In this chapter, we will cover:

- Detecting Harris corners
- Detecting FAST features
- Detecting the scale-invariant SURF features
- Describing SURF features

**Introduction**

In computer vision, the concept of interest points, also called keypoints or feature points, has been largely used to solve many problems in object recognition, image registration, visual tracking, 3D reconstruction, and more. It relies on the idea that instead of looking at the image as a whole, it could be advantageous to select some special points in the image and perform a local analysis on these ones. These approaches work well as long as a sufficient number of such points are detected in the images of interest and these points are distinguishing and stable features that can be accurately localized. This chapter will introduce a few interest point detectors and show you how to use them in image matching.
Detecting and Matching Interest Points

Detecting Harris corners

When searching for interesting feature points in images, corners come out as an interesting solution. They indeed are local features that can be easily localized in an image, and in addition, they should abound in scenes of man-made objects (where they are produced by walls, doors, windows, tables, and so on). Corners are also interesting because they are two-dimensional features that can be accurately localized (even at sub-pixel accuracy) as they are at the junction of two edges. This is in contrast to points located on a uniform area or on the contour of an object and that would be difficult to repeatedly localize precisely on other images of the same object.

The Harris feature detector is a classical approach to detect corners in an image. We will explore this operator in this recipe.

How to do it...

The basic OpenCV function for detecting Harris corners is called `cv::cornerHarris` and is straightforward to use. You call it on an input image and the result is an image of floats which gives the corner strength at each pixel location. A threshold is then applied on this output image in order to obtain a set of detected corners. This is accomplished by the following code:

```cpp
// Detect Harris Corners
cv::Mat cornerStrength;
cv::cornerHarris(image, cornerStrength,
                  3,     // neighborhood size
                  3,     // aperture size
                  0.01); // Harris parameter

// threshold the corner strengths
cv::Mat harrisCorners;
double threshold = 0.0001;
cv::threshold(cornerStrength, harrisCorners,
              threshold, 255, cv::THRESH_BINARY);
```
Here is the original image:
Detecting and Matching Interest Points

The result is a binary map image shown in the following screenshot which is inverted for better viewing (that is, we used `cv::THRESH_BINARY_INV` instead of `cv::THRESH_BINARY` to get the detected corners in black):

![Harris Corner Map](image)

From the preceding function call, we observe that this interest point detector requires several parameters (these will be explained in the next section) which may make it difficult to tune. In addition, the corner map that is obtained contains many clusters of corner pixels which contradict the fact that we would like to detect well-localized points. Therefore, we will try to improve the corner detection method by defining our own class to detect Harris corners.

The class encapsulates the Harris parameters with their default values and corresponding getter and setter methods (which are not shown here):

```cpp
class HarrisDetector {
    private:
        // 32-bit float image of corner strength
        cv::Mat cornerStrength;
        // 32-bit float image of thresholded corners
        cv::Mat cornerTh;
```
To detect the Harris corners on an image, we proceed with two steps. First, the Harris values at each pixel are computed:

```cpp
// Compute Harris corners
void detect(const cv::Mat& image) {
    // Harris computation
    cv::cornerHarris(image, cornerStrength,
                     neighbourhood, // neighborhood size
                     aperture,       // aperture size
                     k);             // Harris parameter

    // internal threshold computation
    double minStrength; // not used
    cv::minMaxLoc(cornerStrength,
                   &minStrength, &maxStrength);

    // local maxima detection
    cv::Mat dilated; // temporary image
    cv::dilate(cornerStrength, dilated, cv::Mat());
    cv::compare(cornerStrength, dilated,
                localMax, cv::CMP_EQ);
}
```
Next, the feature points are obtained based on a specified threshold value. Since the range of possible values for Harris depends on the particular choices of its parameters, the threshold is specified as a quality level defined as a fraction of the maximal Harris value computed in the image:

```cpp
// Get the corner map from the computed Harris values
cv::Mat getCornerMap(double qualityLevel) {
    cv::Mat cornerMap;
    // thresholding the corner strength
    threshold = qualityLevel*maxStrength;
    cv::threshold(cornerStrength, cornerTh, threshold, 255, cv::THRESH_BINARY);
    // convert to 8-bit image
    cornerTh.convertTo(cornerMap, CV_8U);
    // non-maxima suppression
    cv::bitwise_and(cornerMap, localMax, cornerMap);
    return cornerMap;
}
```

This method returns a binary corner map of the detected features. The fact that the detection of the Harris features has been split into two methods allows us to test the detection with a different threshold (until an appropriate number of feature points are obtained) without needing to repeat costly computations. It is also possible to obtain the Harris features in the form of a `std::vector<cv::Point>`:

```cpp
// Get the feature points from the computed Harris values
void getCorners(std::vector<cv::Point> &points, double qualityLevel) {
    // Get the corner map
    cv::Mat cornerMap = getCornerMap(qualityLevel);
    // Get the corners
    getCorners(points, cornerMap);
}
```

```cpp
// Get the feature points from the computed corner map
void getCorners(std::vector<cv::Point> &points, const cv::Mat& cornerMap) {
    // Iterate over the pixels to obtain all features
    for( int y = 0; y < cornerMap.rows; y++ ) {
        const uchar* rowPtr = cornerMap.ptr<uchar>(y);
        for(int x = 0; x < cornerMap.cols; x++ ) {
            // if it is a feature point
            if (rowPtr[x]) {
```
points.push_back(cv::Point(x,y));
}
}
}

This class improves the detection of the Harris corners by adding a non-maxima suppression step which will be explained in the next section. The detected points can now be drawn on an image using the cv::circle function as demonstrated by the following method:

```cpp
// Draw circles at feature point locations on an image
void drawOnImage(cv::Mat &image,
                 const std::vector<cv::Point> &points,
                 cv::Scalar color= cv::Scalar(255,255,255),
                 int radius=3, int thickness=2) {
    std::vector<cv::Point>::const_iterator it=
        points.begin();

    // for all corners
    while (it!=points.end()) {
        // draw a circle at each corner location
        cv::circle(image,*it,radius,color,thickness);
        ++it;
    }
}
```

Using this class, the detection of the Harris points is accomplished as follows:

```cpp
// Create Harris detector instance
HarrisDetector harris;
// Compute Harris values
harris.detect(image);
// Detect Harris corners
std::vector<cv::Point> pts;
harris.getCorners(pts,0.01);
// Draw Harris corners
harris.drawOnImage(image,pts);
```
How it works...

To define the notion of corners in images, Harris looks at the average directional intensity change in a small window around a putative interest point. If we consider a displacement vector \((u, v)\), the average intensity change is given by:

\[
R = \sum (I(x + u, y + v) - I(x, y))^2
\]

The summation is over a defined neighborhood around the considered pixel (the size of this neighborhood corresponds to the third parameter in the `cv::cornerHarris` function). This average intensity change can then be computed in all possible directions which leads to the definition of a corner as a point for which the average change is high in more than one direction. From this definition, the Harris test is performed as follows. We first obtain the direction of maximal average intensity change. Next, check if the average intensity change in the orthogonal direction is also high. If it is the case, then we have a corner.
Mathematically, this condition can be tested by using an approximation of the preceding formula using Taylor expansion:

\[
R = \sum \left( I(x,y) + \frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v - I(x,y) \right) = \sum \left( \frac{\partial I}{\partial x} u \right)^2 + \left( \frac{\partial I}{\partial y} v \right)^2 + 2 \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} uv
\]

Which is then rewritten in matrix form:

\[
R = \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} \sum \frac{\partial I}{\partial x}^2 & \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \\ \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} & \sum \frac{\partial I}{\partial y}^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}
\]

This matrix is a covariance matrix that characterizes the rate of intensity change in all directions. This definition involves the image’s first derivatives that are often computed using the Sobel operator. This is the case of the OpenCV implementation, the fourth parameter of the function corresponding to the aperture used for the computation of the Sobel filters. It can be shown that the two eigenvalues of the covariance matrix gives the maximal average intensity change and the average intensity change for the orthogonal direction. It then follows that if these two eigenvalues are low, we are in a relatively homogenous region. If one eigenvalue is high and the other is low, we must be on an edge. Finally, if both eigenvalues are high, then we are at a corner location. Therefore, the condition for a point to be accepted as a corner is to have the smallest eigenvalue of the covariance matrix higher than a given threshold.

The original definition of the Harris corner algorithm uses some properties of the eigendecomposition theory in order to avoid the cost of explicitly computing the eigenvalues. These properties are:

- The product of the eigenvalues of a matrix is equal to its determinant
- The sum of the eigenvalues of a matrix is equal to the sum of the diagonal of the matrix (also known as the trace of the matrix)

It then follows that we can verify that two eigenvalues are high by computing the following score:

\[
\text{Det}(C) - k \cdot \text{Trace}^2(C)
\]

One can easily verify that this score will indeed be high only if both eigenvalues are also high. This is the score that is computed by the `cv::cornerHarris` function at each pixel location. The value of k is specified as the fifth parameter of the function. It could be difficult to determine what would be the best value for this parameter. However, in practice, it has been shown that a value in the range of 0.05 and 0.5 generally gives good results.
Detecting and Matching Interest Points

To improve the result of the detection, the class described in the previous section adds an additional non-maxima suppression step. The goal here is to exclude Harris corners that are adjacent to others. Therefore, to be accepted, the Harris corner must not only have a score higher than the specified threshold, but it must also be a local maximum. This condition is tested by using a simple trick which consists of dilating the image of Harris score in our detect method:

\[
\text{cv::dilate(cornerStrength,dilated,cv::Mat());}
\]

Since the dilation replaces each pixel value by the maximum in the defined neighborhood, then the only points that will not be modified are the local maxima. That is what is verified by the following equality test:

\[
\text{cv::compare(cornerStrength,dilated,}
\text{localMax,cv::CMP_EQ)}; \\
\]

The localMax matrix will therefore be true (that is non-zero) only at local maxima locations. We then use it in our getCornerMap method to suppress all non-maximal features (using the cv::bitwise_and function).

There's more...

Additional improvements can be made to the original Harris corner algorithm. This section describes another corner detector found in OpenCV which expands the Harris detector to make its corners more uniformly distributed across the image. As we will see, this operator has an implementation in the new OpenCV 2 common interface for feature detector.

Good features to track

With the advent of the floating-point processor, the mathematical simplification introduced to avoid the eigenvalue decomposition has become negligible, and consequently the detection of Harris can be made based on the explicitly computed eigenvalues. In principle, this modification should not significantly affect the result of the detection, but it avoids the use of the arbitrary *k* parameter.

A second modification addresses the problem of feature point clustering. Indeed, in spite of the introduction of the local maxima condition, interest points tend to be unevenly distributed across an image, showing concentrations at locations highly textured. A solution to this problem is to impose a minimum distance between two interest points. This can be achieved by the following algorithm. Starting from the point with the strongest Harris score (that is with the largest minimum eigenvalue), only accept interest points if they are located at least, a given distance from the already accepted points. This solution is implemented in OpenCV in the function cv::goodFeaturesToTrack thus named because the features it detects can be used as a good starting set in visual tracking application. It is called as follows:

\[
// \text{Compute good features to track}
\text{std::vector<cv::Point2f> corners;}
\]
cv::goodFeaturesToTrack(image, corners,
500,   // maximum number of corners to be returned
0.01,   // quality level
10);   // minimum allowed distance between points

In addition to the quality-level threshold value, and the minimum tolerated distance between interest points, the function also uses a maximum number of points to be returned (this is possible since points are accepted in order of strength). The preceding function call produces the following result:

![Good Features to Track Detector](image)

This approach increases the complexity of the detection since it requires the interest points to be sorted by their Harris score, but it also clearly improves the distribution of the points across the image. Note that this function also includes an optional flag to request Harris corners to be detected using the classical corner score definition (using covariance matrix determinant and trace).
Feature detector common interface

OpenCV 2 has introduced a new common interface for its different interest point detectors. This interface allows easy testing of different interest point detectors within the same application.

The interface defines a Keypoint class that encapsulates the properties of each detected feature point. For the Harris corners, only the position of the keypoints is relevant. The recipe Detecting scale-invariant SURF points will discuss the other properties that could be associated to a keypoint.

The cv::FeatureDetector abstract class basically imposes the existence of a detect operation with the following signatures:

```cpp
void detect( const Mat& image, vector<KeyPoint>& keypoints,
            const Mat& mask=Mat() ) const;
void detect( const vector<Mat>& images,
            vector<vector<KeyPoint> >& keypoints,
            const vector<Mat>& masks=vector<Mat>{} ) const;
```

The second method allows interest points to be detected in a vector of images. The class also includes other methods to read and write the detected points in a file.

The cv::goodFeaturesToTrack function has a wrapper class called cv::GoodFeatureToTrackDetector, which inherits from the cv::FeatureDetector class. It can be used in a way similar to what we did with our Harris Corners class, that is:

```cpp
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;
// Construction of the Good Feature to Track detector
cv::GoodFeaturesToTrackDetector gftt(500, 0.01, 10);
// point detection using FeatureDetector method
gftt.detect(image,keypoints);
```

The results are the same as the one obtained before, since the same function is ultimately called by the wrapper.

See also


The article by J. Shi and C. Tomasi, Good features to track, Int. Conference on Computer Vision and Pattern Recognition, pp. 593-600, 1994 which introduced these features.

### Detecting FAST features

The Harris operator proposed a formal mathematical definition for corners (or more generally, interest points) based on the rate of intensity changes in two perpendicular directions. Although this constitutes a sound definition, it requires the computation of the image derivatives which is a costly operation, especially considering the fact that interest point detection is often just the first step in a more complex algorithm.

In this recipe, we present another feature point operator. This one has been specifically designed to allow quick detection of interest points in an image. The decision to accept or not to accept a keypoint being based on only a few pixel comparisons.

#### How to do it...

Using the OpenCV 2 common interface for feature point detection makes the deployment of any feature point detectors easy. The one presented in this recipe is the FAST detector. As the name suggests, it has been designed to be quick to compute:

```cpp
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;

// Construction of the Fast feature detector object
cv::FastFeatureDetector fast(40); // threshold for detection

// feature point detection
fast.detect(image, keypoints);
```

Note that OpenCV also proposes a generic function to draw keypoints on an image:

```cpp
cv::drawKeypoints(image, keypoints, image, cv::Scalar(255,255,255), cv::DrawMatchesFlags::DRAW_OVER_OUTIMG);
```
By specifying the chosen drawing flag, the keypoints are drawn over the output image, thus producing the following result:

![FAST Features](image)

An interesting option is to specify a negative value for the keypoint color. In that case, a different random color will be selected for each drawn circle.

**How it works...**

As in the case of the Harris point, the FAST (Features from Accelerated Segment Test) feature algorithm derives from the definition of what constitutes a "corner". This time, this definition is based on the image intensity around a putative feature point. The decision to accept a keypoint is done by examining a circle of pixels centered at a candidate point. If an arc of contiguous points of length greater than 3/4 of the circle perimeter is found in which all pixels significantly differ from the intensity of the center point, then a keypoint is declared.
This is a simple test that can quickly be computed. Moreover, the algorithm uses an additional trick to further speed-up the process. Indeed, if we first test four points separated by 90° on the circle (for example, top, bottom, right, and left points) it can be easily shown that to satisfy the condition expressed above, at least three of these points must all be brighter or darker than the central pixel. If it is not the case, the point can immediately be rejected without inspecting additional points on the circumference. This is a very effective test since, in practice, most of the image points will be rejected by this simple 4-comparison test.

In principle, the radius of the circle of examined pixels should be a parameter of the method. However, it has been found that, in practice, a radius of 3 gives both good results and high efficiency. There are then 16 pixels to consider on the circumference of the circle as seen below:

```
  16 1  2
  15  3
 14  4
 13  5
 12  6
 11  7
 10  9  8
```

The four points used for the pretest are pixels 1, 5, 9, and 13.

As for Harris features, it is often better to perform non-maxima suppression on the corners found. Therefore, a corner strength measure needs to be defined. Several alternatives could have been considered, and the one that has been retained is the following. The strength of a corner is given by the sum of absolute difference between the central pixel and the pixels on the identified contiguous arc.

This algorithm results in very fast interest point detection and should then be used when speed is a concern. For example, this is often the case in visual tracking applications where several points must be tracked in a video sequence with high frame rates.

See also

Detecting the scale-invariant SURF features

When trying to match features across different images, we are often faced with the problem of scale changes. That is, the different images to be analyzed can be taken at a different distance from the objects of interest, and consequently, these objects will be pictured at different sizes. If we try to match the same feature from two images using a fixed size neighborhood then, because of the scale change, their intensity patterns will not match.

To solve this problem, the concept of scale-invariant features has been introduced in computer vision. The main idea here is to have a scale factor associated with each of the detected feature points. In recent years, several scale-invariant features have been proposed and this recipe presents one of them, the SURF features. SURF stands for Speeded Up Robust Features, and as we will see, they are not only scale-invariant features, but they also offer the advantage of being computed very efficiently.

How to do it...

The OpenCV implementation of SURF features also use the cv::FeatureDetector interface. Therefore, the detection of these features is similar to what we demonstrated in the previous recipes of this chapter:

```cpp
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;
// Construct the SURF feature detector object
cv::SurfFeatureDetector surf(2500.); // threshold
// Detect the SURF features
surf.detect(image,keypoints);
```

To draw these features, we again use the cv::drawKeypoints OpenCV function, but this time with another mask because we also want to show the scale factor associated with each feature:

```cpp
// Draw the keypoints with scale and orientation information
cv::drawKeypoints(image, // original image
    keypoints, // vector of keypoints
    featureImage, // the resulting image
    cv::Scalar(255,255,255), // color of the points
    cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS); //flag
```
The resulting image with the detected feature that is produced by the drawing function is:

As can be seen in the preceding screenshot, the size of the keypoint circles resulting from the use of the `DRAW_RICH_KEYPOINTS` flag is proportional to the computed scale of each feature. The SURF algorithm also associates an orientation with each feature to make them rotation-invariant. This orientation is illustrated by a radial line inside each drawn circle.
Detecting and Matching Interest Points

If we take another picture of the same object but at a different scale, the feature detection results in:

![SURF Features](image)

By carefully observing the detected keypoints, it can be seen that the change in size of corresponding circles is proportional to the scale change. As an example, consider the bottom part of the upper-right window. In both images, a SURF feature has been detected at that location and the two corresponding circles (of different sizes) contain the same visual elements. Of course, this is not the case for all features, but as we will discover in the next chapter, the repeatability rate is sufficiently high to allow good matching between the two images.
In Chapter 6, we learned that the image derivatives of an image can be estimated using Gaussian filters. Those filters make use of a \( \sigma \) parameter defining the aperture (size) of the kernel. As we saw, this \( \sigma \) corresponds to the variance of the Gaussian function used to construct the filter, and it then implicitly defines a scale at which the derivative is evaluated. Indeed, a filter having a larger \( \sigma \) value smoothed out the finer details of the image. This is why we can say that it operates at a coarser scale.

Now, if we compute, for instance, the Laplacian of a given image point using Gaussian filters at different scales, then different values are obtained. Looking at the evolution of the filter response for different scale factors, we obtain a curve which eventually reaches a maximum value at some \( \sigma \) value. If we extract this maximum value for two images of the same object taken at two different scales, the ratio of these two \( \sigma \) maxima will correspond to the ratio of the scales at which the images were taken. This important observation is at the core of the scale-invariant feature extraction process. That is, scale-invariant features should be detected as local maxima in both the spatial space (in the image) and the scale space (as obtained from the derivative filters applied at different scales).

SURF implements this idea by proceeding as follows. First, to detect the features, the Hessian matrix is computed at each pixel. This matrix measures the local curvature of a function and is defined as:

\[
H(x,y) = \begin{bmatrix}
\frac{\partial^2 I}{\partial x^2} & \frac{\partial^2 I}{\partial x \partial y} \\
\frac{\partial^2 I}{\partial x \partial y} & \frac{\partial^2 I}{\partial y^2}
\end{bmatrix}
\]

The determinant of this matrix gives the strength of this curvature. The idea is therefore to define corners as image points with high local curvature (that is, high variation in more than one direction). Since it is composed of second-order derivatives, this matrix can be computed using Laplacian Gaussian kernels of different scale \( \sigma \). This Hessian then becomes a function of three variables: \( H(x,y,\sigma) \). A scale-invariant feature is therefore declared when the determinant of this Hessian reaches a local maximum in both spatial and scale space (that is, \( 3 \times 3 \times 3 \) non-maxima suppression needs to be performed). However, this determinant must have a minimum value as specified by the first parameter in the constructor of the \texttt{cv::SurfFeatureDetector} class.
Detecting and Matching Interest Points

The calculation of all of these derivatives at different scales is computationally costly. The objective of the SURF algorithm is to make this process as efficient as possible. This is achieved by using approximated Gaussian kernels involving only few integer additions. These have the following structure:

The kernel on the left is used to estimate the mixed second derivatives, while the right one estimates the second derivative in the vertical direction. A rotated version of this second kernel estimates the second derivative in the horizontal direction. The smallest kernels have a size of 9x9 pixels corresponding to $\sigma \approx 1.2$. Kernels of increasing size are successively applied. The exact amount of filter that is applied can be specified by additional parameters of the SURF class. By default, 12 different sizes of kernels are used (going up to size 99x99). Note that the fact that integral images are used guarantees that the sum inside each lob can be computed by using only 3 additions independently of the size of the filter.

Once the local maxima is identified, the precise position of each detected interest point is obtained through interpolation in both scale and image space. The result is then a set of feature points localized at sub-pixel accuracy and to which is associated a scale value.

There's more...

The SURF algorithm has been developed as an efficient variant of another well-known scale-invariant feature detector called SIFT (for Scale-Invariant Feature Transform). SIFT also detects features as local maxima in image and scale space, but uses the Laplacian filter response instead of the Hessian determinant. This Laplacian at different scales is computed using difference of Gaussian filters. OpenCV has a wrapper class that detects these features and it is called in a way similar to the SURF features:

```cpp
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;
// Construct the SURF feature detector object
cv::SiftFeatureDetector sift{
  0.03, // feature threshold
  10.}; // threshold to reduce
// sensitivity to lines
// Detect the SURF features
sift.detect(image, keypoints);
```
The results are also very similar:

However, since the computation of the feature point is based on floating-point kernels, it is generally considered to be more accurate in terms of feature localization in space and scale. Although, for the same reason, it is also more computationally expensive.

**See also**


Describing SURF features

The SURF algorithm, discussed in the preceding recipe, defines a location and a scale for each of the detected features. This scale factor can be used to define the size of a window around the feature point such that the defined neighborhood would include the same visual information no matter what scale the object to which the feature belongs has been pictured. In addition, the visual information included in this neighborhood can be used to characterize the feature point to make it distinguishable from the others.

This recipe will show you how to describe a feature point's neighborhood using compact descriptors. In feature matching, feature descriptors are usually N-dimensional vectors that describe a feature point, ideally in a way that is invariant to change in lighting and to small perspective deformations. In addition, good descriptors can be compared using a simple distance metric (for example, Euclidean distance). Therefore, they constitute a powerful tool to use in feature matching algorithms.

How to do it...

The following code is a pattern similar to the one used for feature detection. OpenCV 2 proposes a general class which defines a common interface for the extraction of the various feature point descriptors that are available. To follow up on the preceding recipe, here we use the one proposed in the SURF algorithm. Based on the std::vector of cv::Keypoint instances obtained from feature detection, the descriptors are obtained as follows:

```cpp
// Construction of the SURF descriptor extractor
cv::SurfDescriptorExtractor surfDesc;
// Extraction of the SURF descriptors
cv::Mat descriptors1;
surfDesc.compute(image1, keypoints1, descriptors1);
```

The result is a matrix (that is, a cv::Mat instance) which will contain as many rows as the number of elements in the keypoint vector. Each of these rows is an N-dimensional descriptor vector. In the case of the SURF descriptor, by default, it has a size of 64. This vector characterizes the intensity pattern surrounding a feature point. The more similar the two feature points, the closer their descriptor vectors should be.

These descriptors are particularly useful in image matching. Suppose, for example, that two images of the same scene are to be matched. This can be accomplished by first detecting features on each image, and then extracting the descriptors of these features. Each feature descriptor vector in the first image is then compared to all feature descriptors in the second image. The pair that obtains the best score (that is, the lowest distance between the two vectors) is then kept as the best match for that feature. This process is repeated for all features in the first image. This is the most basic scheme that has been implemented in OpenCV as the cv::BruteForceMatcher. It is used as follows:
// Construction of the matcher
cv::BruteForceMatcher<cv::L2<float>> matcher;
// Match the two image descriptors
std::vector<cv::DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);

This class is a subclass of the cv::DescriptorMatcher class defining the common interface for different matching strategies. The result is a vector of cv::DMatch instances which is the structure used to represent a match pair. Essentially, the cv::DMatch data structure contains a first index referring to an element in the first vector of descriptors, and a second index referring to the matching feature in the second vector of descriptors. It also contains a real value representing the distance between the two matched descriptors. This distance value is used in the definition of operator< comparing two cv::DMatch instances.

In order to visualize the result of the matching operation, OpenCV offers a drawing function that produces an image made of the concatenation of the two input images and on which matching points are linked by a line. In the preceding recipe, we obtained 340 SURF points for the first image. The brute-force approach will then produce the same number of matches. Drawing all of these lines on an image would make the results unreadable. Therefore, we will only display the 25 matches with the lowest distance. This is easily accomplished by using the std::nth_element that positions the nth element in sorted order at the nth position, with all elements smaller placed before this element. Once this is done, the vector is simply purged of its remaining elements:

std::nth_element(matches.begin(),    // initial position
                 matches.begin()+24, // position of the sorted element
                 matches.end());     // end position
// remove all elements after the 25th
matches.erase(matches.begin()+25, matches.end());

Recall that the preceding code works because the operator< has been defined in the cv::DMatch class. These 25 matches can then be visualized through the following call:

cv::Mat imageMatches;
cv::drawMatches(
    image1, keypoints1, // 1st image and its keypoints
    image2, keypoints2, // 2nd image and its keypoints
    matches,            // the matches
    imageMatches,       // the image produced
    cv::Scalar(255,255,255)); // color of the lines
Detecting and Matching Interest Points

That produces the following image:

As can be seen, most of these matches correctly link a point on the left with its corresponding image point on the right. One can notice some errors due to the fact that the observed building has a symmetrical façade which makes some of the local matches ambiguous (the topmost match is one example of wrongly matched features).

**How it works...**

Good feature descriptors must be invariant to small changes in illumination, in viewpoint, and to the presence of image noise. Therefore, they are often based on local intensity differences. This is the case of the SURF descriptors which apply the following simple kernels inside a larger neighborhood around a keypoint:

\[
\begin{pmatrix}
-1 & 1 \\
-1 & 1 \\
\end{pmatrix}
\]
The first one simply measures the local intensity difference in the horizontal direction (designated as $dx$), and the second measures this difference in the vertical direction (designated as $dy$). The size of the neighborhood used to extract the descriptor vector is defined as 20 times the scale factor of the feature (that is, $20\sigma$). This square region is then split into 4x4 smaller square sub-regions. For each sub-region, the kernel responses $dx$ and $dy$ are computed at 5x5 regularly spaced locations (the kernel size being $2\sigma$). All of these responses are summed as follows in order to extract four descriptor values for each sub-region:

$$
\begin{bmatrix}
\sum dx & \sum dy & \sum |dx| & \sum |dy|
\end{bmatrix}
$$

Since there are $4x4=16$ sub-regions, we have a total of 64 descriptor values. Note that in order to give more importance to the neighboring pixel values closer to the keypoint, the kernel responses are weighted by a Gaussian centered at the keypoint location (with a $\sigma=3.3$).

The $dx$ and $dy$ responses are also used to estimate the orientation of the feature. These values are computed (with a kernel size of $4\sigma$) within a circular neighborhood of radius $6\sigma$ at locations regularly spaced by intervals of $\sigma$. For a given orientation, the responses inside a certain angular interval ($\pi/3$) are summed, and the orientation giving the longest vector is defined as the dominant orientation.

With the SURF features and descriptors, scale-invariant matching can be achieved. Here is an example showing the 25 best matches in a match pair containing two images at different scales:
There's more...

The SIFT algorithm also defines its own descriptor. It is based on the gradient magnitude and orientation computed at the scale of the considered keypoint. As for the SURF descriptors, the scaled neighborhood of the keypoint is divided into $4 \times 4$ sub-regions. For each of these regions, an 8-bin histogram of gradient orientations (weighted by their magnitude and by a global Gaussian window centered at the keypoint) is built. Therefore, the descriptor vector is made of the entries of these histograms. There are $4 \times 4$ regions and 8 bins per histogram, which leads to a descriptor of length 128.

As for feature detection, the difference between SURF and SIFT descriptors is mainly speed and accuracy. Since SURF descriptors are mostly based on intensity differences, they are faster to compute. However, SIFT descriptors are generally considered to be more accurate in finding the right matching feature.

See also

The previous recipe for more on the SURF and SIFT features.