COMP24011 Lab 4:
Features for Estimating Autonomous Vehicle Poses

Riza Batista-Navarro
Francisco Lobo

Academic session: 2022-23

Git Tag: `24011-lab4-S-SLAM`

Preparatory Work

For this exercise, we will use the Python bindings for OpenCV, a very popular computer vision framework. As a first step, install the library in your environment by issuing the following command

```
$ pip install opencv-contrib-python
```

You can find more information about this library at `https://github.com/opencv/opencv`
Important: The lab code depends on the Scale-Invariant Feature Transform (SIFT) functionality of OpenCV, which has changed in recent years due to patent issues[1]. For this reason you must ensure that you install at least version 4.4.0 of the library. This should happen by default on the Kilburn lab machines.

This lab exercise will use a simplified version the KITTI data set, which you need to download from `https://www.dropbox.com/s/w4m0iam8kgqj262/MyKITTI.zip`

The original dataset[2] used in the 2012 SLAM Evaluation challenge consists of 4541 images; for this exercise we will use only its first 501 images to reduce running time. Nevertheless, the data set occupies close to 150 MB and so **cannot be included** in your GitLab repo. For this reason, you will need to extract it in another directory.
Important: The examples in this manual assume that you extract `MyKITTI.zip` in your home directory, and hence that the path to the data set is `~/MyKITTI`

Introduction to the Problem

In this exercise, you will implement different strategies for matching visual features in a series of images captured during the navigation of an autonomous vehicle (AV). These features are used in estimating poses (i.e., camera trajectories) based on a visual odometry algorithm. Below are some terms and their definitions to help clarify some concepts in autonomous robot navigation.

Odometry is the use of sensors to estimate a robot's change in position relative to a known position. *Visual odometry (VO)* is a specific type of odometry where only cameras are used as sensors, as opposed to using, e.g., global positioning system (GPS) sensors or light detection and ranging (LIDAR) sensors. It is based on the analysis of a sequence of camera images.

[1] `https://github.com/opencv/opencv/issues/16736`
[2] `http://www.cvlibs.net/datasets/kitti/eval_odometry.php`

Simultaneous localisation and mapping (SLAM) is a task whereby a robot needs to build a map of its current environment while at the same time trying to determine its position relative to that map. In this exercise, you will explore a monocular (single-camera) VO solution to the 2012 SLAM Evaluation challenge, which made use of the KITTI data set. However, it is worth noting that VO is limited in that it can only perform trajectory estimation after each pose, and hence trajectory optimisation is achieved only locally. Global optimisation is achieved through loop closure: the correction of the trajectory upon revisiting an already encountered location.

The monocular VO system provided to you (via Gitlab) is already fully functional[3]. It uses feature tracking to identify feature correspondences between adjacent images. Your only task for this exercise is to allow for inspecting feature matches by implementing the following **feature matching strategies**:

(1) distance thresholding,

(2) nearest neighbour, and

(3) nearest neighbour distance ratio.

Running the Visual Odometry System

In your GitLab repo you will find the following Python files:

`visual_odometry.py`	implementation of the visual odometry algorithm,
`run_VO.py`	runs the visual odometry algorithm on a given image dataset,
`algorithms.py`	implementation of feature matching strategies that are used in `visual_odometry.py` which you need to code.

The visual odometry system can be run in the following two ways.

A Viewing the vehicle's trajectory (without any feature matching)

If you just wish to visualise the trajectory of the autonomous vehicle, the following command will run the VO system without returning any feature matches. You should see the trajectory and camera view in real time.

```
$ python3 run_VO.py view_trajectory <dataset_path>
```

Here, `<dataset_path>` is the **path** to the KITTI data set that you decompressed as part of the Preparatory Work described above. You do not need to quote the data set path as a Python string. For example, you can run the command:

```
$ python3 run_VO.py view_trajectory ~/MyKITTI
```

This will open a window that sequentially displays all 501 images (with frame IDs 0 to 500) in the KITTI data set you downloaded earlier, and will show (in another window) the true trajectory (drawn in green) and the estimated trajectory (drawn in red) of the vehicle.

[3]Attribution: The implementation provided to you builds upon another library available at `https://github.com/uoip/monoVO-python`. However, you cannot use that because it does not have the added functionalities specific for this exercise; it also requires Python 2.

B Using visual odometry with feature matching enabled

To obtain feature matches based on a specified strategy, run the following command.

```
$ python3 run_VO.py drive_car <dataset_path> <frame_id> \
    <matching_algo> <threshold> <feature_number> <visualise>
```

Running the above command will display the sequence of images but only until the specified frame ID; when that frame is reached, the code will:

(1) compute feature matches between that frame's image I_k and the image preceding it I_{k-1};

(2) return those feature matches represented as a list of floats corresponding to Euclidean distance values.

For this command, the mandatory arguments enclosed in '<>' should hold the following values:

`<dataset_path>` A string representing the **path** to the KITTI dataset.

`<frame_id>` An integer giving the ID of the frame for which to compute feature matches; possible range is $1, \ldots, 500$.

`<matching_algo>` One of three integer values specifying the feature matching strategy to apply; these can be

- `1` to use your *distance thresholding* function `dist_thresholding`
- `2` to use your *nearest neighbours* function `nn`, and
- `3` to use your *nearest neighbours distance ratio* function `nndr`.

`<threshold>` A float giving the threshold value that will be used by the `<matching_algo>`; the possible range is as follows:

- between 0 and 500 (inclusive) for the *distance thresholding* and *nearest neighbours* algorithms[4]
- between 0 and 1 (inclusive) for the *nearest neighbours distance ratio* algorithm.

In addition, this parameter **can also be** `-1` to represent that no threshold value should be applied.

`<feature_index>` An integer giving the index of the feature for which to return matches; the possible range is $0, \ldots, 1499$. The reason is that 1500 features are being extracted per frame, as you will see in `visual_odometry.py`.

`<visualise>` A boolean value with the following meaning:

- Setting this to `True` enables real-time visualisation of the camera view and trajectory, and writes an image `matches.png` which shows the feature matches obtained for the given `<frame_id>`.
  The generated image shows two frames side-by-side: the current frame on the right, and the immediately preceding frame on the left.
- Setting it to `False` won't display any visualisation nor write any image file

[4] As this value is applied on Euclidean distances, in principle the threshold can be any positive number. However, for this exercise, only threshold values between 0 and 500 (inclusive) will be tested.

During development, you can, for example, run:

```
$ python3 run_VO.py drive_car ~/MyKITTI 150 1 100 209 True
```

which will run the VO system until the frame with an ID number of 150, apply the *distance thresholding* feature matching algorithm using 100 as the threshold value, return the matches for feature element at index 209, and generate a PNG file named `matches.png` in the working directory.

You can also run the command:

```
$ python run_VO.py drive_car ~/MyKITTI 150 2 -1 209 True
```

which, similarly to above, will run the VO system until the frame with an ID number of 150, but will instead apply the *nearest neighbours* feature matching algorithm **without** any threshold value, and return the match for feature element at index 209 without generating any image file.

The output that you will obtain from the stub code is of course not meaningful.

Important: For this lab exercise, only the `drive_car` functionality will be used when testing your solution. Depending on the particular versions of your Python and OpenCV installation, there may be differences in the output of numerical values. This will not affect the marking as both your solution and the reference implementation will be running on the same machine. The examples below have been run on Kilburn lab machines.

Important: You need to use a graphical environment that the OpenCV library supports, in order to run the `view_trajectory` command or `drive_car` commands with `<visualise>` set to `True`. These will work if you use the Ubuntu graphical login on Kilburn lab machines.

Tasks

For this lab exercise, the only Python file that you need to modify is `algorithms.py`. You will develop your own version of this script, henceforth referred to as "your solution" in this document, following the tasks outlined below.

C Familiarisation with the `visual_odometry.py` script

Although you do not need to make any changes to the `visual_odometry.py` script, it is advisable that you familiarise yourself with its contents before you proceed to developing your solution. In particular, it is useful to take note of the following details in relation to the `featureMatching` function:

- It takes two images as arguments: `image_cur`, the image corresponding to the `<frame_id>` given as input to `drive_car`; and `image_ref`, the image immediately preceding `image_cur` in the frame sequence.

  The other arguments to this function are `matching_algorithm`, `threshold_value`, `feature_index` and `visualise`, each corresponding to an argument supplied to `drive_car`.

- By calling the OpenCV function `SIFT_create`, the function firstly creates a feature detector (`detector`) based on Scale-Invariant Feature Transform[5] (SIFT), that is configured to extract 1500 features from each image (as specified by the `NUM_FEATURES` constant).

[5] `https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html`

- By calling the OpenCV function `detectAndCompute` of the feature detector object, a list of keypoints and a list of descriptors are computed for each of `image_ref` and `image_cur`.

  Note that the size of the list of keypoints and list of descriptors returned, is the same as `NUM_FEATURES`.

- A feature matching strategy is then applied by calling a function in `algorithms.py`, depending on the value of the `matching_algorithm` argument.

  Passed on as arguments to the feature matching function are the descriptor lists for the two images, as well as `threshold_value`.

D Implementation of your solution

This lab exercise requires you to implement three feature matching algorithms, which were explained in the Week 10 lectures. Stub code has been provided in `algorithms.py`, which you must modify to turn it into your own solution.

The stub code already contains lines of code for creating an object of type `BFMatcher` (short for brute-force matcher). This is an OpenCV class that implements some functions for computing the similarity (e.g., in terms of Euclidean distance) between image descriptors[6].

```
import cv2
bf = cv2.BFMatcher()
```

Without supplying any input arguments to `cv2.BFMatcher()`, Euclidean distance is used as the distance metric by default (which is what we need).

For each of the tasks below, you should use the `knnMatch` function[7] of the `BFMatcher` object to compute matches between two sets of descriptors. For example:

```
matches = bf.knnMatch(des1, des2, k=100)
```

will return the closest 100 matches for every descriptor in the query image (i.e., the current image), in order of increasing Euclidean distance.

The arguments `des1` and `des2` correspond to the descriptors computed for the current (query) image and preceding (reference image), respectively. Meanwhile, the value of `k` specifies the maximum number of closest matches that should be returned for each query descriptor.

The above call to `knnMatch` will return a list `matches` which is of the same size as the number of descriptors in the query image. Each element in `matches` is a list itself that contains `k` elements which are of type `DMatch`[8].

Any `DMatch` object has a `distance` attribute whose value holds the Euclidean distance between the query descriptor and the matched descriptor.

For example, if `des_match` is a `DMatch` object, one can obtain the value of the Euclidean distance between `des_match` and a query descriptor with:

```
dist_value = des_match.distance
```

Important: Do NOT simply return the raw return value of the call to `knnMatch`. The return type of the functions that you are required to implement should similarly be a list of lists of `DMatch` objects; however, you need to write code to remove any matches that should be eliminated depending on the specified algorithm and/or threshold value.

[6] `https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html`
[7] `https://docs.opencv.org/4.4.0/db/d39/classcv_1_1DescriptorMatcher.html`
[8] `https://docs.opencv.org/4.4.0/d4/de0/classcv_1_1DMatch.html`

Task 1: In your solution, write a function called `dist_thresholding` that implements feature matching based on the *distance thresholding* algorithm.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ python run_VO.py drive_car ~/MyKITTI 150 1 300 209 False
debug run: drive_car('/home/username/MyKITTI',150,1,300,209,False)
ret value: [78.26876831054688, 288.0329895019531, 297.3735046386719]
```

```
$ python run_VO.py drive_car ~/MyKITTI 150 1 100 209 False
debug run: drive_car('/home/username/MyKITTI',150,1,100,209,False)
ret value: [78.26876831054688]
```

Task 2: In your solution, write a function called `nn` that implements feature matching based on the *nearest neighbours* algorithm. Here, consider only the **first** nearest neighbour.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ python run_VO.py drive_car ~/MyKITTI 150 2 -1 209 False
debug run: drive_car('/home/username/MyKITTI',150,2,-1,209,False)
ret value: [78.26876831054688]
```

```
$ python run_VO.py drive_car ~/MyKITTI 150 2 150 209 False
debug run: drive_car('/home/username/MyKITTI',150,2,150,209,False)
ret value: [78.26876831054688]
```

```
$ python run_VO.py drive_car ~/MyKITTI 150 2 50 209 False
debug run: drive_car('/home/username/MyKITTI',150,2,50,209,False)
ret value: []
```

Task 3: In your solution, write a function called `nndr` that implements feature matching based on the *nearest neighbours distance ratio* algorithm.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ python run_VO.py drive_car ~/MyKITTI 120 3 0.97 45 False
debug run: drive_car('/home/username/MyKITTI',120,3,0.97,45,False)
ret value: [238.5351104736328]
```

```
$ python run_VO.py drive_car ~/MyKITTI 120 3 0.93 45 False
debug run: drive_car('/home/username/MyKITTI',120,3,0.93,45,False)
ret value: []
```

Submission

Please follow the `README.md` instructions in your `COMP24011_2022` Gitlab repo to refresh the files of your `lab4` branch. As described above, although you will find three Python scripts, you only need to modify `algorithms.py` to implement your solution. When you are ready to push your solution, remember that you **need to** tag your commit with the command

```
$ git tag 24011-lab4-S-SLAM
```

There must be **only one** file called `algorithms.py` in your commit. Also, any other files that you include **will be ignored**. General instructions on coursework submission using Gitlab can be found on the CS Handbook.

The deadline for submission is **18:00 on Monday, 12th December**. The lab will be **auto-marked** offline. The marking program will download your code from Gitlab and test it against a reference implementation. (This will take time so expect marking to take at least 2 weeks.)

During auto-marking, your code will be imported from `visual_odometry.py`

```
from algorithms import dist_thresholding, nn, nndr
```

in the same way as when you use the `run_VO.py` script. **Important:** The Python code provided for this lab exercise already includes the necessary top level code, so you **should not** add any and only complete the required function definitions in `algorithms.py`.

Marking Scheme

There are 20 marks in total which are distributed across the tasks as shown in the following table.

Task	Function	Marks
1	`dist_thresholding()`	6
2a	`nn()` with a specified threshold value	5
2b	`nn()` without a specified threshold value	3
3	`nndr()`	6

Each of your functions will be tested on various test cases, and return values compared to the reference implementation.

- In tasks 1 and 3 you obtain 1 mark if all tests complete without runtime errors. The proportion of tests with fully correct return values determines the remaining marks (you need 100% for 5 marks, 80% for 4 marks, 60% for 3 marks, 40% for 2 marks, and 20% for 1 mark).
- In task 2a you obtain 1 mark if all tests complete without runtime errors. The proportion of tests with fully correct return values determines the remaining marks (you need 100% for 4 marks, 75% for 3 marks, 50% for 2 marks, and 25% for 1 mark).
- In task 2a you obtain 0.5 mark if all tests complete without runtime errors. The proportion of tests with fully correct return values determines the remaining 2.5 marks. Your final score is truncated to an integer value. To get 1 mark you either pass at least 40% of tests or complete all and pass at least 20% of tests. To get 2 mark you either pass at least 80% of tests or complete all and pass at least 60% of tests. To get 3 marks you need to pass all tests.

Appendix: Setting up your development environment

As mentioned in the Preparatory Work section of this manual, you need to install a recent version of the OpenCV library and its Python bindings. It may happen that this is not available for your default Python environment. While we can only support software running on the Kilburn lab machines, this section offers guidance on setting up the required environment. You are welcome to share your experiences of this on the Blackboard discussion board.
Important: You should not attempt this on any of the Kilburn lab machines, because the limited disk space may not enough for the following installation steps.

1. If you have not installed it yet, install Anaconda by downloading and running the appropriate installer from `https://www.anaconda.com/products/distribution`

   The current versions include Python 3.9 (which you can also use for this lab).

2. For instructions on how to complete the Anaconda installation, refer to Steps 3 to 7 of this tutorial. Although written for Ubuntu 18.04 users, the tutorial instructions are generic for Unix-like systems.

   You need to initialise Conda for your shell and reload its configuration. If you don't do this as part of the installation process, then you'll need to run commands like the following.

   ```
   $ conda init bash
   $ source ~/.bashrc
   ```

   You will notice that your prompt has slightly changed to something like:

   ```
   (base) $
   ```

3. Create a new Conda virtual environment for this lab exercise. This will provide a fresh environment where you can install new packages without affecting the base Conda environment. If you want to run Python 3.8 as in the Kilburn lab machines, then run the following command:

   ```
   (base) $ conda create -y -n COMP24011_Lab4 python=3.8
   ```

   This will automatically download the required packages.

4. Activate the lab environment by running:

   ```
   (base) $ conda activate COMP24011_Lab4
   ```

   You will notice that your prompt has slightly changed to something like:

   ```
   (COMP24011_Lab4) $
   ```

   which indicates that you are currently using the newly created virtual environment.

5. You can now install the `opencv-contrib-python` library as explained in the Preparatory Work section of this manual:

   ```
   (COMP24011_Lab4) $ pip install opencv-contrib-python
   ```

6. Work on your solution using the newly created virtual environment. Whenever you wish to stop using this environment and return to your base Conda environment, you can run:

   ```
   (COMP24011_Lab4) $ conda deactivate
   ```