COMP24011 Lab4:
Features for Estimating Autonomous Vehicle Poses

Riza Batista-Navarro and Francisco Lobo
Academic session: 2023-24

Introduction

In this exercise, you will investigate 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. Let's start by introducing some terms and their definitions to help clarify 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. *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 SLAM Evaluation 2012 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. In contrast, global optimisation is achieved through loop closure: the correction of the trajectory upon revisiting an already encountered location.

The original KITTI data set consists of 4541 images and occupies several gigabytes. To reduce disk quota and running time requirements, this lab exercise will use a simplified version of the data set that we refer to as MyKITTI. This contains only the first 501 images of the original, but these still occupy close to 150 MB and so **cannot be included** in your GitLab repo. For this reason, you will need to download `MyKITTI.zip` from Dropbox and extract it.

> 👉 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`. The command-line tool for this lab accepts an optional parameter allowing you to set a different path for the MyKITTI images.

In the `lab3` branch of your `COMP24011_2023` GitLab repo you will find a fully functional monocular VO system. The implementation provided to you builds upon another library available on GitHub, which was ported to Python 3 and recent OpenCV 4 versions, and has additional functionality specific to this lab exercise. To use our VO system, you will need to install the Python bindings for OpenCV by issuing the following command

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

OpenCV is a comprehensive computer vision framework. It is very popular in both academic and industrial contexts, and thus benefits from active development.

> 👉 The lab code depends on OpenCV 4 functionality which has changed in recent years. Indeed, the continuous development of OpenCV has involved changes in data structures as well as changes in the API due to patent issues. The above command installs version 4.8.0.x on the Kilburn lab machines which will be the **only supported version** for this assignment.

In this exercise you will use OpenCV to complete computer vision tasks, which will involve getting familiar with its 2D Features Framework. The VO system for the lab uses the Scale-Invariant Feature Transform (SIFT) algorithm for feature detection and matching, so you will find the OpenCV-Python tutorials on these topics very useful.

The code we provide already implements feature tacking, which identifies feature correspondences between adjacent images. Your job is to implement *feature matching* and allow for inspecting any feature matches. This will require coding the following feature matching strategies:

1) distance thresholding,
2) nearest neighbour, and
3) nearest neighbour distance ratio.

You will also need to extract relevant information about the resulting feature matches; namely the coordinates of all the features involved and the distance for each matching pair.

The VO System

Once you refresh the `lab3` branch of your GitLab repo you will find the following Python files.

`run_odometry.py`	This is the command-line tool that runs the visual odometry according to the subcommand (and the parameters) provided by the user. It contains the `RunOdometry` class.
`visual_odometry.py`	This is the main module of the visual odometry system implementing the `PinholeCamera` and `VisualOdometry` classes.
`vision_tasks_base.py`	This module contains the `VisionTasksBase` "abstract" class that specifies the signatures of four methods you need to implement, and implements the interface used in `RunOdometry` and `VisualOdometry`.
`vision_tasks.py`	This is the module that you need to complete for this exercise. It contains the `VisionTasks` class that is derived from `VisionTasksBase`, and must implement its abstract methods in order to complete feature matching and retrieve their details.

In order to complete this exercise you will need to understand both `vision_tasks_base.py` and `vision_tasks.py` but you do not need to know the details of how `run_odometry.py` or `visual_odometry.py` are coded.

The VO tool provides comprehensive help messages. To get started run the command

```
$ ./run_odometry.py -h
usage: run_odometry.py [-h] -d DATASET
                       {view_trajectory,view_feature,view_info,get_info} ...

options:
  -h, --help            show this help message and exit
  -d DATASET, --dataset DATASET
                        path to KITTI dataset directory (option required
                        unless dataset is located at ~/MyKITTI)

subcommands:
  select which odometry command to run

  {view_trajectory,view_feature,view_info,get_info}
    view_trajectory     show the car camera view and calculated trajectory
    view_feature        show the matches for a frame feature using OpenCV (and
                        save this image as opencv_visual.png)
    view_info           show the matches for a frame feature using both OpenCV
                        and the info algorithm (and save this composite image
                        as custom_visual.png)
    get_info            use the info algorithm to get details of matches for a
                        frame feature
```

As noted in the introduction, you do not need to specify where the MyKITTI data set is located if you extract it to `~/MyKITTI`, otherwise every time you run `run_odometry.py` you will need to add the appropriate `-d` option. The VO tool supports four subcommands: `view_trajectory`, `view_feature`, `view_info` and `get_info`. Each of these subcommands has its own help message which you can access with commands like

```
$ ./run_odometry.py view_trajectory -h
usage: run_odometry.py view_trajectory [-h] [frame_id]

positional arguments:
  frame_id    index of frame to stop visualisation (1 to 500)

options:
  -h, --help  show this help message and exit
```

Thus, if you extracted MyKITTI at `/tmp/MyKITTI` and want to visualise the trajectory of the AV up to frame 150, you should run the command

```
$ ./run_odometry.py -d /tmp/MyKITTI view_trajectory 150
vo params: (None, None, '~/MyKITTI')
debug run: view_trajectory(150,)
Processing frame 150
ret value: None
```

You should see the trajectory and camera view in real time (depending on processing power). The VO tool will open a window that sequentially displays the first 151 images in the data set (with IDs 0 to 150), and will show in another window the true trajectory (drawn in green) and the estimated trajectory (drawn in red) of the vehicle. While it is running you will get on the shell progress indication showing the frame index being processed.

You need to use a graphical environment that the OpenCV library supports, in order to run the `view_trajectory`, `view_feature` and `view_info` subcommands. These will work as expected if you use the Ubuntu graphical login on Kilburn lab machines.

Determining the vehicle's trajectory only involves feature tracking, which has been implemented for you in `VisionTasksBase`. Once you start implementing the feature matching algorithms, you can begin using the subcommands to show and report on feature matches. You will probably use `view_feature` first, as this requires you to code only a particular matching algorithm. To get the command-line help run

```
$ ./run_odometry.py view_feature -h
usage: run_odometry.py view_feature [-h] -a {dt,nn,nndr} -t THRESHOLD
                                    frame_id feature_id

positional arguments:
  frame_id              index of chosen frame (1 to 500)
  feature_id            index of chosen feature (0 to 1499)

options:
  -h, --help            show this help message and exit
  -a {dt,nn,nndr}, --algorithm {dt,nn,nndr}
                        name of the matching algorithm used to match features
  -t THRESHOLD, --threshold THRESHOLD
                        decimal value of threshold for feature matching
                        (option required except for nn algorithm)
```

The subcommands `view_info` and `get_info` have similar command-line options. The possible choices for algorithm correspond to the feature matching strategies that you'll implement in Tasks 1 to 3 below. These are:

`dt` to use your implementation of *distance thresholding*,

`nn` to use your implementation of *nearest neighbour*, and

`nndr` to use your implementation of *nearest neighbours distance ratio*.

The valid ranges of threshold value depend on the choice of matching algorithm, as follows:

- between 0 and 500 (inclusive[1]) for the `dt` and `nn`,
- between 0 and 1 (inclusive) for the `nndr` algorithm.

The threshold value is optional when using `nn` as described in Task 2.

As an example, once you completed Task 1 below, you can view the matches of the feature with index 127 of frame 100 according to the distance thresholding algorithm for a threshold value of 232.5 by running

```
$ ./run_odometry.py view_feature -a dt -t 232.5 101 127
vo params: ('dt', 232.5, '~/MyKITTI')
debug run: view_feature(101, 127)
Processing frame 101
ret value: None
```

This will display frames 100 and 101 side-by-side and OpenCV, using its own `cv2.drawMatchesKnn` function, will draw lines from feature 127 in the first frame to its matches in the second, as shown in Figure 1.

Figure 1: Viewing a feature

Because of the small size of Figure 1, it is difficult to see that feature 127 is the bottom right corner of the lower window in frame 100, and that it is matched with the bottom right corners of the upper and lower windows in frame 101, but this will be easier on the screen.

- While frame indices correspond to images in the MyKITTI data set, feature indices are generated by OpenCV at runtime. This means that there is no guarantee that feature 127 is as shown in Figure 1 if you run the above example. There will be a feature corresponding to the bottom right corner of the lower window in frame 100, but depending on the C libraries installed on the lab machine this feature may be assigned a different index.
- This issue affects all of the examples below (though hopefully feature indices won't be too unreliable when using Kilburn lab machines). However, please note that this **will not affect the marking** as both your solution and the reference implementation will be running on the same machine!

You will be able to use the `view_info` and `get_info` subcommands, once you've completed Task 4 below. Using `view_info` instead of `view_feature` will produce the image in Figure 2. It combines the OpenCV visualisation of feature matches with a rendering of the coordinates that your implementation of `VisionTasks.matching_info()` returns (in green). This will allow you to visually inspect if your code calculates the correct coordinate values.

1. 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.

Figure 2: Viewing feature information

To get the details of the matches that your function returns, you need to run

```
$ ./run_odometry.py get_info -a dt -t 232.5 101 127
vo params: ('dt', 232.5, '../MyKITTI')
debug run: get_info(101, 127)
Processing frame 101
ret value: ((1066, 152),
            [(1032, 152), (1033, 9)],
            [70.21395874023438, 172.14529418945312])
ret count: 3
```

As the example shows, this will be a tuple with the following three elements:

- a tuple of two integers,
  giving the x and y coordinates of the chosen feature in the previous frame,
- a list of tuples,
  each with x and y coordinates in the current frame matching the chosen feature; and
- a list of floats,
  each giving the distance between the chosen feature and the corresponding match.

For the example, in frame 100 the bottom right corner of the lower window is located $(1066, 152)$ and its matches in frame 101 have coordinates $(1032, 152)$ and $(1033, 9)$ on the lower and upper window, respectively.

Assignment

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

Although you do not need to make any changes to the `vision_tasks_base.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 `VisionTasksBase` class:

- Its constructor creates a `detector` object by calling the OpenCV function `cv2.SIFT_create`, which firstly creates a feature detector based on the Scale-Invariant Feature Transform (SIFT) algorithm that is configured to extract 1500 features from each image (as specified by the `NUM_FEATURES` constant).
- Its `featureMatching` function takes two images as arguments: `cur_image`, the image corresponding to the `<frame_id>` given as input to `drive_car`; and `prev_image`, the image immediately preceding `cur_image` in the frame sequence.

- By calling the `detectAndCompute` function of the `detector` object, a list of keypoints and a list of descriptors are computed for each of `prev_image` and `cur_image`.

  Note that the size of the list of keypoints and list of descriptors returned, is the same as `NUM_FEATURES`.
- A feature matching strategy — based on your own implementation of the `dt`, `nn` and `nndr` functions in `vision_tasks.py` — is then applied through a call to `matching_algo`, taking as parameters the list of descriptors extracted for each of `prev_image` and `cur_image`, as well as a threshold value (if provided).

This lab exercise requires you to implement three feature matching algorithms (which were explained in the Week 7 lectures) and one function for determining the coordinates of the matches. Stub code has been provided in `vision_tasks.py`, which you must modify to turn it into your own solution.

In your implementation, you can make use of a class called `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 (i.e., features).

```
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 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 an image, in order of increasing Euclidean distance.

The arguments `des1` and `des2` correspond to the descriptors computed for the query (preceding or previous) image and reference (current) 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`.

👉 The return type of the feature matching 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.

Any `DMatch` object has the following attributes, which you will likely find useful:

`distance` the Euclidean distance between the query descriptor and the matched descriptor,

`queryIdx` the index of the query keypoint in the list of keypoints detected from the query (i.e. preceding) image,

`trainIdx` the index of the reference keypoint in the list of keypoints detected from the reference (i.e. current) image.

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

```
dist_value = m.distance
```

A `DMatch` object contains information on which keypoints from the query and reference images have been considered as a match. The first step towards obtaining these keypoints is by determining their indices, as follows:

```
query_kp_index = m.queryIdx
ref_kp_index = m.trainIdx
```

In the above, each of `query_kp_index` and `ref_kp_index` is an integer which is an index. Thus, one can obtain the actual keypoints with:

```
query_kp = prev_keypoints[query_kp_index]
ref_kp = cur_keypoints[ref_kp_index]
```

assuming that `prev_kp` and `cur_kp` are the list of keypoints detected from the query (preceding or previous) and reference (current) images, respectively. Here, each of `query_kp` and `ref_kp` is of type `KeyPoint`. It has an attribute called `pt` which is a tuple of two elements, corresponding to the x and y coordinates of the keypoint. However, the coordinates are provided as floating point values, thus one needs to cast them into integers, as in the following example:

```
x_coord_query_kp = int(query_kp.pt[0])
y_coord_query_kp = int(query_kp.pt[1])
```

Bearing in mind the above notes on OpenCV functions and classes, you can proceed to completing the tasks below.

Task 1: In your solution, write a function called `dt` that implements feature matching based on the *distance thresholding* algorithm. It returns a list of lists of `DMatch` objects.

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

```
$ ./run_odometry.py get_info -a dt 150 209 -t 100
vo params: ('dt', 100.0, '~/MyKITTI')
debug run: get_info(150, 209)
Processing frame 150
ret value: ((226, 119), [(208, 115)], [72.02082824707031])
ret count: 3
$ ./run_odometry.py get_info -a dt 150 209 -t 300
vo params: ('dt', 300.0, '~/MyKITTI')
debug run: get_info(150, 209)
Processing frame 150
ret value: ((226, 119), [(208, 115), (394, 52), (519, 213), \
(872, 160), (402, 48)], [72.02082824707031, 242.6643829345703, \
282.074462890625, 290.75762939453125, 294.5997314453125])
ret count: 3
```

Task 2: In your solution, write a function called `nn` that implements feature matching based on the *nearest neighbours* algorithm. It returns a list of lists of `DMatch` objects.

Recall that the nearest neighbours algorithm can be used with or without a threshold value; hence the `--threshold` argument is optional if `--algorithm` is set to `nn`. In this case your function will get the value `None` as the threshold argument.

Again, you can verify that your function behaves correctly on the command line. For example, you should obtain output that is similar to the following.

```
$ ./run_odometry.py get_info -a nn 150 209
vo params: ('nn', None, '~/MyKITTI')
debug run: get_info(150, 209)
Processing frame 150
ret value: ((226, 119), [(208, 115)], [72.02082824707031])
ret count: 3
$ ./run_odometry.py get_info -a nn 150 209 -t 100
vo params: ('nn', 100.0, '~/MyKITTI')
debug run: get_info(150, 209)
Processing frame 150
ret value: ((226, 119), [(208, 115)], [72.02082824707031])
ret count: 3
$ python run_odometry.py get_info -a nn 150 209 -t 50
vo params: ('nn', 50.0, '~/MyKITTI')
debug run: get_info(150, 209)
Processing frame 150
ret value: ((0, 0), [], [])
ret count: 3
```

Task 3: In your solution, write a function called `nndr` that implements feature matching based on the *nearest neighbours distance ratio* algorithm. It returns a list of lists of `DMatch` objects.

As above, you can verify that your function behaves correctly on the command line. For example, you should obtain output that is similar to the following.

```
$ ./run_odometry.py get_info -a nndr 120 45 -t 0.97
vo params: ('nndr', 0.97, '~/MyKITTI')
debug run: get_info(120, 45)
Processing frame 120
ret value: ((22 , 14), [(61 , 163)], [156.46725463867188])
ret count: 3
$ ./run_odometry.py get_info -a nndr 120 45 -t 0.93
vo params: ('nndr', 0.93, '~/MyKITTI')
debug run: get_info(120, 45)
Processing frame 120
ret value: ((0, 0), [], [])
ret count: 3
```

Task 4: In your solution, write a function called `matching_info` that takes the matches obtained by any of the above-described feature matching functions, and determines the image coordinates of the query keypoint and of every matching reference keypoint. It returns a tuple with the following three elements:

- a tuple of two integers, which corresponds to the x and y coordinates of the query keypoint;
- a list of tuples, where each tuple has two integers corresponding to the x and y coordinates of a matching reference keypoint; and
- a list of floats, which corresponds to the distances between the query keypoint and the matching keypoints.

If the matches passed on the function is an empty list, the return value should be a tuple whose first element is `(0,0)` and whose second and third elements are empty lists: `((0, 0), [], [])`.

You can verify that your function behaves correctly on the command line by inspecting the return values of of the `get_info` subcommand, as exemplified in Tasks 1, 2 and 3 above. Additionally, you can use the `view_info` subcommand to visualise the coordinates that you have obtained and check if they are the same as what OpenCV's own `cv2.drawMatchesKnn` function returns.

Submission

Please follow the `README.md` instructions in your `COMP24011_2023` GitLab repo. Refresh the files of your `lab3` branch and develop your solution to the lab exercise. The solution consists of a single file called `vision_tasks.py` which must be submitted to your GitLab repo and tagged as `lab3_sol`. The `README.md` instructions that accompany the lab files include the `git` commands necessary to commit, tag, and then push **both** the commit and the tag to your `COMP24011_2023` GitLab repo. Further instructions on coursework submission using GitLab can be found in the CS Handbook, including how to change a `git` tag after pushing it.

The deadline for submission is **18:00 on Friday 24th November**. In addition, no work will be considered for assessment and/or feedback if submitted more than **2 weeks** after the deadline. (Of course, these rules will be subject to any mitigating procedures that you have in place.)

The lab exercise will be **auto-marked** offline. The automarker program will download your submission from GitLab and test it against our reference implementation. For each task the return value of your function will be checked on a random set of valid arguments. A time limit of 10 seconds will be imposed on every function call, and exceeding this time limit will count as a runtime error. If your function does not return values of the correct type, this will also count as a runtime error.

A total of 20 marks is available in this exercise, distributed as shown in the following table.

Task	Function	Marks
1	`VisionTasks.dt()`	5
2	`VisionTasks.nn()`	5
3	`VisionTasks.nndr()`	5
4	`VisionTasks.matching_info()`	5

The marking scheme for all tasks is as follows:

- You obtain the first 0.5 marks if all tests complete without runtime errors.
- The proportion of tests with fully correct return values determines the remaining 4.5 marks.

Important Clarifications

- It will be very difficult for you to circumvent time limits during testing. If you try to do this, the most likely outcome is that the automarker will fail to receive return values from your implementation, which will have the same effect as not completing the call. In any case, an additional time limit of 300 seconds for all tests of each task will be enforced.
- This lab exercise is fully auto-marked. If you submit code which the Python interpreter does not accept, you will score 0 marks. The Python setup of the automarker is the same as the one on the department's Ubuntu image, but only a **minimal** set of Python modules are available. If you choose to add `import` statements to the sample code, it is **your responsibility** to ensure these are part of the default Python package available on the lab machines.
- It doesn't matter how you organise your `lab3` branch, but you should avoid having multiple files with the same name. The automarker will sort your directories alphabetically (more specifically, in ASCII ascending order) and find submission files using breadth-first search. It will mark the first `vision_tasks.py` file it finds and ignore all others.
- Every file in your submission should only contain printable ASCII characters. If you include other Unicode characters, for example by copying and then pasting code from the PDF of the lab manuals, then the automarker is likely to reject your files.