Feature Mapping Example Notes

Rafael Radkowski, last edited: Feb 24, 2014

Introduction

The example HCI571X_Feature_Matching is a natural feature tracking (NFT) code example that introduces the OpenCV methods necessary for feature matching and pose estimation. It creates a database from a set or photos; this photos are referred to as training images, the database is called training database (train_db). It matches descriptors extracted from photo files or from a video camera image against the descriptors in the training database.

The application supports SIFT and SURF descriptors and uses k nearest neighbors(knn)-feature matching with kd-trees (k-dimensional trees) for feature matching. To increase the robustness, it applies a ratio test, a symmetry test, and an epipolar test.

For preparation, create a set of training images, take photos, and provide them in a folder; refered to as the *train*_db folder. A set of example images are provided inside the folder *teachdb*.

A set of query images is provided inside the folder *querydb*. However, it is recommended to test the video capability and to prepare and track a particular image on your own.

Start the application

The application works with start arguments that you have to add after the executable's name when you start the program from terminal, e.g

```
HCI571X_Feature_Matching.exe -type -source
```

-type: set the descriptor type. It can be either -*SIFT* or -*SURF*

-*source* can either be -*video* or -*file*

- -*file*: this loaded the query images and the database images from files located in the specified folder. Syntax:
  -*file path_to_train_db path_to_query_db*
  - *path_to_train_db*: the path to the folder that contains all your training images e.g. ../teach_db
  - *path_to_query_db:* the path to all your test images.
- -*video*: this loads the database / training images from files located in the specified folder. Syntax:
  -*video path_to_teach_db device_id*
  - *path_to_teach_db*: the path to the folder that contains all your training images e.g. ../teach_db
  - *device_id*: an integer that indicates the video camera, e.g. 0

Example for *-file*

```
HCI571X_Feature_Matching.exe -SIFT -file ../teach_db ../query_db
```

Example for *-video*

```
HCI571X_Feature_Matching.exe -SIFT -video ../teach_db 0
```

Code Details

The table describes important variables and functions of the natural feature tracking code

Variable / Function	Description
`cv::Ptr<cv::FeatureDetector> _detector;`	The keypoint detector
`cv::Ptr<cv::DescriptorExtractor> _extractor;`	The descriptor extractor
`std::vector<cv::Mat> _descriptorsRefDB;`	This variable keeps all the descriptors. It is the descriptor database.
`std::vector< std::vector<cv::KeyPoint> > _keypointsRefDB;`	This variable is the keypoint database is all reference keypoints. Keypoints and descriptors with the same vector index belong to one tracking target.
`cv::FlannBasedMatcher _matcher;`	A knn matcher, it works with a kd-tree and can be trained in advance
`cv::BruteForceMatcher< cv::L1<float> > _brute_force_matcher`	A brute force matcher. Training of this matcher is not possible. Simply use the function match.
`init_database(.....)`	Inits the database by loading all images from a certain directory, extracts the feature keypoints and descriptors and saves them in the databases keypointsRefDB and _descriptorsRefDB

`ratioTest(....)`
 	
`float _ratio`
	Applies the ratio test to clear matches for which a ratio that is > than a threshold _ratio.

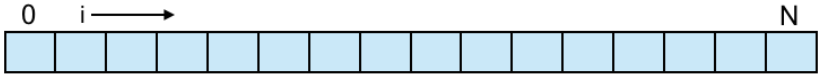
`ransacTest(...)`	Identify good matches using RANSAC.
`symmetryTest(...)`	Check for symmetrical matches in matches1 and matches2 and adds them into a variable ATTENTION: the method returns only the closest match when the input are the two best matches with knn, k=2. It does not work when only one best matches has been identified.
`knn_match(..)`	This code carries out the knn-matching and the refinement.
`run_matching(std::string directory_path, std::vector<std::string> files)`	This function contains the knn-matching meta process. It reads all photos from the specified directory path as query images and pass them to the knn-match.
`run_matching(int video_device);`	This function opens a video camera, fetches the video stream, and matches the descriptors found in the video against the

	database.
`brute_force_match(...)`	This function applies a brute force match without a trained data structure.
`run_bf_matching( std::string directory_path, std::vector<std::string> files)`	This function contains the brute-force matching meta process. Itreads all photos from the specified directory path as query images and pass them to the knn-match.
`run_bf_matching(int video_device);`	This function opens a video camera, fetches the video stream, and matches the descriptors found in the video against the database.

Iterators

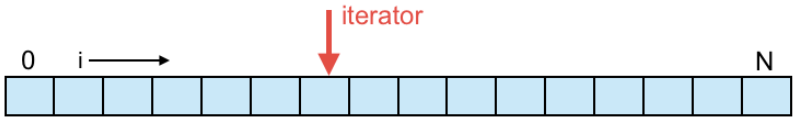
The example uses several iterators that allow us to run through all elements of a vector using the boolean operators ++ or --.

Consider a `std::vector` object with N elements:



Each vector element can be addressed with an index i.

An iterator is a pointer element that directly points to a particular element of this vector. It gives direct access to the indicated cell.



Functions

Creating an empty iterator object

Use:

```
std::vector<[datatype]>::iterator itr
```

- with *iterator*, the keyword to create an iterator object,
- *[datatype],* the datatype of your vector,
- *itr,* the name of the variable.

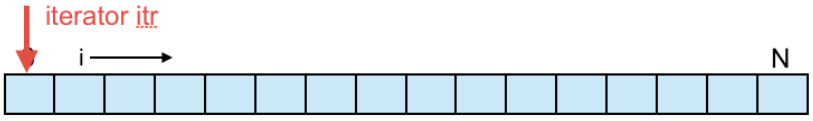
The object *itr* is empty at this moment and cannot be used.

Assign an iterator

The iterator itself is part of the vector object and can be fetched from this object using the function *begin()*.

```
itr = variable.begin();
```

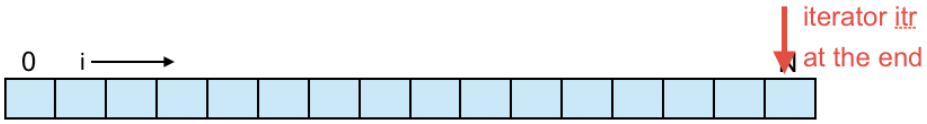
The function returns an iterator that points to the first element of this vector.



The function end()

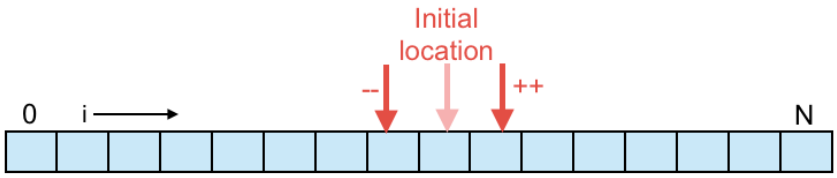
```
itr = variable.end();
```

returns an iterator that points to the last element of this vector.



Stepping forward

The operators ++ and -- allow us to move the iterator through the vector where ++ moves the iterator forwards and -- moves the iterator backwards. Both operators can be called multiple times but be careful not to step over the ends of the vector; the application will crash.



Accessing data

The iterator is a pointer object that points to the data element but it is not the data which is stored inside the vector. To access the data, the asterisk operator must be used.

Example:

```
std::vector<int>::iterator itr;
itr = variable.begin();

int data = (*itr);
```

In this example, the data stored in the vector is an integer value. Using the function begin(), we receive a pointer that points to the first element. In line 3, the asterisk operator * is used to receive the integer value from the iterator *itr*.

Running through a vector

A for-loop or a while-loop are typically used to run through all elements of an interator,

Example:

```
std:vector<int> myVector;

// Adding data
myVector.push_back(4);
myVector.push_back(5);
myVector.push_back(7);
myVector.push_back(121);
myVector.push_back(35);

// Creating an iterator
std::vector<int>::iterator itr = myVector.begin();

// Using a loop to run through all elements
while(itr != myVector.end())
{
     // Fetch the data
     int data = (*itr);

     // print the data
     std::cout << data << std::endl;

     // Stepping to the next element
     itr++;
}
```

The condition to exit the loop is:

```
itr != myVector.end()
```

We check whether the *itr* arrived at the end of the vector; if so, the loop stops to operate.

Attention: never forget to step through your vector using *itr*++. Forgetting this line will end in a deadlock.