Notes on the OpenSURF Library

Christopher Evans

January 18, 2009

1 SURF: Speeded Up Robust Features

In this document, the SURF detector-descriptor scheme used in the OpenSURF library is discussed in detail. First the algorithm is analysed from a theoretical standpoint to provide a detailed overview of how and why it works. Next the design and development choices for the implementation of the library are discussed and justified. During the implementation of the library, it was found that some of the finer details of the algorithm had been omitted or overlooked, so Section 1.5 serves to make clear the concepts which are not explicitly defined in the SURF paper [1].

1.1 Integral Images

Much of the performance increase in SURF can be attributed to the use of an intermediate image representation known as the "Integral Image" [7]. The integral image is computed rapidly from an input image and is used to speed up the calculation of any upright rectangular area. Given an input image I and a point (x, y) the integral image $I_{\sum}$ is calculated by the sum of the values between the point and the origin. Formally this can be defined by the formula:

$$I_{\sum}(x,y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(x,y) \tag{1}$$

Using the integral image, the task of calculating the area of an upright rectangular region is reduced four operations. If we consider a rectangle bounded by vertices A, B, C and D as in Figure 1, the sum of pixel intensities is calculated by:

$$\sum = A + D - (C + B) \tag{2}$$

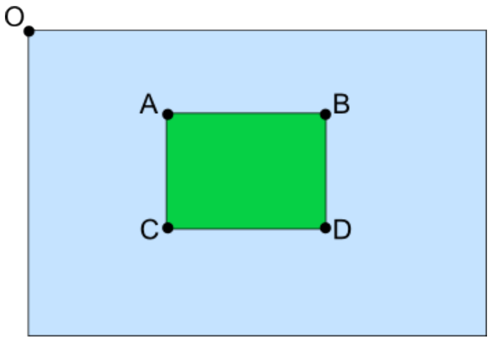

Figure 1: Area computation using integral images

Since computation time is invariant to change in size this approach is particularly useful when large areas are required. SURF makes good use of this property to perform fast convolutions of varying size box filters at near constant time.

1.2 Fast-Hessian Detector

1.2.1 The Hessian

The SURF detector is based on the determinant of the Hessian matrix. In order to motivate the use of the Hessian, we consider a continuous function of two variables such that the value of the function at (x, y) is given by $f(x, y)$. The Hessian matrix, H, is the matrix of partial derivates of the function f.

$$H(f(x,y)) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{3}$$

The determinant of this matrix, known as the discriminant, is calculated by:

$$det(H) = \frac{\partial^2 f}{\partial x^2}\frac{\partial^2 f}{\partial y^2} - \left(\frac{\partial^2 f}{\partial x \partial y}\right)^2 \tag{4}$$

The value of the discriminant is used to classify the maxima and minima of the function by the second order derivative test. Since the determinant is the product of eigenvalues of the Hessian we can classify the points based on the sign of the result. If the determinant is negative then the eigenvalues have different signs and hence the point is not a local extremum; if it is positive then either both eigenvalues are positive or both are negative and in either case the point is classified as an extremum.

Translating this theory to work with images rather than a continuous function is a fairly trivial task. First we replace the function values $f(x, y)$ by the image pixel intensities $I(x, y)$. Next we require a method to calculate the second order partial derivatives of the image. We can calculate the derivatives by convolution with an appropriate kernel. In the case of SURF the second order scale normalised Gaussian is the chosen filter as it allows for analysis over scales as well as space (scale-space theory is discussed further later in this section). We can construct kernels for the Gaussian derivatives in x, y and combined xy direction such that we calculate the four entries of the Hessian matrix. Use of the Gaussian allows us to vary the amount of smoothing during the convolution stage so that the determinant is calculated at different scales. Furthermore, since the Gaussian is an isotropic (i.e. circularly symmetric) function, convolution with the kernel allows for rotation invariance. We can now calculate the Hessian matrix, H, as function of both space $\mathbf{x} = (x, y)$ and scale σ.

$$H(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}, \tag{5}$$

Here $L_{xx}(\mathbf{x}, \sigma)$ refers to the convolution of the second order Gaussian derivative $\frac{\partial^2 g(\sigma)}{\partial x^2}$ with the image at point $\mathbf{x} = (x, y)$ and similarly for L_{yy} and L_{xy}. These derivatives are known as Laplacian of Gaussians.

Working from this we can calculate the determinant of the Hessian for each pixel in the image and use the value to find interest points. This variation of the Hessian detector is similar to that proposed by Beaudet [2].

Lowe [4] found performance increase in approximating the Laplacian of Gaussians by a difference of Gaussians. In a similiar manner, Bay [1] proposed an approximation to the Laplacian of Gaussians by using box filter representations of the respective kernels. Figure 2 illustrates the similarity between the discretised and cropped kernels and their box filter counterparts. Considerable performance increase is found when these filters are used in conjunction with the integral image described in Section 1.1. To qauntify the difference we can consider the number of array accesses and operations required in the convolution. For a 9 × 9 filter we would require 81 array accesses and operations for the original real valued filter and only 8 for the box filter representation. As the filter is increased in size, the computation cost increases significantly for the original Laplacian while the cost for the box filters is independent of size.

In Figure 2 the weights applied to each of the filter sections is kept simple. For the D_{xy} filter the black regions are weighted with a value of 1, the white regions with a value of -1 and the remaining areas not weighted at all. The D_{xx} and D_{yy} filters are weighted similarly but with the white regions have a value of -1 and the black with a value of 2. Simple weighting allows for rapid calculation of areas but in using these weights we need to address the difference in response values between the original and approximated kernels. Bay [1] proposes the following formula as an accurate approximation for the

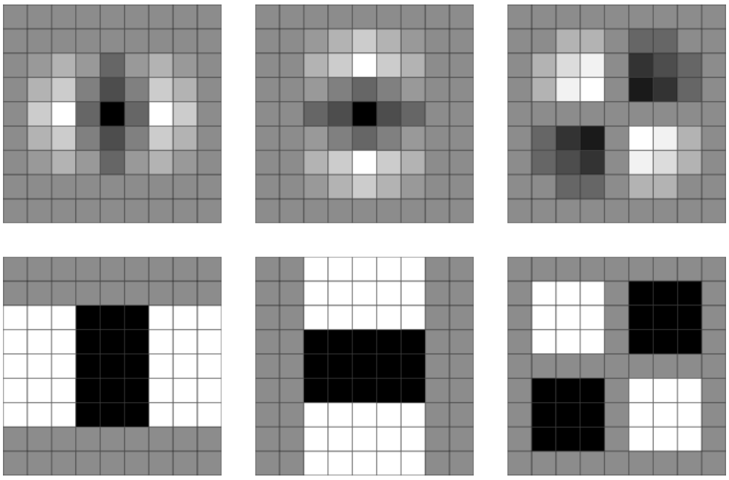

Figure 2: Laplacian of Gaussian Approximation. Top Row: The discretised and cropped second order Gaussian derivatives in the x, y and xy-directions. We refer to these as L_{xx}, L_{yy}, L_{xy}. Bottom Row: Weighted Box filter approximations in the x, y and xy-directions. We refer to these as D_{xx}, D_{yy}, D_{xy}

Hessian determinant using the approximated Gaussians:

$$det(H_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \tag{6}$$

In [1] the two filters are compared in detail and the results conclude that the box representation's negligible loss in accuracy is far outweighed by the considerable increase in efficiency and speed. The determinant here is referred to as the blob response at location $\mathbf{x} = (x, y, \sigma)$. The search for local maxima of this function over both space and scale yields the interest points for an image. The exact method for extracting interest points is explained in the following section.

1.2.2 Constructing the Scale-Space

In order to detect interest points using the determinant of Hessian it is first necessary to introduce the notion of a scale-space. A scale-space is a continuous function which can be used to find extrema across all possible scales [8]. In computer vision the scale-space is typically implemented as an image pyramid where the input image is iteratively convolved with Gaussian kernel and repeatedly sub-sampled (reduced in size). This method is used to great effect in SIFT [4] but since each layer relies on the previous, and images need to be resized it is not computationally efficient. As the processing time of the kernels used in SURF is size invariant, the scale-space can be created by applying kernels of increasing size to the original image. This allows for multiple layers of the scale-space pyramid to be processed simultaneously and negates the need to subsample the image hence providing performance increase. Figure 3 illustrates the difference between the traditional scale-space structure and the SURF counterpart.

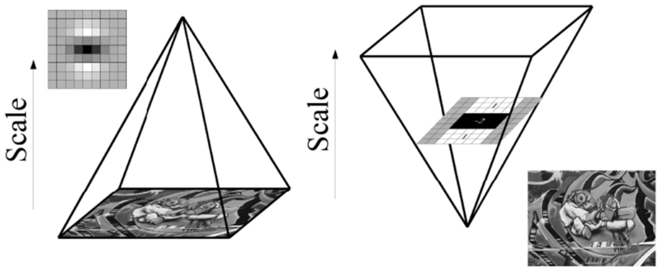

Figure 3: Filter Pyramid. The traditional approach to constructing a scale-space (left). The image size is varied and the Guassian filter is repeatedly applied to smooth subsequent layers. The SURF approach (right) leaves the original image unchanged and varies only the filter size.

The scale-space is divided into a number of octaves, where an octave refers to a series of response maps of covering a doubling of scale. In SURF the lowest level of the scale-space is obtained from the output of the 9×9 filters shown in 2. These filters correspond to a real valued Gaussian with $\sigma = 1.2$. Subsequent layers are obtained by upscaling the filters whilst maintaining the same filter layout ratio. As the filter size increases so too does the value of the associated Gaussian scale, and since ratios of the layout remain constant we can calculate this scale by the formula:

$$\begin{aligned}\sigma_{approx} &= \text{Current Filter Size} \cdot \frac{\text{Base Filter Scale}}{\text{Base Filter Size}} \\ &= \text{Current Filter Size} \cdot \frac{1.2}{9}\end{aligned}$$

When constructing larger filters, there are a number of factors which must be take into consideration. The increase in size is restricted by the length of the positive and negative lobes of the underlying second order Gaussian derivatives. In the approximated filters the lobe size is set at one third the side length of the filter and refers to the shorter side length of the weighted black and white regions. Since we require the presence of a central pixel, the dimensions must be increased equally around this location and hence the lobe size can increase by a minimum of 2. Since there are three lobes in each filter which must be the same size, the smallest step size between consecutive filters is 6. For the D_{xx} and D_{yy} filters the longer side length of the weighted regions increases by 2 on each side to preserve structure. Figure 4 illustrates the structure of the filters as they increase in size.

1.2.3 Accurate Interest Point Localisation

The task of localising the scale and rotation invariant interest points in the image can be divided into three steps. First the responses are thresholded such that all values below

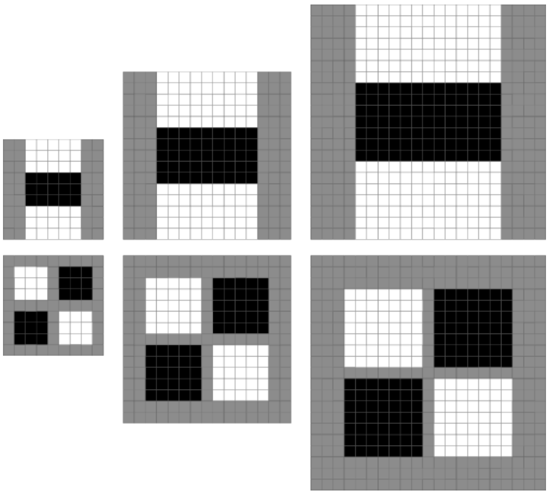

Figure 4: Filter Structure. Subsequent filters sizes must differ by a minimum of 6 to preserve filter structure.

the predetermined threshold are removed. Increasing the threshold lowers the number of detected interest points, leaving only the strongest while decreasing allows for many more to detected. Therefore the threshold can be adapted to tailor the detection to the application.

After thresholding, a non-maximal suppression is performed to find a set of candidate points. To do this each pixel in the scale-space is compared to its 26 neighbours, comprised of the 8 points in the native scale and the 9 in each of the scales above and below. Figure 5 illustrates the non-maximal suppression step. At this stage we have a set of interest points with minimum strength determined by the threshold value and which are also local maxima/minima in the scale-space.

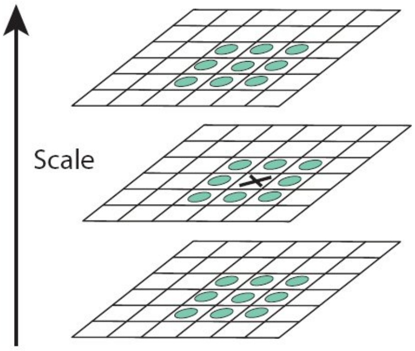

Figure 5: Non-Maximal Suppression. The pixel marked 'X' is selected as a maxima if it greater than the surrounding pixels on its interval and intervals above and below.

The final step in localising the points involves interpolating the nearby data to find the location in both space and scale to sub-pixel accuracy. This is done by fitting a 3D quadratic as proposed by Brown [3]. In order to do this we express the determinant of the Hessian function, $H(x, y, \sigma)$, as a Taylor expansion up to quadratic terms centered at detected location. This is expressed as:

$$H(\mathbf{x}) = H + \frac{\partial H}{\partial \mathbf{x}}^{T} \mathbf{x} + \frac{1}{2} \mathbf{x}^{T} \frac{\partial^2 H}{\partial \mathbf{x}^2} \mathbf{x} \tag{7}$$

The interpolated location of the extremum, $\hat{x} = (x, y, \sigma)$, is found by taking the derivative of this function and setting it to zero such that:

$$\hat{x} = -\frac{\partial^2 H}{\partial \mathbf{x}^2}^{-1} \frac{\partial H}{\partial \mathbf{x}} \tag{8}$$

The derivatives here are approximated by finite differences of neighbouring pixels. If $\hat{x}$ is greater than 0.5 in the x, y or σ directions we adjust the location and perform the interpolation again. This procedure is repeated until $\hat{x}$ is less than 0.5 in all directions or the the number of predetermined interpolation steps has been exceeded. Those points which do not converge are dropped from the set of interest points leaving only the most stable and repeatable.

1.3 Interest Point Descriptor

The SURF descriptor describes how the pixel intensities are distributed within a scale dependent neighbourhood of each interest point detected by the Fast-Hessian. This approach is similar to that of SIFT [4] but integral images used in conjunction with filters known as Haar wavelets are used in order to increase robustness and decrease computation time. Haar wavelets are simple filters which can be used to find gradients in the x and y directions.

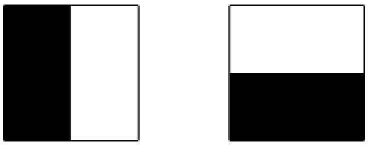

Figure 6: Haar Wavelets. The left filter computes the response in the x-direction and the right the y-direction. Weights are 1 for black regions and -1 for the white. When used with integral images each wavelet requires just six operations to compute.

Extraction of the descriptor can be divided into two distinct tasks. First each interest point is assigned a reproducible orientation before a scale dependent window is constructed in which a 64-dimensional vector is extracted. It is important that all calculations for the descriptor are based on measurements relative to the detected scale in order to achieve scale invariant results. The procedure for extracing the descriptor is explained further in the following.

1.3.1 Orientation Assignment

In order to achieve invariance to image rotation each detected interest point is assigned a reproducible orientation. Extraction of the descriptor components is performed relative to this direction so it is important that this direction is found to be repeatable under varying conditions. To determine the orientation, Haar wavelet responses of size 4σ are calculated for a set pixels within a radius of 6σ of the detected point, where σ refers to the scale at which the point was detected. The specific set of pixels is determined by sampling those from within the circle using a step size of σ.

The responses are weighted with a Gaussian centered at the interest point. In keeping with the rest the Gaussian is dependent on the scale of the point and chosen to have standard deviation 2.5σ. Once weighted the responses are represented as points in vector space, with the x-responses along the abscissa and the y-responses along the ordinate. The domimant orientation is selected by rotating a circle segment covering an angle of $\frac{\pi}{3}$ around the origin. At each position, the x and y-responses within the segment are summed and used to form a new vector. The longest vector lends its orientation the interest point. This process is illustrated in Figure 7.

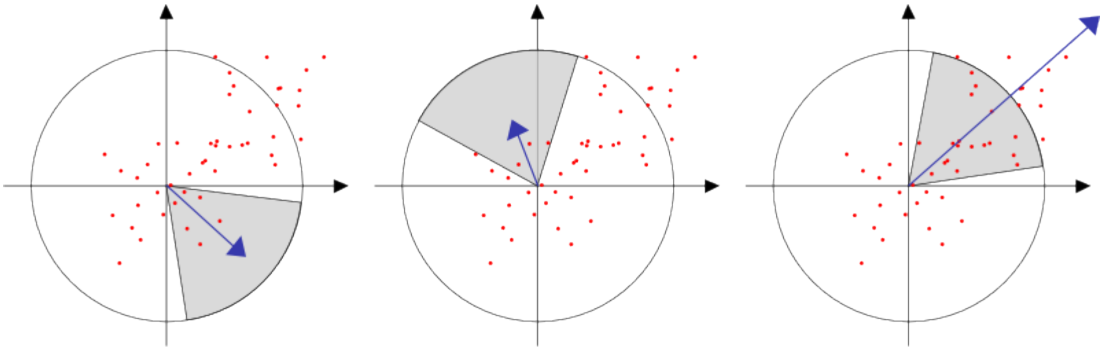

Figure 7: Orientation Assignment: As the window slides around the origin the components of the responses are summed to yield the vectors shown here in blue. The largest such vector determines the dominant orientation.

In some applications, rotation invariance in not required so this step can be omitted, hence providing further performance increase. In [1] this version of the descriptor is reffered to as Upright SURF (or U-SURF) and has been shown to maintain robustness for image rotations of up to $+/-15\,\mathrm{deg}$.

1.3.2 Descriptor Components

The first step in extracting the SURF descriptor is to construct a square window around the interest point. This window contains the pixels which will form entries in the descriptor vector and is of size 20σ, again where σ refers to the detected scale. Furthermore the window is oriented along the direction found in Section 1.3.1 such that all subsequent calculations are relative to this direction.

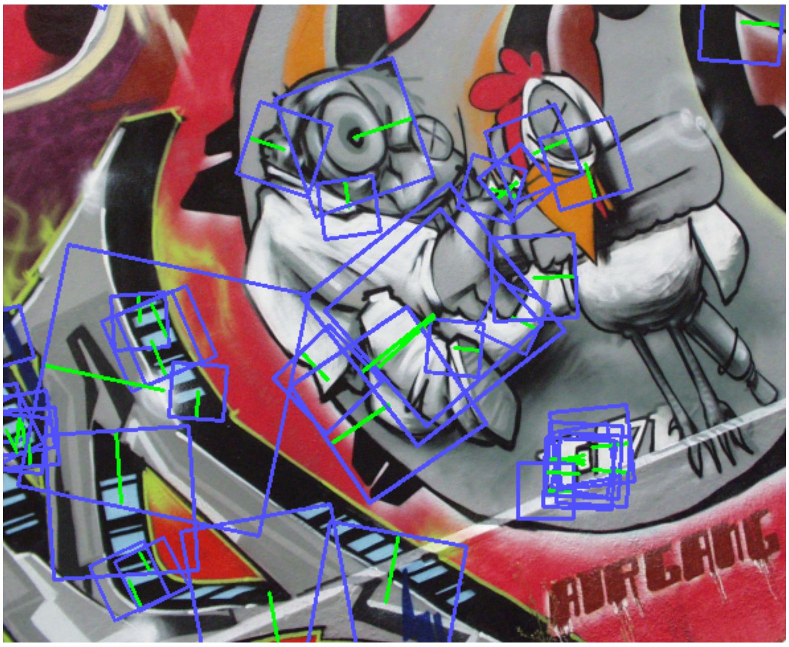

Figure 8: Descriptor Windows. The window size is 20 times the scale of the detected point and is oriented along the dominant direction shown in green.

The descriptor window is divided into 4×4 regular subregions. Within each of these subregions Haar wavelets of size 2σ are calculated for 25 regularly distributed sample points. If we refer to the x and y wavelet responses by dx and dy respectively then for these 25 sample points (i.e. each subregion) we collect,

$$v_{subregion} = \left[\sum dx, \sum dy, \sum |dx|, \sum |dy| \right]. \tag{9}$$

Therefore each subregion contributes four values to the descriptor vector leading to an overall vector of length $4 \times 4 \times 4 = 64$. The resulting SURF descriptor is invariant to rotation, scale, brightness and, after reduction to unit length, contrast.

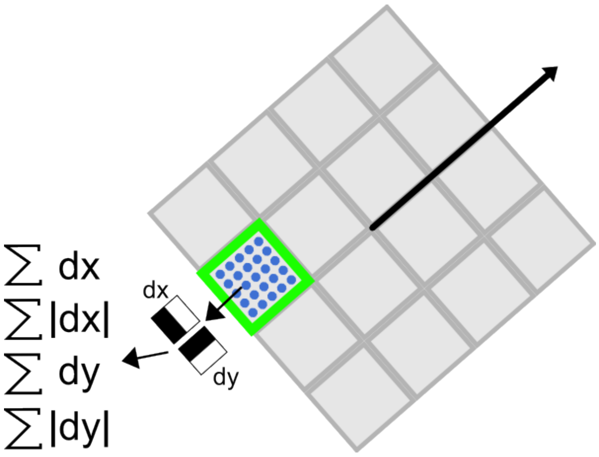

Figure 9: Descriptor Components. The green square bounds one of the 16 subregions and blue circles represent the sample points at which we compute the wavelet responses. As illustrated the x and y responses are calculated relative to the dominant orientation.

1.4 Design

This section outlines the design choices which have been made to implement the SURF point correspondence library.

1.4.1 Language and Environment

C++ has been chosen as the programming language to develop the SURF library for the following reasons:

1. Speed: Low level image processing needs to be fast and C++ will facilitate the implementation of a highly efficient library of functions.

2. Usability: In my research, almost all image processing appears to be carried out in C++, C and Matlab. Once complete the library is to be fully documented and freely available, so C++ seems the obvious choice in making a useful contribution to the field.

3. Portability: While C++ may not be entirely portable across platforms it is possible, by following strict standards, to write code which is portable across many platforms and compilers.

4. Image Processing Libraries: OpenCV[1] is a library of C++ functions which lends itself well to real time computer vision. It provides functionality for reading data from image files, video files as well as live video feeds direct from a webcam or other vision device. The library is well supported and works on both Linux and Windows.

The chosen development environment for the implementation of the library is Microsoft Visual C++ Express 2008[2]. VC++ is a powerful IDE (Integrated Development Environment) allowing for easy code creation and visual project organisation. OpenCV also integrates well with the compiler and is fully supported. As both OpenCV and Visual C++ are free, this will allow the finished SURF library to be distributed without licensing restrictions.

1.4.2 Architecture Design

The architecture design provides a top-down decomposition of the library into modules and classes.

[1] Open Source Computer Vision Library. Provides a simple API for working with images and videos in C++. Available from: http://opencvlibrary.sourceforge.net/

[2] Free C++ IDE for Windows. Available from: www.microsoft.com/express/

Integral Image
Overview: Module creates and handles Integral Images
Inputs: An Image
Processes: Creates the integral image representation of supplied input image. Calculates pixel sums over upright rectangular areas.
Outputs: The integral image representation.

Fast-Hessian
Overview: Finds hessian based interest points/regions in a given integral image.
Inputs: An integral image representation of an image.
Processes: Builds determinant of Hessian response map. Performs a non maximal suppresion to localise interest points in a scale-space. Interpolates detected points to sub-pixel accuracy.
Outputs: A vector of accurately localised interest points.

SURF Descriptor:
Overview: Extracts descriptor components for a given set of detected interest points.
Inputs: An integral image representation of an image, vector of interest points.
Processes: Calculates Haar wavelet responses. Calculates dominant orientation of an interest point. Extracts 64-dimensional descriptor vector based on sums of wavelet responses.
Outputs: A vector of 'SURF described' interest points.

Interest Point:
Overview: Stores data associated with each individual interest point.
Inputs: Interest Point data.
Processes: Accessor/Mutator Methods for data.
Outputs: None

Utilities:
Overview: Module contains all non SURF specific functions.

1.4.3 Data Structures

There are two important non-standard data structures which will be included in the implementation of the SURF library. These are the IplImage which is used by OpenCV to store image data, and the user defined Ipoint which will store data about interest points. As the IplImage is supplied by an external library, it is not included in the design of this work. The Ipoint data type and its interface is detailed in Section 1.4.4.

1.4.4 Interface Design

The class/function interfaces are described for each of the components listed in Section 1.4.2, where the interface refers to the public components which can be called from external modules.

Listing 1: Integral Image Interface

```
//! Returns integral image representation of supplied image
IplImage *Integral(IplImage *img);

//! Calculates sum of pixel intensities in the area
float Area(IplImage *img, int x, int y, int width, int height);
```

Listing 2: Fast-Hessian Interface

```
FastHessian {

  //! Constructor
  FastHessian(IplImage *img,
              vector<Ipoint> &ipts,  // Vector to store Ipoints
              int octaves,  // Number of Octaves to analyse
              int intervals,  // Number of Intervals per Octave
              int sample, // Sampling step in the image
              float thres);  // Hessian response threshold

  //! Find the image features and write into vector of features
  void getIpoints();

};
```

Listing 3: SURF Descriptor Interface

```
Surf {

  //! Constructor
  Surf(IplImage *integral_img);

  //! Set the Ipoint of interest
  void setIpoint(Ipoint *ipt);

  //! Assign the supplied Ipoint an orientation
  void getOrientation();

  //! Get the descriptor vector of the provided Ipoint
  void getDescriptor();

};
```

Listing 4: Interest Point Interface

```
Ipoint {

  //! Coordinates of the detected interest point
  float x, y;

  //! Detected scale
  float scale;

  //! Orientation measured anti-clockwise from +ve x-axis
  float orientation;

  //! Vector of descriptor components
  float descriptor[64];

};
```

1.5 Implementation

Based on the theoretical analysis of Sections 1.2 and 1.3, and the design in Section 1.4, the resulting implementation of the SURF algorithm is presented. This section serves to provide further insight into SURF whilst simultaneously making clear the concepts which are not explicitly defined in [1]. For each component the methods which have been used are demonstrated from algorithmic approach with excerpts of code provided where necessary.

1.5.1 Integral Images

Computation of the Integral is a very simple process. We iterate over all rows and columns such that the value in the integral image at each point is the sum of pixels above and to the left. To implement this in an efficient manner we calculate a cumulative row sum as we move along each row. For the first row this sum forms the value at each location in the resulting integral image. For each subsequent row the resulting value is the cumulative row sum plus the value in the cell above in the integral image. In simplified C++ code this is written as:

Listing 5: Integral Image Computation

```
// Loop over rows and cols of input image
for(int row=0; row<height; row++) {
  row_sum = 0;
  for(int col=0; col<width; col++) {

    // Calculate cumulative row sum
    row_sum = row_sum + source_data[i][j];

    // Set value in integral image
    if (row > 0)
      integral_data[i][j] = integral_data[i-1][j] + row_sum;
    else
      integral_data[i][j] = row_sum;
  }
}
```

1.5.2 Fast-Hessian

The first step in the Fast-Hessian process is to construct the blob-response map for the desired number of octaves and intervals. This is implemented as a 2-dimensional array of image structures of size octaves×intervals. Therefore if we wish to analyse a scale-space comprised of three octaves and four intervals, as is the default setting, we create an array of twelve image structures. The values for each octave-interval pair dictate the size of the filter which will be convolved at that layer in the scale-space. The filter size is given by the formula,

$$\text{Filter Size} = 3\left(2^{\text{octave}} \times \text{interval} + 1\right). \tag{10}$$

The expected filter size for the first interval in the first octave is the base 9 × 9 filter as detailed in Section 1.2.2. Using the formula with the the values octave = interval = 1 yields,

$$\text{Filter Size} = 3\left(2^1 \times 1 + 1\right) = 9.$$

Using this formula we can both construct and convolve the filters at each level in the scale-space automatically. In order to avoid edge effects in the convolution, we set a border for each octave such that the largest filter in the octave remains entirely over the image. This is important as edge effects can yield anomolous results which reduce the reliability of the detector. Once the responses are calculated for each layer of the scale-space, it is important that they are scale-normalised. Scale-normalisation is an important step which ensures there is no reduction in the filter responses as the underlying Gaussian scale increases. For the approximated filters, scale-normalisation is achieved by dividing the response by the area. This yields near perfect scale invariance. The code listing below demonstrates a stripped down approach to creating the response map.

Listing 6: Constructing the Scale Space

```
for(int o=0; o<octaves; o++) {
  // Calculate filter border for this octave
  border = (3 * pow(2,o+1)*(intervals)+1) + 1)/2;

  for(int i=0; i<intervals; i++) {

    // Calculate lobe length (filter side length/3)
    lobe = pow(2,o+1)*(i+1)+1;

    // Calculate area for scale-normalisation
    area = pow((3*lobe),2);

    // Calculate response at each pixel in the image
    for(int r=border; r<height-border; r++) {
      for(int c=border; c<width-border; c++) {

        // Caclulate scale-normalised filter responses
        Dyy = [Filter Response] / area;
        Dxx = [Filter Response] / area;
        Dxy = [Filter Response] / area;

        // Calculate approximated determinant of hessian value
        detHess = (Dxx*Dyy - 0.9f*0.9f*Dxy*Dxy);
      }
    }
  }
}
```

The search for extrema in the response map is conducted by a non-maximal suppression. The implementation of this is very straightforward. Each pixel is compared to it's neighbours and only those which are greater than all those surrounding it are classified as interest points. This cost of checking neighbouring values is relatively low as many pixels will be determined non-maximal after only a few checks.

The interpolated position and scale of the detected interest points is found by the formula proposed by Brown [3]. This can be computed by a few simple matrix multiplications. To find the interpolated location, $\hat{x} = (x, y, \sigma)$, in scale and space we need to calculate the result of,

$$\hat{x} = -\frac{\partial^2 H}{\partial \mathbf{x}^2}^{-1} \frac{\partial H}{\partial \mathbf{x}}.$$

To do this we need to first compute the entries in the 3×3 matrix $\frac{\partial^2 H}{\partial \mathbf{x}^2}$ and the 3×1 vector $\frac{\partial H}{\partial \mathbf{x}}$. These entries are computed by finite differences of pixel intensities and the specific values are given by,

$$\frac{\partial^2 H}{\partial \mathbf{x}^2} = \begin{bmatrix} d_{xx} & d_{yx} & d_{sx} \\ d_{xy} & d_{yy} & d_{sy} \\ d_{xs} & d_{ys} & d_{ss} \end{bmatrix}$$

and,

$$\frac{\partial H}{\partial \mathbf{x}} = \begin{bmatrix} d_x \\ d_y \\ d_s \end{bmatrix}.$$

Here d_x refers to $\frac{\partial I}{\partial x}$, d_{xx} refers to $\frac{\partial^2 I}{\partial x^2}$ and likewise for the other entries. Using these two matrices a simple inversion and matrix multiplication yields the interpolated location. Each interpolated interest point is added to a vector which is the final output of the Fast-Hessian class.

1.5.3 Descriptor

As detailed in the technical summary of SURF, the first step in describing the interest points found by the detector is to extract the dominant orientation. In the implementation this is broken into two stages with the first calculating Haar wavelet responses around the point and the second iterating over the region to compute the responses which fall within a sliding circle segment.

Computing the responses is a trivial task. By considering a square region of size 12σ centered on the interest point, we calculate only those responses whose distance is less than 6σ from the center. These responses are weighted by the Gaussian function and added to a vector. The code listing below demonstrates the approach used in the implementation of this step.

Listing 7: Calculating Haar Responses

```
for(int x = -6*s; x <= 6*s; x+=s) {
  for(int y = -6*s; y <= 6*s; y+=s) {

    // Check whether current sample point is within circle
    if( x*x + y*y  < 36*s*s ) {

      // Calculate Gaussian weight
      gauss = gaussian(x, y, 2*s);

      // Add response to vectors
      resX.push_back( gauss * haarX(x, y, 4*s) );
      resY.push_back( gauss * haarY(x, y, 4*s) );
    }
  }
}
```

To calculate the responses within the sliding window, we step through a discrete set of angles between 0 and 2π and compute the sum of the responses which form an angle inside the window. This procedure is clear when looking at a simplified version the code.

Listing 8: Sliding Window Sums

```
// Loop slides pi/3 window around feature point
for(ang1 = 0; ang1 < 2*pi; ang1+=0.1) {
  sumX = sumY = 0;

  // Loop iterates over all Haar responses
  for(unsigned int i = 0; i < resX.size(); i++) {

    // Get angle from the x-axis of the response
    ang = getAngle(resX.at(i), resY.at(i));

    // Determine whether the point is within the window
    if (ang1 < ang && ang < ang1+pi/3) {
      sumX+=resX.at(k);
      sumY+=resY.at(k);
    }
  }

  // If the vector produced from this window is longer than all
  // previous vectors then this forms the new dominant direction
  if (sumX*sumX + sumY*sumY > max) {
    max = sumX*sumX + sumY*sumY;
    orientation = getAngle(sumX, sumY);
  }
}
```

Extraction of the descriptor is implemented in a very similar manner to the response calculation for orientation assignment. Instead of a circular regions, the descriptor extracts responses from a square window. In the code, one loop iterates over each of the sixteen subregions while another calculates the 25 wavelet responses within each. Since the windows are oriented along the dominant direction, and the Integral image only retrieves upright rectangular areas, Bay [1] proposes interpolating the responses to remain invariant to rotation. In this work, rather than interpolating the responses, the descriptor windows are first rotated so that the dominant orientation aligned with the positive y-axis. There is a small sacrifice in computation time but the results have proved more robust (See Results Section). Both versions of the descriptor are included in the library. The code listing below demonstrates the method for extracting the four vector components ($[\sum dx, \sum dy, \sum |dx|, \sum |dy|]$) from each subregion.

Listing 9: Extracting Descriptor Components

```
// Outer loops select top left corners of each subregion
for (int i = 0; i < 20*s; i+=5*s)
  for (int j = 0; j < 20*s; j+=5*s) {

    // These values store response sums for the subregion
    dx=dy=mdx=mdy=0;

    // Inner loops select 25 sample points from subregion
    for (int k = i; k < i + 5*s; k+=s) {
      for (int l = j; l < j + 5*s; l+=s) {

        // Compute Gaussian weighted responses
        gauss = gaussian(k-10*s, l-10*s, 3.3f*s);
        rx = gauss * haarX(k, l, 2*s);
        ry = gauss * haarY(k, l, 2*s);

        // Sum the responses and the their absolute values
        dx += rx;
        dy += ry;
        mdx += fabs(rx);
        mdy += fabs(ry);
      }
    }
  }
```

2 Results

In this section we test the SURF library using an image dataset provided by Kristian Mikolajczyk[3]. This dataset contains sequences of images which exhibit real geometric and photometric transformations, such as scaling, rotation, illumination and JPEG compression. Comparing the results of various detector-descriptors is a complex task and a full body of work in itself. Mikolajczyk [5, 6] compared the most widely used detectors and descriptors and Bay [1] used the same testing strategy to show how SURF outperforms its predecessors. The following sections therefore detail the results of this SURF library in isolation.

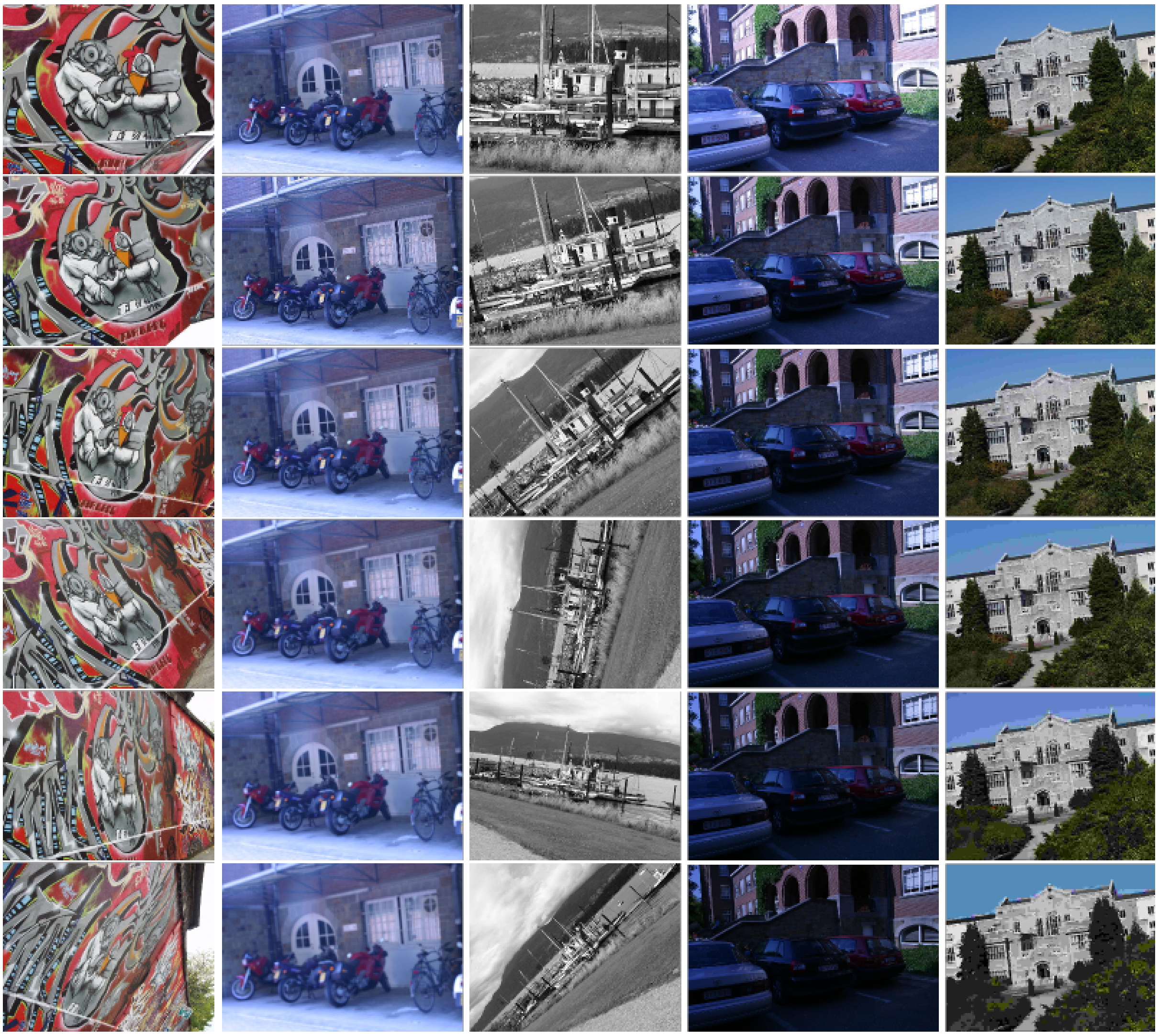

Figure 10: Test Dataset. Columns left to right: Graffiti with viewpoint rotation, Bikes with blur, Boats with optical axis rotation, Leuven with brightness and UBC with JPEG compression.

[3] Dataset acquired from http://www.robots.ox.ac.uk/∼vgg/research/affine/

2.1 Detector Results

The criteria used to evaluate the performance of the detector is the repeatability. This is an important measure of a detector's performance as it refers the likeliness of interest points being detected under varying image transformations. More specifically it measures the percentage of geometrically correct correspondences as found in images of the same scene or object under different viewing conditions. Geometrical correctness refers to the point being localised on the same image structure or region and can be measured in a number of ways. A simple approach is to evaluate the correspondences by eye, but this is laborious and lacks robustness. A better method, and the one used in these tests, is to relate the points by a homography.

$$\hat{x} = H \cdot x. \tag{11}$$

Here H is the 3×3 homography matrix which maps x-points in one image to $\hat{x}$-points in another. Using this function we can determine whether a point has a correspondence by performing the mapping operation and checking for detected points within a neighbourhood of the target location. For the purpose of the tests carried out in this work the neighbourhood is set at 1.5 pixels. Furthermore, the image dataset (as shown in Figure 10) is supplied with accurate homography matrices which relate the points to a high degree of accuracy.

The following graphs illustrate the repeatability rate under the various transformations. The detector performs well in these tests, with the exception of the Graffiti scene where the viewpoint change introduces an out of plane transformation. This is an expected result as SURF is not designed to be affine invariant.

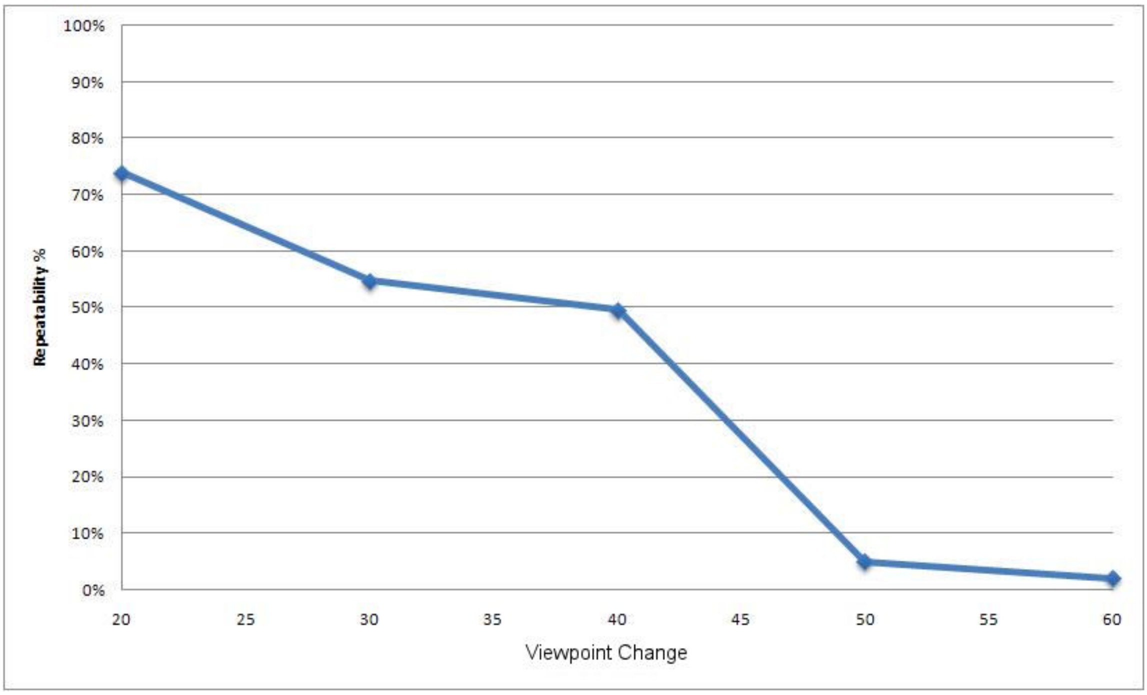

Figure 11: Repeatability under viewpoint change.

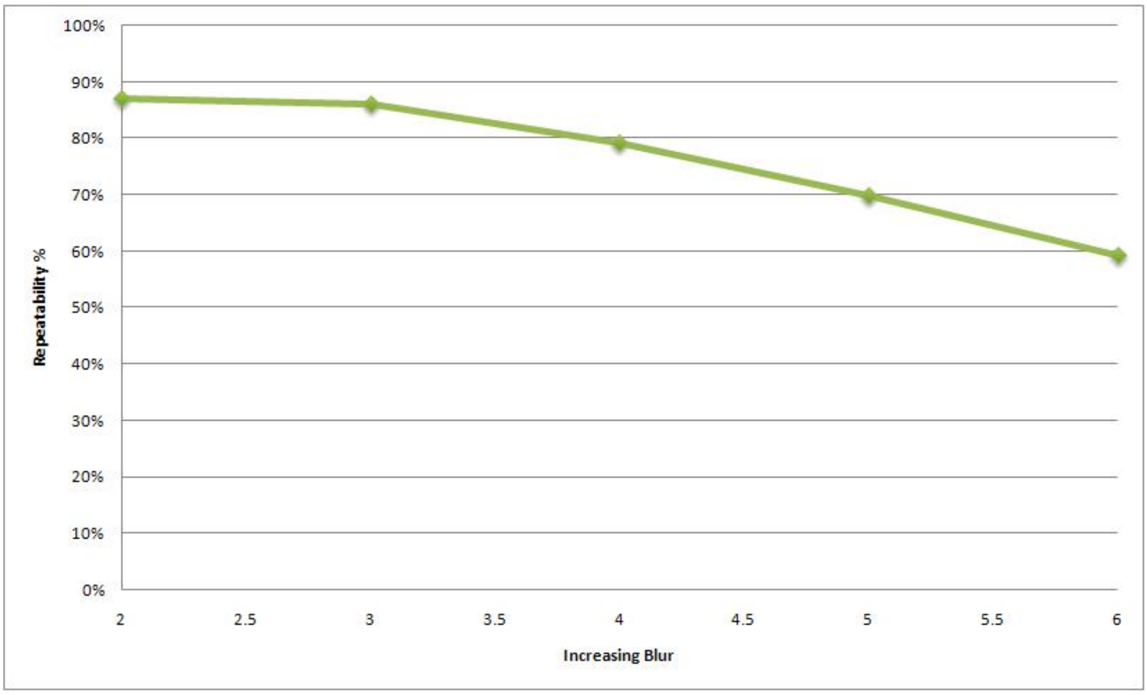

Figure 12: Repeatability under image blurring.

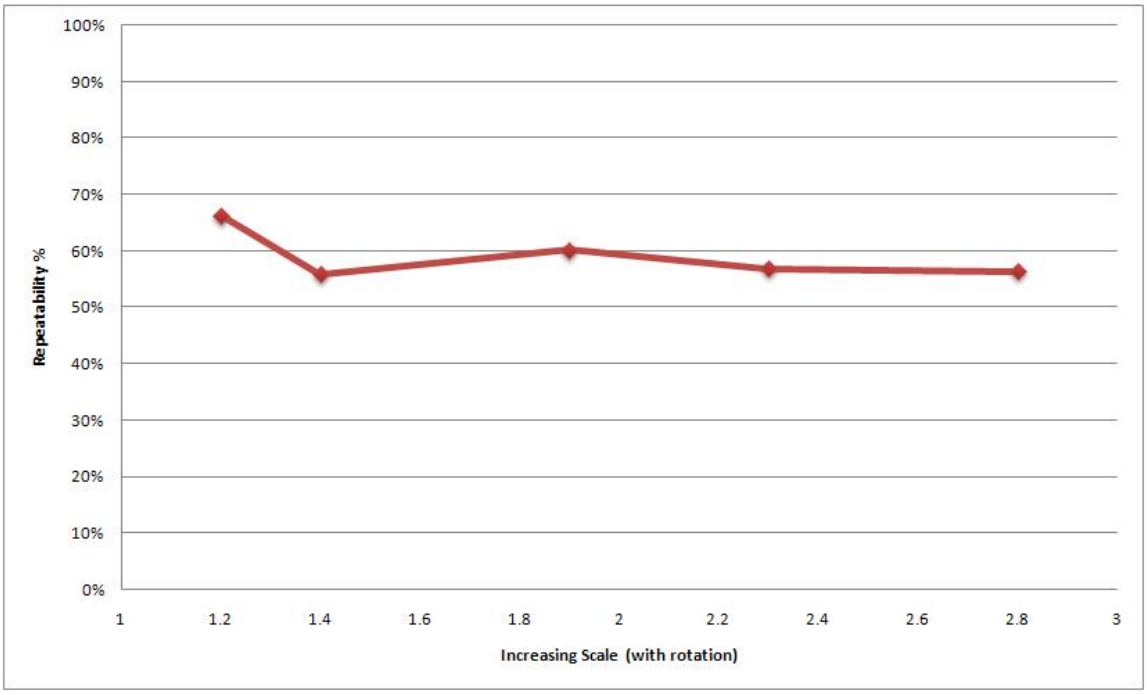

Figure 13: Repeatability under rotation about the optical axis with scale change.

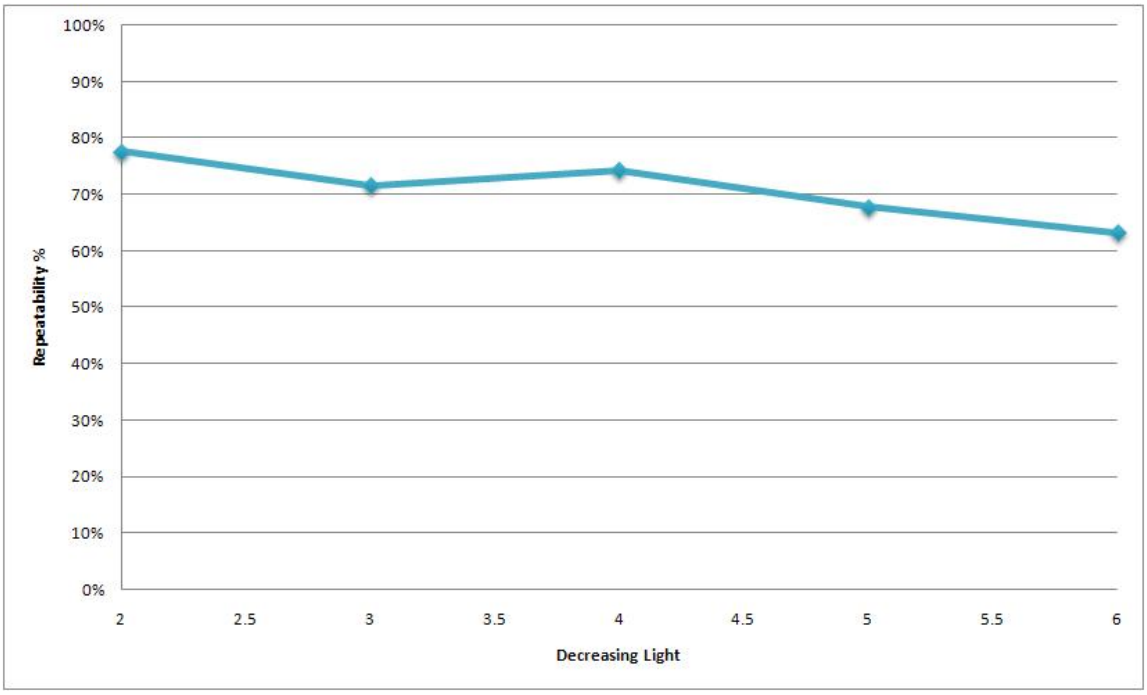

Figure 14: Repeatability under decreasing image brightness.

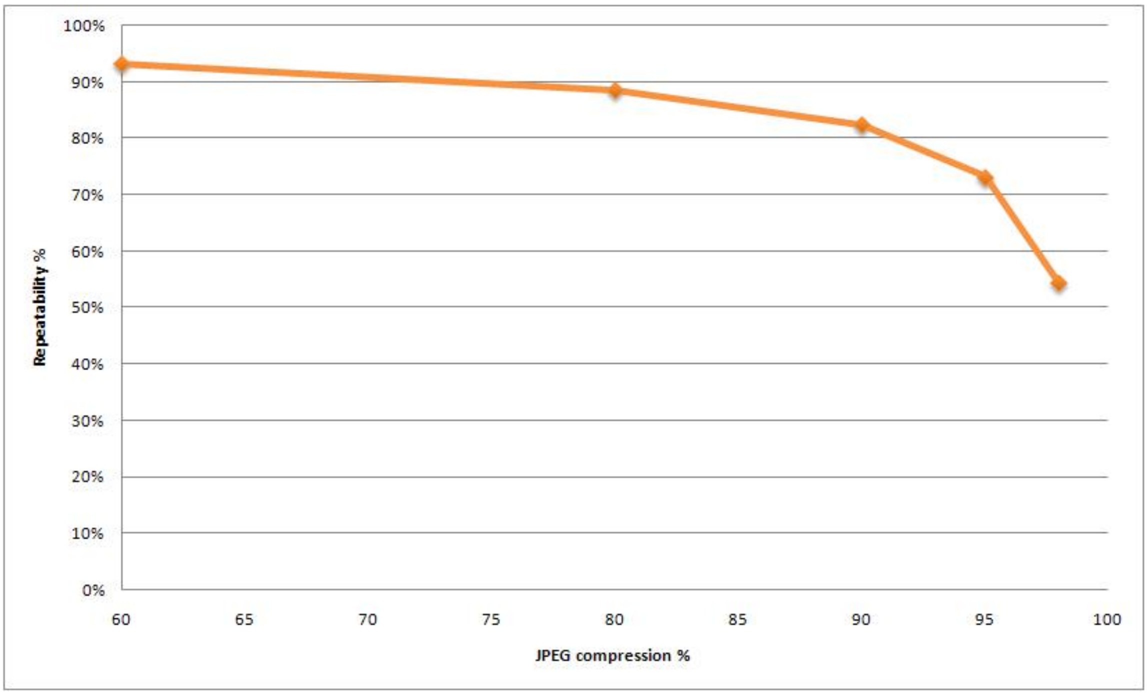

Figure 15: Repeatability under JPEG compression.

References

[1] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *European Conference on Computer Vision*, 1:404–417, 2006.

[2] P.R. Beaudet. Rotationally invariant image operators. *International Joint Conference on Pattern Recognition*, 579:583, 1978.

[3] M. Brown and D.G. Lowe. Invariant features from interest point groups. *British Machine Vision Conference, Cardiff, Wales*, pages 656–665, 2002.

[4] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[5] K. Mikolajczyk and C. Schmid. An affine invariant interest point detector. *Proc. ECCV*, 1:128–142, 2002.

[6] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(10):1615–1630, 2005.

[7] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *cvpr*, 1:511, 2001.

[8] A. Witkin. Scale-space filtering, int. joint conf. *Artif. Intell*, 2:1019–1021, 1983.