Image Matching Challenge

1. Overview

Problem Definition

The goal is to construct precise 3D maps using sets of images in diverse scenarios and environments. The solution is expected to generate accurate spatial representations, regardless of the source domain—images taken from drones, amidst dense forests, during nighttime, or any of the six problem categories.

Competition webpage

https://www.kaggle.com/competitions/image-matching-challenge-2024/overview

Metric

The metric is **mean Average Accuracy** (mAA) of the registered camera centers.
Detailed information is available in *Section 3. Metric* and at the completion webpage:
https://www.kaggle.com/competitions/image-matching-challenge-2024/overview/evaluation

Data

The dataset consists of several scenes, each scene containing a varying number of images. Each image is represented by camera settings: rotation matrix and translation vector.

train_labels.csv

A list of all images with ground truth information about the camera rotation and translation.

Images

All images belonging to a single scene of the dataset are stored in folders `<scene>/images`.

Strategy for solving the problem

For each scene:

1. Identify pairs of similar images.
2. Detect and match keypoints for all registered pairs of images.
3. Use colmap library to reconstruct the scene's cameras.

2. Training Data

Source code

`01_data.ipynb` – jupyter notebook with overview of the dataset and target.

Images

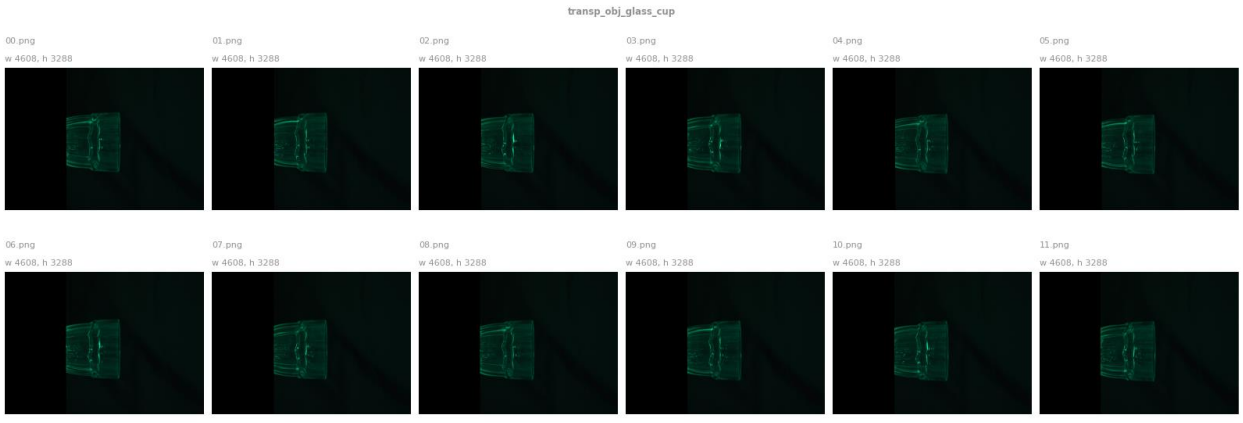
Insights

- Different categories of scenes: transparent objects, modern architecture, historical architecture, nature.
- Varying number of images in scenes:

scene	*number of images*
church	110
dioscuri	70
lizard	711
multi-temporal-temple-baalshamin	68
pond	1117
transp_obj_glass_cup	36
transp_obj_glass_cylinder	36

- Images from the same scene vary in quality, image size, orientation.
- Lighting conditions may vary (day/night).

Examples



dioscuri

img_0311.png
w 1024, h 768

img_0119.png
w 1024, h 768

3dom_fbk_img_1542.png
w 1024, h 683

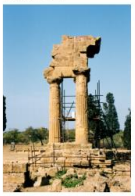
img_0391.png
w 1024, h 768

archive_0001.png
w 1024, h 700

archive_0013.png
w 700, h 1024

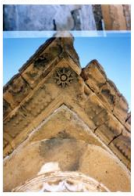
archive_0230.png
w 701, h 1024

archive_0003.png
w 700, h 1024



archive_0010.png
w 1024, h 700

archive_0025.png
w 700, h 1024



archive_0015.png
w 700, h 1024

3dom_fbk_img_1582.png
w 1024, h 683

pond

00992.png
w 576, h 1024

00891.png
w 576, h 1024

00906.png
w 576, h 1024

00242.png
w 1024, h 768

00295.png
w 1024, h 768

00252.png
w 1024, h 768

00884.png
w 576, h 1024

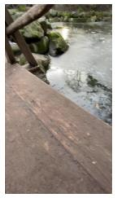
00543.png
w 576, h 1024

00184.png
w 768, h 1024

00379.png
w 576, h 1024

00428.png
w 576, h 1024

00793.png
w 576, h 1024



church

00100.png
w 768, h 1024

00108.png
w 768, h 1024

00047.png
w 768, h 1024

00105.png
w 1024, h 768

00024.png
w 768, h 1024

00111.png
w 768, h 1024

00056.png
w 768, h 1024

00073.png
w 768, h 1024

00003.png
w 768, h 1024

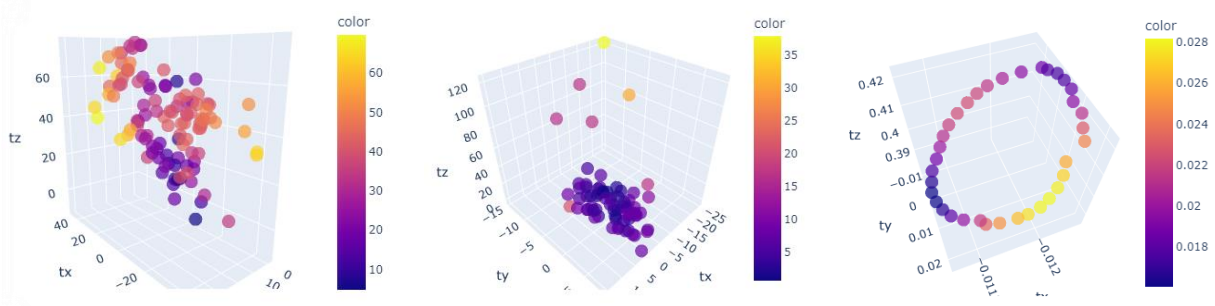
00013.png
w 768, h 1024

00053.png
w 768, h 1024

00044.png
w 768, h 1024

Target

Example of ground truth translation vectors for a scene are shown below. Colored circles represent the end of the vector (with the start of the vector at coordinate (0,0,0)); color denotes the length of the vector:



For an interactive visualization of rotation matrices (visualization of rotation vectors around each axis) refer to the `01_data.ipynb` notebook.

Note: there is some trouble with rendering plotly charts in the GitHub copy of the notebook. The notebook needs to be rerun for rendering 3D charts.

3. Metric

Source code

`imc24.py` – metric source code, copy of the public Kaggle notebook https://www.kaggle.com/code/nartaa/imc24.

`02_metric.ipynb` – a jupyter notebook with exploration of the metric.

Overview

The metric is **mean Average Accuracy** (mAA) of the registered camera centers across all images of all scenes of the dataset.

Camera centers are characterized by a matrix $C = -R^T T$, where R is a 3×3 rotation matrix and T is a translation vector.

For each image of the scene, a camera with computed center C is registered if

$$\|C_g - \mathcal{T}(C)\| < t,$$

where C_g is the ground-truth camera center, t is a given threshold, and $\mathcal{T}$ is the best similarity transformation (scale, rotation and translation) that is able to register onto the ground-truth the highest number of cameras for the scene.

The best similarity transformation $\mathcal{T}$ is obtained by a 2-step process:

1. Using a RANSAC-like approach, exhaustively verify all feasible similarity transformations $\mathcal{T}'$ that can be derived by the Horn's method on triplets of corresponding camera centers (C, C_g);

2. Each transformation $\mathcal{T}'$ is further refined into $\mathcal{T}''$ by registering again the camera centers using the Horn's method, but this time including the previously registered cameras together with the initial triplets. The transformation with the highest number of registered cameras is considered the best.

Insights

For each scene:

- Invariance to translation vectors' scale – only relative measurements matter;
- Invariance to rotation of rotation matrices.

Smallest noise in registered cameras affects the score: for ground truth predictions with additive noise level 1% in either rotation matrices or translation vectors → the average mAA score is ~0.4.

4. Solution

Solution pipeline

Source code

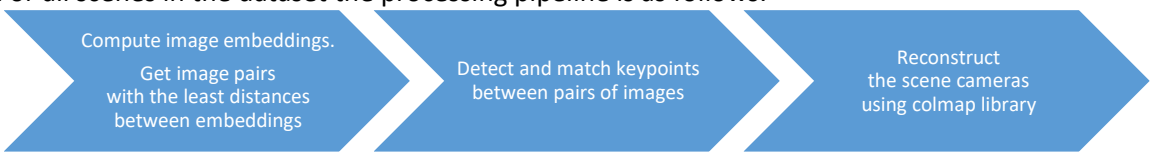
`IMC_solution.py` – solution class for processing all scenes in the dataset and logging the results using Weights&Biases.

`IMC_pipeline.py` – class for the processing pipeline of a single scene.

`IMC_utils.py` – utility functions for the solution.

Overview

For all scenes in the dataset the processing pipeline is as follows:



Each stage is represented in code by a class.

Image embedding and pair matching

Source code

`03_image_pairs.ipynb` – jupyter notebook with examples of matched image pairs.

`image_pairs_matcher.py` – class for computing and ranking distances between image embeddings.

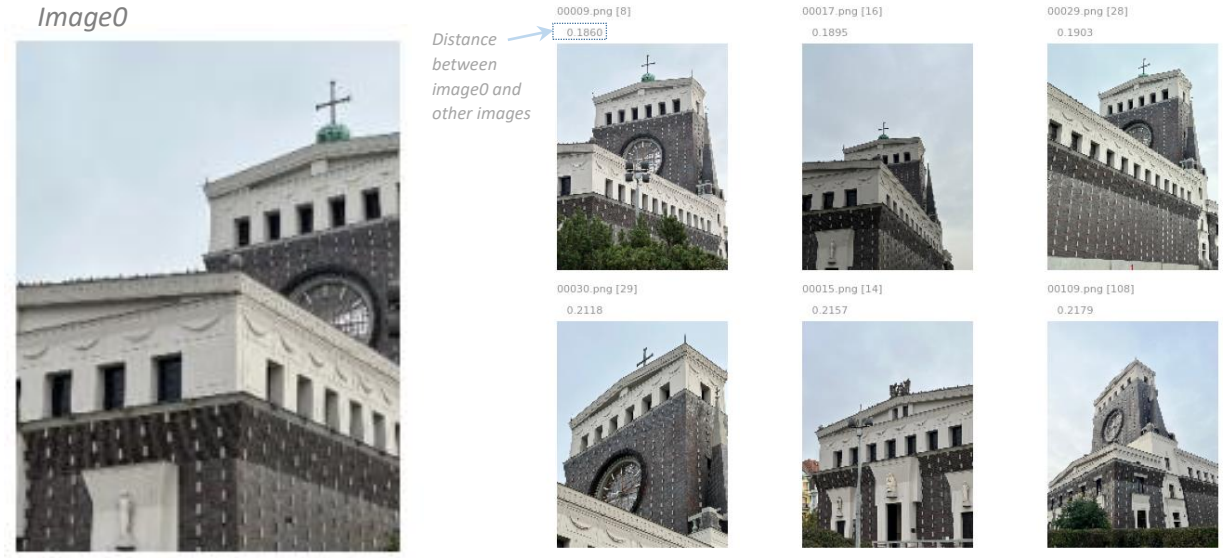
Overview

Images of a scene cannot be compared directly. Pretrained DINOv2 model is used to obtain normalized image embeddings, then the Euclidean distances between the embeddings are computed for all possible pairs. Image pairs are then sorted according to the computed distance. Pairs with the least distances are then passed to the next stage of the processing pipeline.

Note: Image embeddings are precomputed outside of the solution pipeline and saved in an `h5` file. The pipeline operates under the assumption that the embedding file exists; otherwise, the scene is not processed.

Results

Example of computed distances between images for the `'church'` scene:



Example of computed distances between images for the `'dioscuri'` scene:

Keypoint detection and matching

Source code

`04_display_keypoint_matches.ipynb` – jupyter notebook with examples of matched image keypoints.

`keypoints_matcher_base.py` – base abstract class `KeypointsMatcherBase` for all keypoints matching methods.

`keypoints_LightGlue.py` – wrapper class `Keypoints_LightGlue` for matching keypoints using the LightGlue model.

`keypoints_cv2.py` – wrapper class `Keypoints_cv2` for brute force keypoint matching method from the OpenCV library.

`keypoints_LoFTR.py` – wrapper class `Keypoints_LoFTR` for the LoFTR keypoint matching method.

`parser_matched_keypoints.py` – class for preparing registered keypoint matches for the scene reconstruction step of the pipeline.

KeypointsMatcherBase

`KeypointsMatcherBase` is an abstract class that provides a uniform interface for all keypoint matching methods.

The main method `run` takes in an array of scene image paths and an array of image index pairs from the previous step of the pipeline. For all image pairs the matching is performed using the method `_match_image_pair_with_rotation`.

The abstract method `_match_image_pair` returns an array of matched coordinates of the keypoints of the image pair and is different for all implemented methods.

Image rotation

Keypoint matching methods are sensitive to orientations of the images, which impacts the reconstruction of the scene's cameras. In this project, the following approach is used for dealing with varying orientations of the images:

1. Assume the rotation of the first image `image1` to be 0°.
2. For all four rotations 0°, 90°, 180°, 270° of the second image `image2`, match keypoints of `image1` and `image2` (downsized x4 for speed). If one of the four pairs produces significantly more matches than the other 3, this orientation `R` is considered correct for the `image2`; otherwise assume the correct orientation is 0°.
3. Keypoints are matched for full-sized `image1` and correctly rotated full-sized `image2`. The coordinates of detected image2 keypoints are then inversely rotated to match the initial orientation of `image2`.

One of the scenes in the training dataset contains images with varying orientations. By using correctly rotated `image2` the score for the `'dioscuri'` scene changes from 0.1 to 0.4 without changing other parameters of the pipeline.

Keypoints_LightGlue

`Keypoints_LightGlue` class performs keypoint matching using LightGlue method with a choice of keypoint extraction methods:

1. ALIKED,
2. SuperPoint,
3. DoGHardNet,
4. DISK,
5. SIFT.

Keypoints_LoFTR

`Keypoints_LoFTR` class performs keypoint matching using LoFTR method.

Keypoints_cv2

`Keypoints_cv2` class performs keypoint matching using OpenCV library's `BFMatcher` method with a choice of keypoint extraction methods:

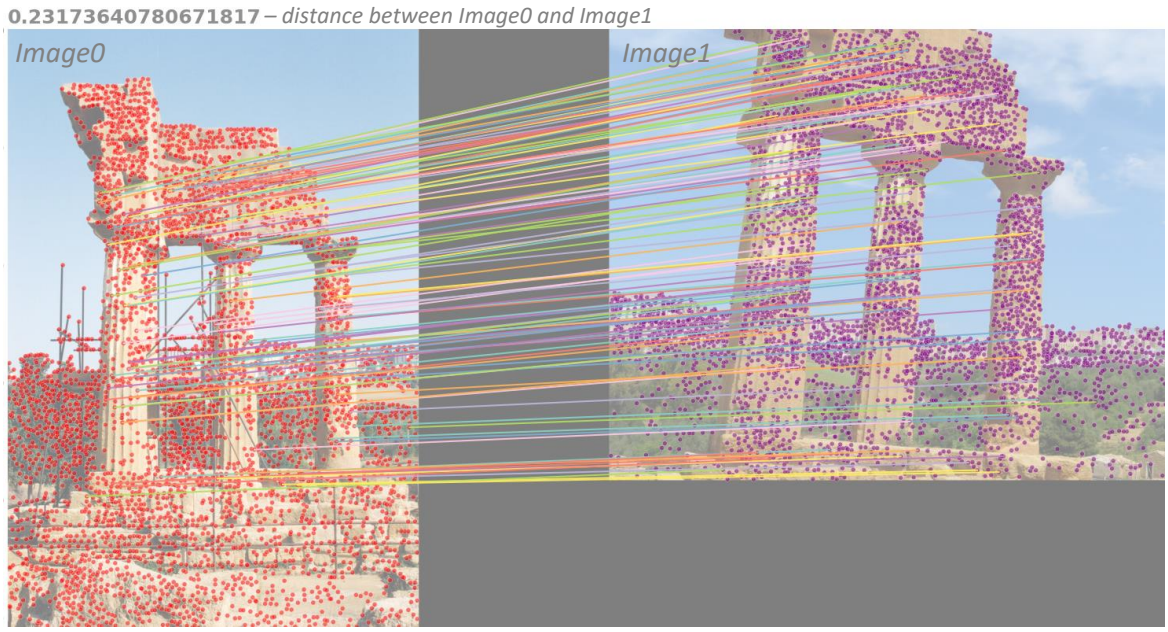
1. ORB,
2. AKAZE.

Results

Overall, LightGlue detects significantly more keypoints than LoFTR and cv2's BFMatcher. Examples of image pairs with matched keypoints are presented below. For a detailed comparison of different methods refer to the section 5. Results.

Example of matched keypoints for the `'church'` scene:

Example of matched keypoints for the `'dioscuri'` scene:



Scene reconstruction

Source code

`scene_reconstructor.py` – a class for reconstructing scene's cameras from matched keypoints.

`database.py`, `h5_to_db.py` – utility functions for importing keypoints into COLMAP-compatible database (provided by the competition organizers).

Overview

`SceneReconstructor` class provides functionality for importing files `'keypoints.h5'` and `'matches.h5'` from the previous step of the pipeline into a COLMAP database and processing the scene using the `incremental_mapping` method from the pycolmap library.

Reproducibility of the reconstruction is ensured by explicitly setting the number of processing threads to 1.

The result of this step is considered to be the reconstruction with the highest number of registered images.

5. Results

Source code

`05_run_solution.ipynb` – a jupyter notebook for running the main solution loop.
`06_results.ipynb` – a jupyter notebook for comparing results of all experiments.

Summary

Scenes 'lizard' and 'pond' were not processed due to the high number of images in these scenes, which is too computationally heavy for the available hardware.

Top scores for each processed scene:

scene	*score*	*pairs threshold*	*pairs min number*	*extractor*	*matcher*	*resize to*	*match threshold*
transp_obj_glass_cylinder	**0.005051**	0.3	5	aliked	LightGlue	1024	0.01
	0.000000	0.1	30	disk	LightGlue	1024	0.01
transp_obj_glass_cup	**0.035354**	0.1	30	LoFTR	LoFTR	512	0.4
	0.025253	0.1	30	disk	LightGlue	1024	0.01
multi-temporal-temple-baalshamin	**0.241026**	0.1	30	disk	LightGlue	768	0.3
	0.235897	0.1	30	disk	LightGlue	1024	0.01
dioscuri	**0.405473**	0.1	30	disk	LightGlue	768	0.3
	0.398010	0.1	30	disk	LightGlue	1024	0.01
church	**0.285047**	0.1	30	disk	LightGlue	1024	0.01
	0.285047	0.1	30	disk	LightGlue	768	0.01

Insights

- The best keypoint matching method is LightGlue with keypoint extraction method DISK.
- All of the methods failed to correctly match keypoints of transparent objects.
- Detecting relative rotation of image pairs is very important for successfully matching keypoints.

6. Approaches that didn't work

Automatic image orientation detection

Source code

`07_rotations.ipynb` – a jupyter notebook for automatic rotation detection.

Summary

Random examples of using a pretrained model from the library `check_orientation` show that the detected orientation can be faulty:

✗ Detected orientation 0° (correct 270°)

✗ Detected orientation 90° (correct 0°)

✓ Detected orientation 0°