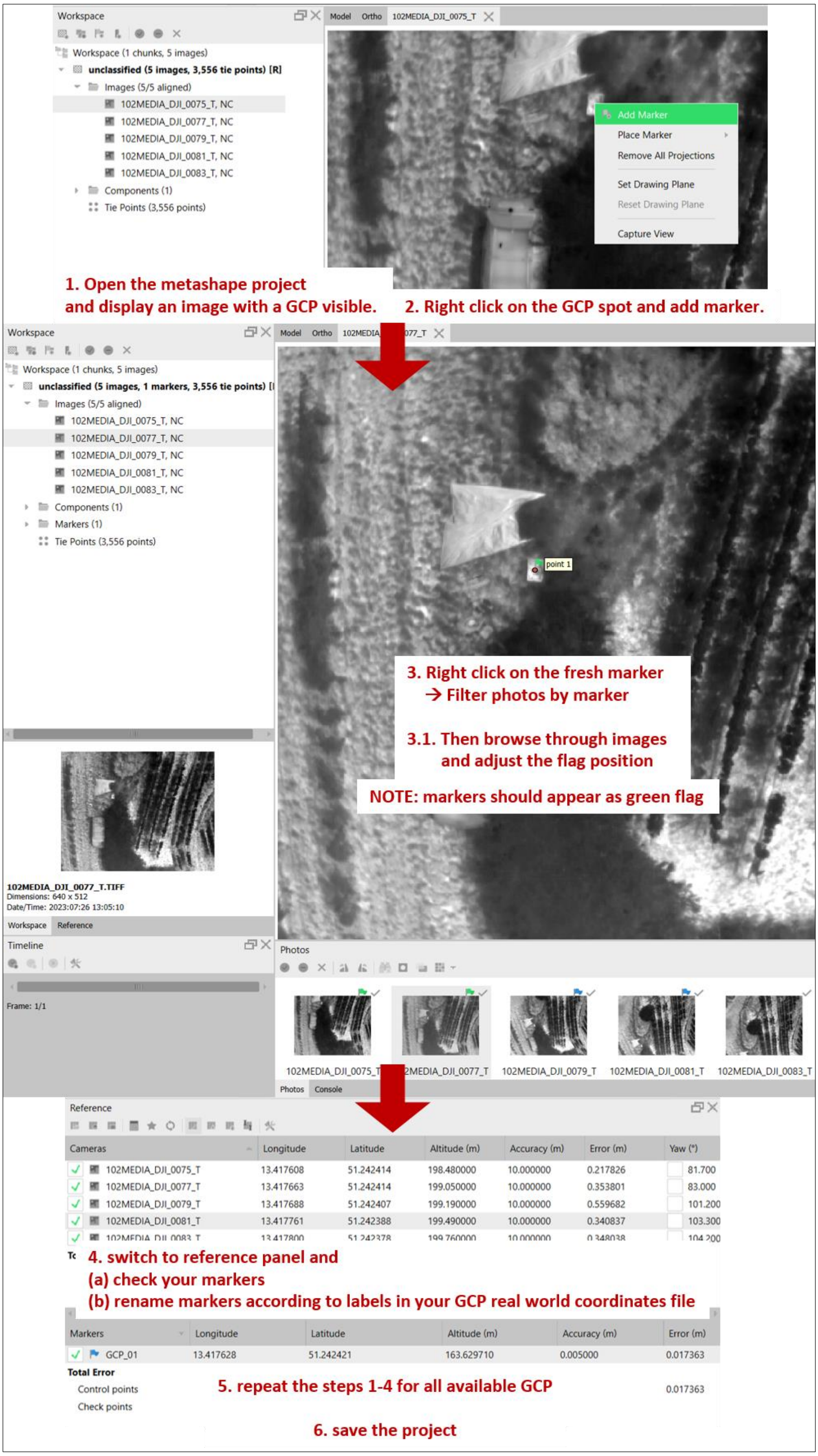


Figure 3: Suitable workflow for manually selecting marker flags in drone images with the Metashape GUI.

optimize_cameras(project, update_transform, save_project=(False, “”), **kwargs)

→ Performs optimization of tie points / camera parameters. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- update_transform — (bool): update chunk transformation before optimizing cameras
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)
- **kwargs — refers to the Python API class *Metashape.PointCloud.classifyGroundPoints*

update_transform(project, save_project=(False, “”))

→ Updates chunk transformation based on reference data. Returns the updated project.

- project — Metashape project instance; your drone/stereo project
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

h. Class TiePointCloud()

This class provides tools for managing and filtering tie point clouds in Metashape including point selection and removal based on thresholds for specified criteria, ensuring compatibility with both Metashape versions 1 and 2.

Class Filter()

gradual_selection(project, criterion, threshold, save_project=(False, “”))

→ Selects points from a tie point cloud based on a threshold for the specified criterion. Compatible with both Metashape versions 1 and 2. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- criterion — refers to the Python API class *Metashape.TiePoints.Filter.Criterion*
- threshold — (float): used for tie point selection
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

gradual_removal (project, criterion, threshold, optimize_cameras, save_project=(False, “”))

→ Removes points from a tie point cloud based on a threshold for the specified criterion. Compatible with both Metashape versions 1 and 2. Returns the updated project.

- project — Metashape project instance; your drone/stereo project instance
- criterion — refers to the Python API class *Metashape.TiePoints.Filter.Criterion*
- threshold — (float): used for tie point selection
- save_project — (tuple): (True/False, ‘label of the chunk to be activated’)

return_minmax(project, criterion)

→ Calculates the minimum and maximum values of a tie point cloud based on the specified criterion. Compatible with both Metashape versions 1 and 2.

- project — Metashape project instance; your drone/stereo project instance
- criterion — refers to the Python API class *Metashape.TiePoints.Filter.Criterion*
- threshold — (float): used for tie point selection

2.4.5. UtilityTools: Collection of custom and useful tools

The UtilityTools module offers various tools for data handling and file manipulation within photogrammetric workflows. These functions support format conversions, project setup, file transfers, and efficient project management, including directory operations and configuration management. Highlights include:

- **Format Conversion**: *convert_JSON_to_YAML, convert_YAML_to_JSON*
  > Convert data between JSON and YAML formats.
- **Project Setup**: *create_stereo_project, create_UAV_project*
  > Initialize structured project folders with optional marker copying.
- **File Operations**: *copy_marker_ref, transfer_images, get_filelist*
  > Efficient file copying and listing, with recursive options.
- **Utility Functions**: *extract_date_stamps, log, open_parameters, open_project, some filters*
  > Assist with logging, parameter access, date extraction, and more.

a. Methods

convert_JSON_to_YAML (source_path, save_path, directory=False, recursive=False)

→ Converts the contents of a JSON file into YAML format and saves it.

- source_path (str): directory of JSON files (if directory==True) or full path to a single JSON file (if directory==False)
- save_path (str): directory, where converted YAML files will be saved
- directory (bool): is the source a directory (True) or a file name (False)
- recursive (bool): searching also subdirectories for JSON files

convert_YAML_to_JSON (source_path, save_path, directory=False, recursive=False)

→ Converts the contents of a YAML file into JSON format and saves it.

- source_path (str): directory of YAML files (if directory==True) or full path to a single YAML file (if directory==False)
- save_path (str): directory, where converted JSON files will be saved
- directory (bool): is the source a directory (True) or a file name (False)
- recursive (bool): searching also subdirectories for JSON files

copy_marker_ref(config_data, prj_dir)

→ Copies csv files with marker coordinates, marker image projections, or reflectance panel information from the specified source (as defined in the config file) to target project directory.

- config_data (dict): contents of the photogrammetric project configuration file
- prj_dir (str): target project directory

create_stereo_project(config_data, config_path)

→ Creates an empty stereo project structure. The project name and the target project path are specified in the configuration file's input section (general, campaign, and project). Copies the configuration file to the target project path. Returns the image and project paths of the *StereoProject*.

- config_data (dict): contents of the photogrammetric project configuration file
- config_path (str): full path to the project configuration file

create_stereo_project(config_data, config_path, copy_markers=True)

→ Creates an empty UAV project structure. The project name and the target project path are specified in the configuration file's input section (general, campaign, and project). Copies the configuration file to the target project path. Returns the image and project paths of the *DroneProject*.

- config_data (dict): contents of the photogrammetric project configuration file
- config_path (str): full path to the project configuration file
- copy_markers (bool): copy marker files to new project directory

extract_date_stamps(file_list, pattern="")

→ Extracts the date from filenames in a given list by matching them against a specified regex pattern. Returns a list of strings.

- file_list (list): list of filenames
- pattern (str): regex pattern (Python module *re*) to extract an ID from an image name

filter_filelist_by_string(filelist, string_filter)

→ Filters a given list of file paths by a specified substring. Returns the filtered filelist.

- filelist (list): list of full file paths
- string_filter (str): specified substring used to filter a given filelist

filter_filelist_by_stringlist(filelist, list_filter)

→ Filters a given list of file paths by multiple specified substrings. Returns the filtered filelist.

- filelist (list): list of full file paths
- list_filter (list): ['AND'/'OR', [list of substrings]]

get_filelist(file_dir, ext, recursive=False)

→ Returns a list of files of a specified format.

- file_dir (str): local image storage directory
- ext (str): file format
- recursive (bool): list files from subdirectories as well

log(start_time, string="", dim="sek")

→ Displays the time difference from a specified start time and the current time in the console.

- start_time (obj): *time.time()* object of the built in Python module *time*
- string (str): message displayed together with time difference
- dim (str): choose an output format as follows
  'ms' – milli sec; *'mus'* – µ sec; *'MS'* – min:sec; *'HMS'* – hr:min:sec

open_parameters(path)

→ Returns a dictionary of parameters from a configuration file. Supported file formats are JSON and YAML.

- path (str): absolute path to the configuration file

open_project(config_data, config_path)

→ Returns project and image paths of an existing metashape project structure.

- path (str): full path to the project configuration file in target project folder
- config_data (dict): contents of the photogrammetric project configuration file

transfer_images(config_data, dest_dir, recursive=True)

→ Transfers images from a specified source, as defined in the config file, to target directory.

- config_data (dict): contents of the photogrammetric project configuration file
- dest_dir (str): target directory for images to copy
- recursive (bool): collect and copy image files also from subdirectories

class suppress_stdout()

→ Suppresses console output. Is not working for modules based on C or C++ (e.g. Metashape).

Call it as a context manager using the with statement as follows:

with suppress_stdout():
print('Execute your syntax here; no console output will appear!')

2.4.6. DataManagement: File operations

The DataManagement module offers a set of essential file operations to efficiently manage directories and files within a photogrammetric workflow. It supports tasks such as creating, copying, moving, and deleting files and directories. Additionally, it includes MinIO support for interacting with MinIO object storage, enabling seamless data download and upload.

a. class Local()

create_directory(dir_path)

→ Creates a new directory in the target path.

- dir_path (str): full path of the directory to be created

copy_directory(source_path, target_path)

→ Recursively copies a directory and its content from a source location to target directory.

- source_path (str): full path of the directory to be copied
- target_path (str): full path to the destination directory

copy_file(source_path, target_path)

→ Copies a file from a source location to target directory.

- source_path (str): full path of the file to be copied
- target_path (str): full path to the destination directory

copy_filelist(file_list, target_path)

→ Copies all files specified in a list to target directory.

- file_list (list): a list of full file paths to be copied
- target_path (str): full path to the destination directory

get_filelist(file_dir, ext, recursive=False)

→ Returns a list of files of a specified format.

- file_dir (str): local image storage directory
- ext (str): file format
- recursive (bool): list files from subdirectories as well

move_directory(source_path, target_path)

→ Moves a directory and its content from a source location to target destination.

- source_path (str): full path of the directory to be moved
- target_path (str): full path to the destination directory

move_file(source_path, target_path)

→ Moves a file from a source location to target directory.

- source_path (str): full path of the file to be move
- target_path (str): full path to the destination directory

move_filelist(file_list, target_path)

→ Moves all files specified in a list to target directory.

- file_list (list): a list of full file paths to be moved
- target_path (str): full path to the destination directory

remove_directory(dir_path)

→ Removes/deletes a directory and all of its contents.

- dir_path (str): full path of the directory to be removed

remove_file(file_path)

→ Removes/deletes a file from target directory.

- file_path (str): full path of the file to be removed

remove_filelist(file_list)

→ Removes/deletes all files specified in a list from target directory.

- file_list (list): a list of full file paths to be removed

b. class MinIO()

The interaction with MinIO is managed through a configuration file, and an example template can be found in the PyExpress module: *./PyExpress/WorkflowExamples/UserSettings/.* It is crucial to properly adjust the configuration file for MinIO access, particularly for download/upload operations. Dependencies, input types, and keyword hierarchies must be strictly followed, and the file must be formatted in valid Python syntax to ensure optimal functionality and smooth performance! The keywords specified within the configuration file are inherited and accessible as parameters of the MinIO class. This allows seamless integration, where the configuration settings directly influence the behavior of the MinIO interactions without the need for additional manual input.

Class MinIO(config_MinIO, temp_dir="./", get_filelist=True)

→ Creates an instance of the MinIO class, serving as a container for data operations and providing methods for interacting with the MinIO storage system.

Parameters

- config_MinIO (str): full path to the MinIO configuration file
- temp_dir (str): path to a temporary folder for downloaded images
- get_filelist (bool): automatically create a file list based on config file specifications

Public project class methods and their parameters

get_objectlist(client, bucket, prefix, recursive, string_filter=[False, ""], list_filter=[False, "AND", list()])

→ Lists objects/files in a specified MinIO bucket. The object list can be filtered by (a) a specified filename prefix, (b) a specified string within the filename, and/or (c) using a list of specified strings. Returns the filtered object list as specified in the configuration file.

- client (obj): instance of a MinIO client to interact with the service
- bucket (str): name of the MinIO bucket (container for objects)
- prefix (str): starting part of the file path to filter the MinIO object list
- recursive (bool): list files also from subdirectories
- string_filter (list): [bool, specified substring used to filter the MinIO object list]
- list_filter (list): [bool, 'AND'/'OR', list of specified substrings to filter object list]

Examples for filtering options

(a) prefix = 'M2EA/2024'
→ Each filename starting exactly with the specified prefix will be retained.

(b) string_filter = [true, '20_08_2024']
→ If true, each filename containing the specified sequence of character will be retained.

(c) list_filter = [true, 'AND', ['flightA', '20_08_2024', 'M2EA_drone']]
→ If true, each filename containing ALL of the specified strings will be retained
list_filter = [true, 'OR', ['M3T_drone', 'M2EA_drone', 'Mako_camera']]
→ If true, each filename containing at at least ONE specified strings will be retained.

filter_filelist_by_string(filelist, string_filter)

→ Filters a given list of file paths by a specified substring. Returns the filtered filelist.

- filelist (list): list of full file paths
- string_filter (str): specified substring used to filter a given filelist

filter_filelist_by_stringlist(filelist, list_filter)

→ Filters a given list of file paths by multiple specified substrings. Returns the filtered filelist.

- filelist (list): list of full file paths
- list_filter (list): ['AND'/'OR', [list of substrings]]

download_from_minio(as_path=False)

→ Downloads all files listed in the MinIO client class parameter 'filelist' to the path specified in the 'temp_dir' parameter.

- as_path (bool): if True - creates a path from the MinIO filepath with '_' as the separator
  if False - creates the exact directory structure as in the MinIO source bucket

upload_to_minio(filelist, directory)

→ Uploads data to a new object path within MinIO object storage, created dynamically.

- filelist (list): list of full file names
- directory (str): name of target directory, created dynamically within the MinIO bucket

Example 3: Connecting to a MinIO instance and interacting with it.

```
# This example workflow demonstrates how to create a MinIO instance, download a filtered list of images, and then upload point cloud objects after photogrammetric analysis. If get_filelist=True, the MinIO objectlist will be filtered automatically.

import PyExpress

cloudData = PyExpress.DataManagement.MinIO(config_MinIO  = 'path_to_yaml_file.yaml',
                                           temp_dir      = 'directory_to_download_images',
                                           get_filelist  = False)

cloudData.filelist =   cloudData.get_objectlist(client        = cloudData.client,
                                                bucket        = cloudData.bucket,
                                                prefix        = cloudData.prefix,
                                                recursive     = cloudData.recursive,
                                                string_filter = cloudData.str_filter,
                                                list_filter   = cloudData.list_filter)

cloudData.download_from_minio()

*** some photogrammetric analysis steps ***

filelist_upload = PyExpress.DataManagement.get_filelist(file_dir='path_to_pointclouds', ext='obj')

clouData.upload_to_minio(filelist = filelist_upload, directory = 'POINTCLOUDS')
```

List of accessible MinIO project class internal objects

- acc_key (str): access key for the MinIO API as specified in the MinIO configuration file
- bucket (str): name of the MinIO bucket
- client (obj): instance of the MinIO container
- conn_info (str): connection gateway as specified in the MinIO configuration file
- filelist (list): list of files with specified prefix stored in the MinIO bucket
- list_filter (list): list for filtering the filelist as specified in the MinIO config file
- port_API (str): port for API access
- prefix (str): starting part of the file path to filter the MinIO object list
- recursive (bool): whether to list files also from subdirectories
- region_API (str): region for API access
- sec_key (str): secret key for the MinIO API
- server (str): server address for MinIO connection
- str_filter (list): string for filtering the filelist as specified in the MinIO config file
- temp_dir (str): target directory for downloading images from MinIO storage
- url (str): complete URL for connection

2.5. Future plans

The PyExpress module is considered as a dynamic framework, allowing for flexible extension and enhancement based on additional (photogrammetric) workflows. Features under development are the following (refer to Fig. 2):

- **Integration of an interface to CloudCompare**, a 3D point cloud processing software offering tools for editing 3D point clouds and triangular meshes. CloudCompare also includes advanced analytical capabilities, such as cloud-to-cloud distance computation, statistical analyses, and geometric feature estimation.
- **Methods for quality assessment of image files**, incorporating machine learning techniques for image preselection via classification (e.g., using Random Forests) and preprocessing functionalities. These include histogram matching and adjustments to illumination and contrast implemented through point operations using OpenCV and Scikit-image libraries.
- **Incorporation of a Secure File Transfer Protocol (SFTP) service** for efficient online data transfer.
- **Integration of cloud computing functionality** into PyExpress workflows.

3. Workflow example on a test data set

The PyExpress v1.0 module is accompanied by a photogrammetric workflow example available on GitHub in the *TestPipeline* directory. This example allows users to evaluate the performance of an automated photogrammetric workflow, combining routines adapted from Metashape with custom processes using the PyExpress package. The *TestPipeline* directory contains:

- a main Python control script *TestWorkflowMAIN.py*
- a dataset with 125 airborne image files (*data/images*)
- marker image projections data (*./GCP/marker_projections.csv*)
- marker real-world coordinates (*./GCP/marker_reference.csv*.)
- configuration files *TEST_config_NEW.yaml* and *TEST_config_EXI.yaml*

This setup provides a comprehensive framework for testing and refining an automated photogrammetric pipeline using PyExpress. It turned out that it is suitable to follow the following steps when setting up a project or application-specific individual workflow:

(a) Configuration file: Adjusting according to specific project needs

(b) Photogrammetric image analysis: Setting up an individual, Metashape-based workflow

(c) Writing a global control script for the automated extensive data pipeline

3.1. Configuration file adjustment

The initial step involves reviewing and adjusting the photogrammetric workflow configuration, managed via the configuration file. The provided configuration files, *TEST_config_NEW.yaml* and *TEST_config_EXI.yaml*, serve as examples for distinct project scenarios. The first file defines the setup for initializing a new project using the drone data, while the second file is tailored for reopening and continuing work on an existing project. Both configuration files are illustrated in Figures 4, 5, and 6 and a comparison modus. For more detailed information, please refer to the extensioned comments within the graphics. Both provided configuration files are already tailored to the requirements of the test project. **NOTE here**: Be sure to set the *input|general|new_project* parameter in your configuration file according to your project status, **as setting it to *true* will overwrite the project if it already exists!**

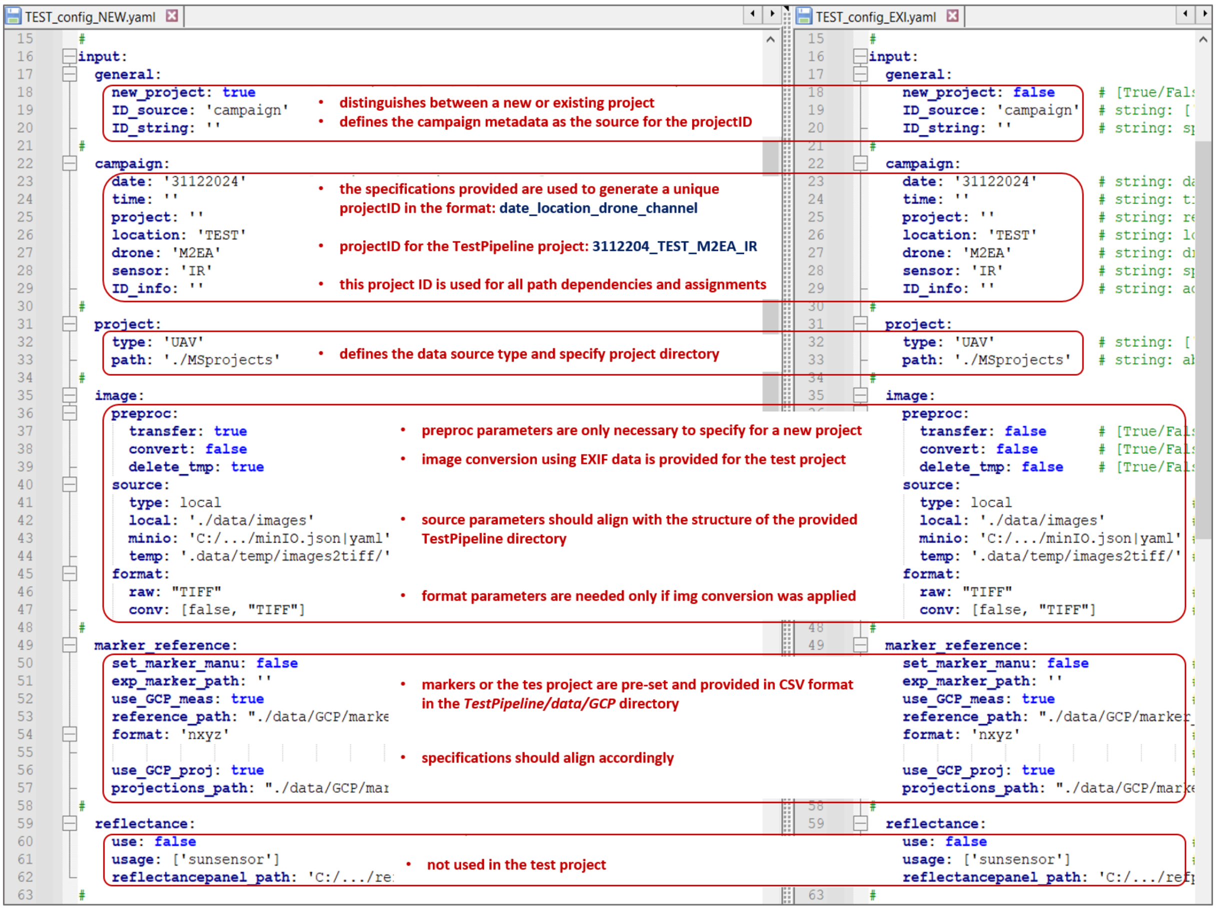


Abbildung 4: Comparison of the input section in the provided example configuration files for a new and an existing project, including explanatory comments.

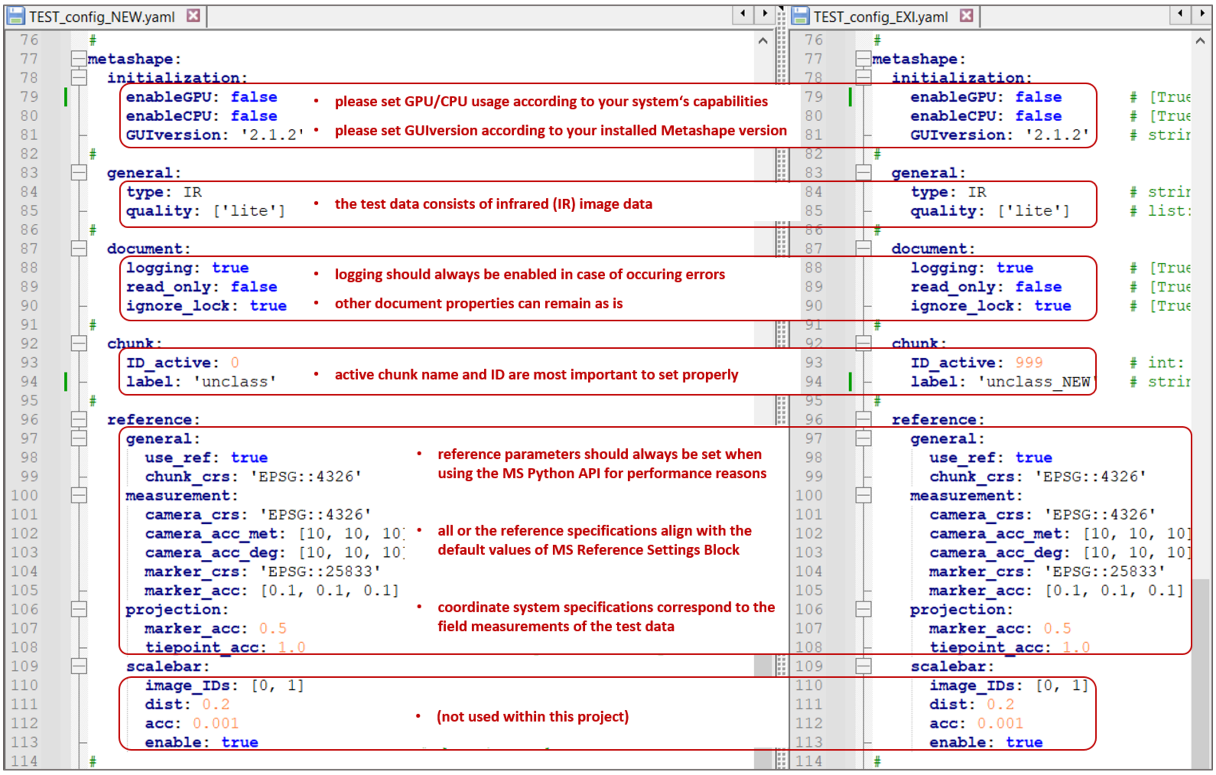


Abbildung 5: Comparison of the Metashape document settings in the example configuration files for a new and an existing project, including explanatory comments.

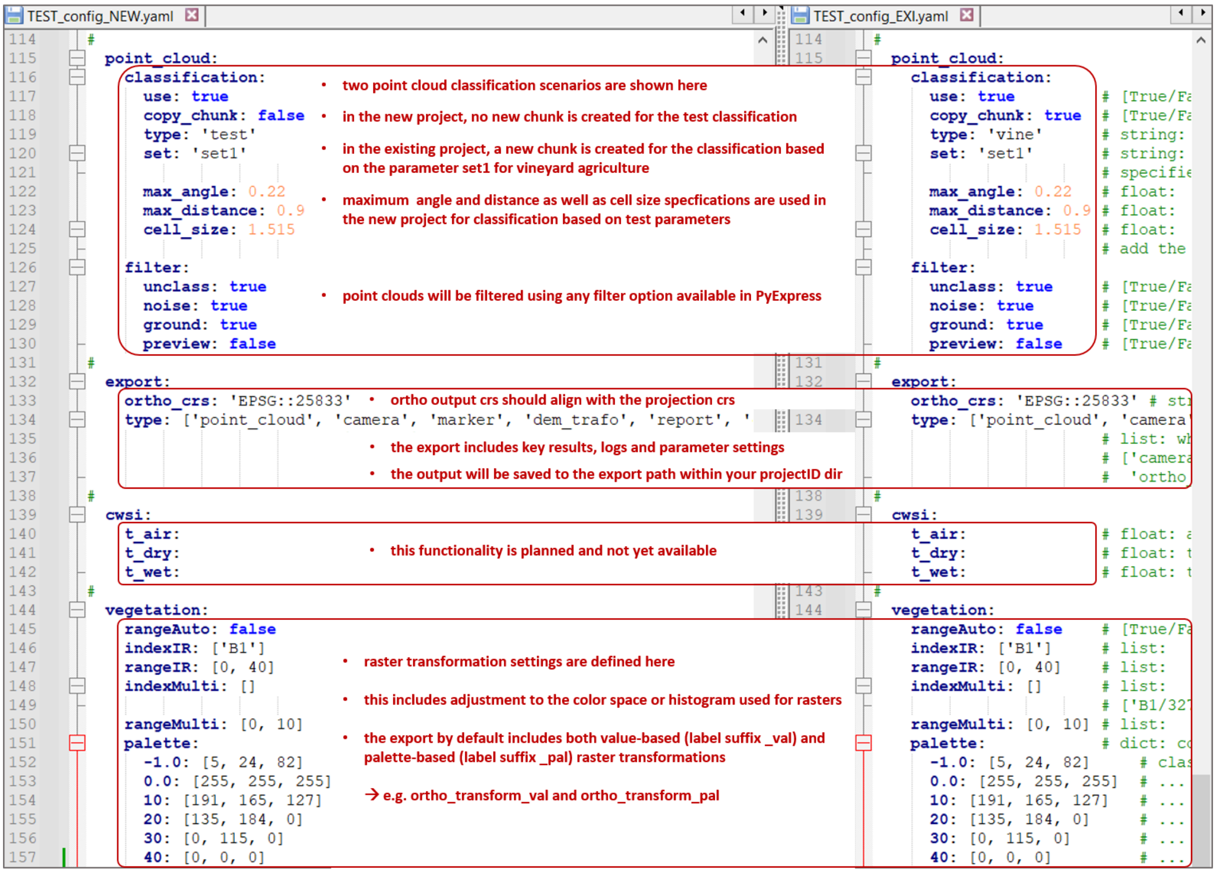


Abbildung 6: Comparison of the Metashape workflow settings in the example configuration files for a new project and an existing project, including explanatory comments.

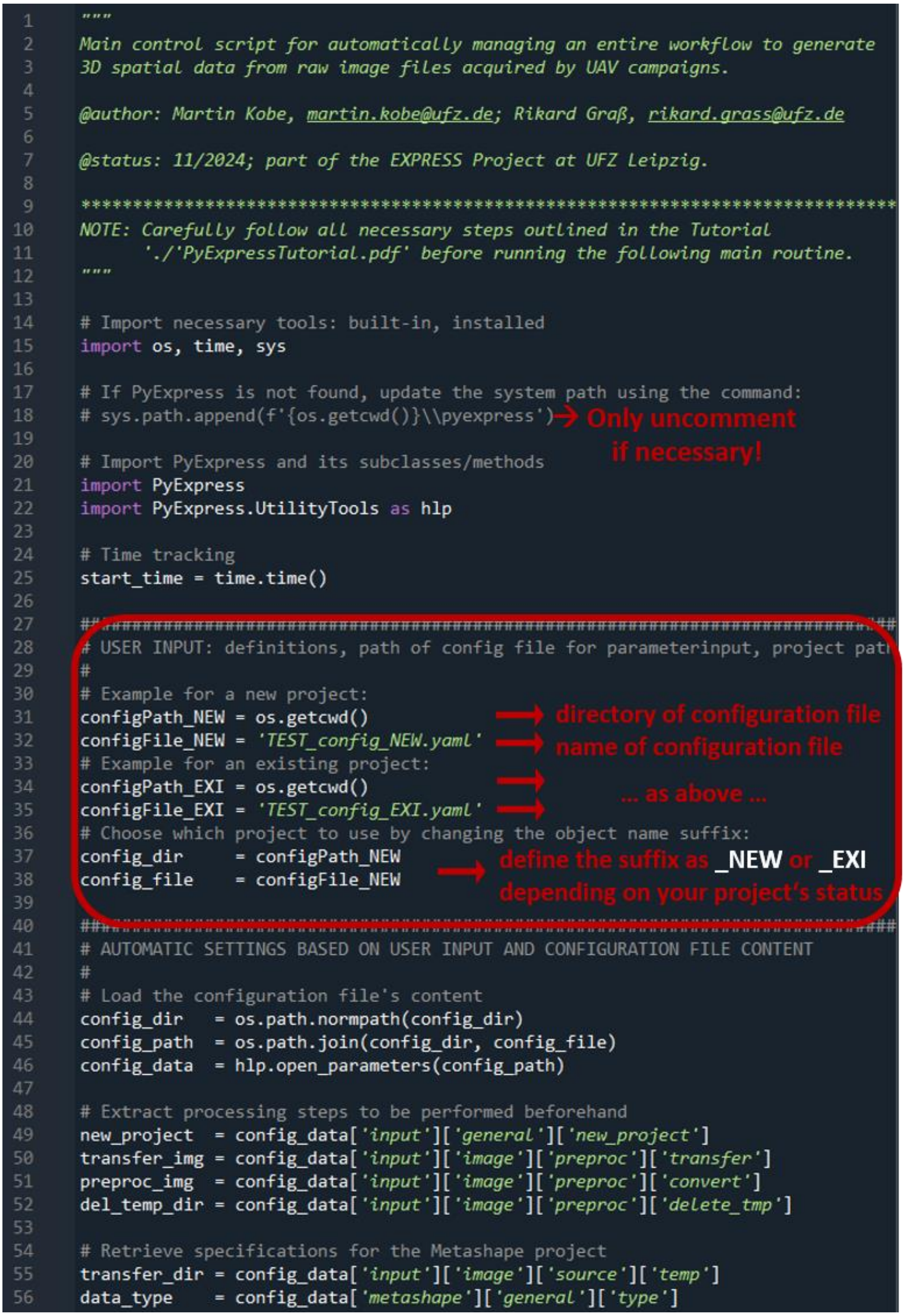
3.2. Photogrammetric Metashape-based image analysis

The Metashape workflow presented here includes the core Metashape workflow methods (cf. numbers) and optional techniques (cf. Opt) described earlier in this tutorial. The functionalities are tailored to the project, the drone used, and the specific outcome. For detailed information on the specific workflow steps, the extensively commented script containing the *TEST_workflow class* can be found at the path *PyExpress/WorkflowExamples/DroneProject/TEST_workflow.py.* For instruction on calling and feeding the specific PyExpress functions, see the examples in the Sections 2.4.2. and 2.4.3.! In summary, the workflow consits of the following steps:

(Prep)	Extract parameters for the photogrammetric workflow
(0)	Create an instance of a drone project (cf. Section 2.4.2)
(1)	Add Photos into the active chunk
(2)	Match Photos
(3)	Align cameras
(Opt)	Marker specifications
(Opt)	Reference specifications
(4)	Build depth map
(5)	Build point cloud
(Opt)	Classification of the point cloud within a new added chunk
(6)	Generate 3D Model
(7)	Generate Digital Elevation Model (DEM)
(8)	Build UV mapping for the model
(9)	Create orthorectified projection
(Opt)	Apply specifications for raster transformation (DEM, ortho mosaic)
(Opt)	Export project results
(Finally)	Return the final project instance

3.3. Main control script

The main script *TestWorkflowMain.py* calls and globally integrates the specific photogrammetric workflow within a broader working environment, which includes basic functionalities for data management, data transfer, as well as the organization of project structure and parameters, and can be enhanced as needed. It specifically distinguishes between two options: new and existing projects. At this point, the user has one key task: defining the paths to the directories where the configuration files are located, and specifying the name of a particular configuration file. For clarity and easier management, the paths in the test example are differentiated by their suffix – the user needs to specify the suffix of the object (_NEW or _EXI). For detailed information and illustration, please refer to Figures 7 and 8, which include additional comments. For testing and verification purposes, it is recommended to execute the main control script twice: first to create a new project and second to reopen the existing one.



```
"""
Main control script for automatically managing an entire workflow to generate
3D spatial data from raw image files acquired by UAV campaigns.

@author: Martin Kobe, martin.kobe@ufz.de; Rikard Graß, rikard.grass@ufz.de

@status: 11/2024; part of the EXPRESS Project at UFZ Leipzig.

*******************************************************************************
NOTE: Carefully follow all necessary steps outlined in the Tutorial
      './'PyExpressTutorial.pdf' before running the following main routine.
"""

# Import necessary tools: built-in, installed
import os, time, sys

# If PyExpress is not found, update the system path using the command:
# sys.path.append(f'{os.getcwd()}\\pyexpress')

# Import PyExpress and its subclasses/methods
import PyExpress
import PyExpress.UtilityTools as hlp

# Time tracking
start_time = time.time()

###############################################################################
# USER INPUT: definitions, path of config file for parameterinput, project pat
#
# Example for a new project:
configPath_NEW = os.getcwd()
configFile_NEW = 'TEST_config_NEW.yaml'
# Example for an existing project:
configPath_EXI = os.getcwd()
configFile_EXI = 'TEST_config_EXI.yaml'
# Choose which project to use by changing the object name suffix:
config_dir      = configPath_NEW
config_file     = configFile_NEW

###############################################################################
# AUTOMATIC SETTINGS BASED ON USER INPUT AND CONFIGURATION FILE CONTENT
#
# Load the configuration file's content
config_dir    = os.path.normpath(config_dir)
config_path   = os.path.join(config_dir, config_file)
config_data   = hlp.open_parameters(config_path)

# Extract processing steps to be performed beforehand
new_project   = config_data['input']['general']['new_project']
transfer_img = config_data['input']['image']['preproc']['transfer']
preproc_img   = config_data['input']['image']['preproc']['convert']
del_temp_dir = config_data['input']['image']['preproc']['delete_tmp']

# Retrieve specifications for the Metashape project
transfer_dir = config_data['input']['image']['source']['temp']
data_type     = config_data['metashape']['general']['type']
```


Figure 7: First part of the main control script TestWorkflowMAIN.py showing the important user input for specifying the project configuration file path and name.

```
################################################################################
# NEW PROJECT: Example workflow for photogrammetric data analysis

if new_project == True:

# a) Create a basic project directory structure
    img_dir, prj_dir = hlp.create_UAV_project(config_data = config_data,
                                              config_path = config_path)

# b) Transfer images from various sources (local, minIO) to either
#     - project image folder (no preprocessing) or
#     - temp image folder (preprocessing required)
#    NOTE: In this test example, the project image folder is defined as default
    if transfer_img == True:
        if data_type == 'IR':  transfer_dir = img_dir                        # (!)
        if data_type == 'RGB': transfer_dir = img_dir                        # (!)

        hlp.transfer_images(config_data = config_data,
                            dest_dir    = transfer_dir,
                            recursive   = True)

# c) Preprocess raw image data based on user and project requirements
#    NOTE: In this test example, already preprocessed images are used/provided
    pass

# d) Run example workflow for a UAV-based vegetation monitoring project
    MultiProject = PyExpress.WorkflowExamples.TEST_workflow(config_data = config_data,
                                                            prj_dir     = prj_dir,
                                                            img_dir     = img_dir,
                                                            config_name = config_file)

# e) Optionally delete log file after processing
    if input('Delete lock file [y,n]?: ') == 'y': MultiProject.doc.clear()

################################################################################
# EXISTING PROJECT: Resume or recalculate photogrammetric data analysis

if new_project == False:

# a) Open a basic existing project directory structure
    img_dir, prj_dir = hlp.open_project(config_data = config_data,
                                        config_path = config_path)

# b) Run example workflow for a UAV-based vegetation monitoring project
    MultiProject = PyExpress.WorkflowExamples.TEST_workflow(config_data = config_data,
                                                            prj_dir     = prj_dir,
                                                            img_dir     = img_dir,
                                                            config_name = config_file)

# c) Optionally delete log file after processing
    if input('Delete lock file [y,n]?: ') == 'y': MultiProject.doc.clear()

################################################################################
# Execution time log
hlp.log(start_time=start_time, string='\nAutomated pipeline execution time', dim='HMS')
```

Abbildung 8: Second part of the main control script *TestWorkflowMAIN.py* showing all the workflow steps from raw UAV data to key results export within the TEST_workflow routine.

3.4. Directory / project structure created

After successfully executing the main control script for an automated pipeline, which processes raw image data into photogrammetric results, a structured hierarchy of folders and files is generated in the specified result path. This structure, conceptually depicted in Figure 9, organizes all created elements of the workflow. Objects labeled in blue are dynamically adjusted based on the parameters defined in the configuration file or specific requirements of the photogrammetric workflow. This ensures clear navigation through the results, facilitating their reuse in future processing steps.

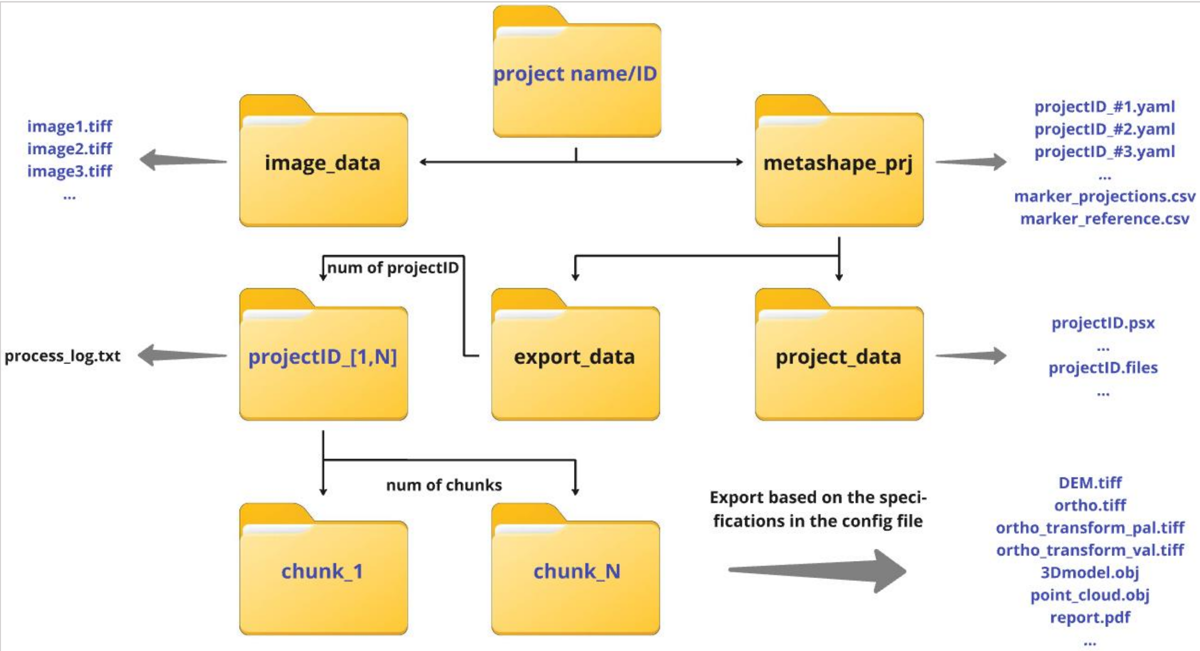


Abbildung 9: Automatically generated folder and file structure after executing the TestWorkflowMain.py *script. Labels shown in blue are dynamically adjusted according to the specifications in the configuration file or the photogrammetric workflow.*

Literaturverzeichnis

Agisoft, 2024. Agisoft Metashape: Discover intelligent photogrammetry [WWW Document]. URL: https://www.agisoft.com/ (last accessed: 12.02.2025).

Axelsson, P., 2000. International Archives of Photogrammetry and Remote Sensing. Vol. XXXIII, Part B4. Amsterdam 2000. Int. Arch. Photogramm. Remote Sens. 33, 110–117.

DJI, 2024. DJI - Offizielle Webseite [WWW Document]. URL: https://www.dji.com/de (last accessed: 12.02.2025).

James, M.R., Robson, S., Smith, M.W., 2017. 3-D uncertainty-based topographic change detection with structure-from-motion photogrammetry: precision maps for ground control and directly georeferenced surveys. Earth Surf. Process. Landf. 42, 1769–1788. https://doi.org/10.1002/esp.4125

Klápště, P., Fogl, M., Barták, V., Gdulová, K., Urban, R., Moudrý, V., 2020. Sensitivity analysis of parameters and contrasting performance of ground filtering algorithms with UAV photogrammetry-based and LiDAR point clouds. Int. J. Digit. Earth 13, 1672–1694. https://doi.org/10.1080/17538947.2020.1791267

Kobe, M., Elias, M., Merbach, I., Schädler, M., Bumberger, J., Pause, M., Mollenhauer, H., 2024. Automated Workflow for High-Resolution 4D Vegetation Monitoring Using Stereo Vision. Remote Sens. 16, 541. https://doi.org/10.3390/rs16030541

MinIO, Inc, 2024. MinIO | Enterprise Grade, High Performance Object Storage [WWW Document]. MinIO. URL: https://min.io (last accessed: 12.02.2025).