Slovenská technická univerzita
v Bratislave
Stavebná fakulta

Študentská vedecká konferencia
Akademický rok 2011/2012

Extrakcia dobre popísateľných bodov z obrazov a ich porovnávanie
Extraction and matching of feature points in bitmaps

Meno Priezvisko študenta:	Peter Kottáš
Ročník a program/odbor štúdia:	3.ročník, matematicko-počítačové modelovanie
Vedúci práce:	Mgr.art., Mgr. Ladislav Šipeky
Katedra:	KMDG

Bratislava 18. apríl 2012

Extraction and matching of feature points in bitmaps

Peter Kottáš; petokottas@gmail.com

Introduction

We are going to cover the problem of feature points recognition in bitmaps. Follow-up to this problem is the usage of the database as a source of potential matches. This approach brings us closer to understand object recognition. Main purpose of this article is to introduce variety of steps required for basic feature recognition application to work. Main idea was thought out by David G. Lowe and is covered in "Object recognition from local scale-invariant features", International Conference on Computer Vision, Corfu, Greece (September 1999). This document contains huge amount of theoretical information necessary to fully understand whole problem. However there are parts that might be confusing, especially during algoritmization process. I would like to create a user-friendlier description of this state-of-art idea and introduce few of my observations and potential improvements that I thought-off during implementation. This paper might be as well considered to be jumping-board for computer vision itself. In fact I expect it to lay ground work for understanding feature recognition at all levels. Hopefully it will make it possible to introduce my own solutions for numerous problems still present in computer vision. I consider this paper to be a guideline for a person with no prior knowledge on the topic of feature recognition. Still I want it to be as fluent as possible. As a consequence, the first chapter is devoted solely to build up foundation of terms one should be familiar with before going further inside the subject.

1. Familiarization of technicalities

1.1 Feature point

Co-ordinates in bitmap that refers to point in R2 as a projection from R3. What we are trying to achieve is reliable recognition of these R3 points in projections (bitmaps) that have different properties. Our features should be rotation and scale invariant. It is although required for features to be illumination and viewpoint invariant, however this is only approximation. It is understandable that under certain kinds of lighting conditions (complete darkness) and or viewpoint orientation (close to 90 degrees) feature points would not be extracted correctly.
This is the first time probability comes into mind of perceptive reader. The better the input image (less illumination or viewpoint changes) the higher probability of recognizing stable features thus higher probability to find exact matches later on.

1.2 Feature descriptor

Feature point is described only by its R2 coordinates. Given two sets of feature points, one from each image (or one being the database), we need some tool to identify possible feature points corresponding to the same R3 point in real world. Descriptor is the desired tool represented as multidimensional vector. This vectors entries must be as invariant as feature points themselves. To compare two descriptors Euclidean distance is used. The less the distance between two descriptors, the higher the probability that it is the same R3 point projected in two R2 planes.

1.3 Epipolar geometry

As described in Richard Hartley and Andrew Zisserman (2003), epipolar geometry is the geometry of stereo images. Consider 3D scene with two pinhole cameras both looking at same R3 point from different viewpoints. Plane described by two centers of projection of each individual camera and R3 point is called Epipolar plane. This plane intersects each camera`s image plane in line called Epipolar line. Epipole is the center of projection of camera 2. as seen from camera 1. Fundamental (F) or essential (E) matrix is used to describe relation between two images while encoding epipolar geometry.

$$E = K'^{T} F\, K \tag{1}$$

Where $\boldsymbol{K}'$ and $\boldsymbol{K}$ are the intrinsic calibration matrices of the two images involved.

$$x'^{T} F\, x = 0 \quad x'^{T} E\, x = 0 \tag{2}$$

$\boldsymbol{x}$ and $\boldsymbol{x}'$ being projections of single point in R3 to stereo images. For indefinite $\boldsymbol{x}'$ (2) gives us one equation of two unknowns, Epipolar line equation. Therefore if F or E is known for static scene it is possible to reject false matches based on distance from epipolar line.

1.4 Ransac

Ransac is an algorithm used to fit model to the database of data points. Consider a simple example:
Dataset of *n* points is provided and we would like to use Ransac to fit line in that set of data points. Initial image shows points plotted on R2 plane. Other two images show two realizations of Ransac main loop.

1. 4.1 Ransac main loop

Chose random *m* points from dataset (*m* is number of points required for model fitting, e.g. line needs two points to be fit through). Fit model to the *m* chosen points. Iterate thought the rest of the dataset and decide if approximation of the model is good enough for the checked point. In this example distance from the line must be less than some chosen constant for the point to be an "inlier" shown in blue. Otherwise the point is considered to be "outlier" shown in red. This process is repeated *k* times and the best result is kept in artificial variable. At the end the best model stored is considered to be the best approximation. These steps are briefly described in the following fig.1.

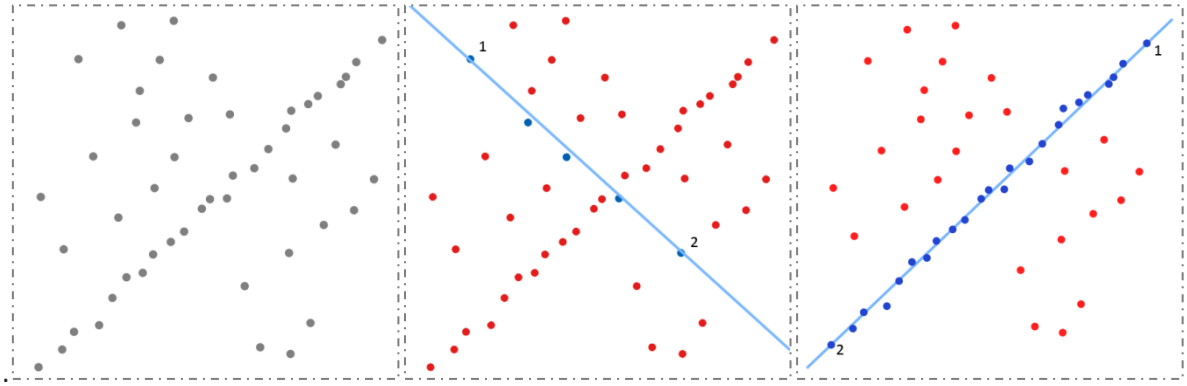

Fig.1: Ransac algorithm visualization

To compute fundamental or essential matrix we need as many as 8 points so the Ransac is modified for the model of "solving epipolar geometry" and outliers are then rejected as false matches. F or E could be solved by Eight-Point Algorithm which we are not going to cover it in this paper.

1.5 Gaussian blur

Gaussian blur, widely known as Gaussian smoothing, is filter that convolves image according to Gaussian function as enumerated around the interest point. A Gaussian blur effect is typically generated by convolving an image with a kernel of Gaussian values. These values are results of:

$$G(x,y) = \frac{1}{2\pi\sigma^2} * e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

Values are computed for each surrounding pixel. Convolution area´s (square) width in pixels is equal to *6*σ+1*. (3) plotted as 3D object shows that it consists of concentric circles. This property is known as rotation invariance. That means that the process of convolving could be split in two passes, horizontal and vertical. Advantage of this approach is described in this example. Consider Gaussian smoothing with σ=1. Kernel dimension is 7 times 7 giving

49 entries. If we would not implement rotation invariance optimization, single blurred pixel in image L(x,y) would be computed as product of,

$$L^{blurred}(x,y) = \sum_{u=x-3}^{x+3} \sum_{v=y-3}^{y+3} L(u,v) * G(u-x+3, v-y+3)$$

resulting in 49 floating-point multiplications and additions. We only need one dimensional kernel thanks to the proposed optimization.

$$G(x) = \frac{1}{2\pi\sigma^2} * e^{-\frac{x^2}{2\sigma^2}} \tag{4}$$

This kernel has 7 entries. Pixel in image L(x,y) is convolved as,

$$L^{vertical\_blur}(x,y) = \sum_{v=y-3}^{y+3} L(x,v) * G(v-y+3)$$

$$L^{horizontal\_blur}(x,y) = \sum_{u=x-3}^{x+3} L(u,y) * G(u-x+3)$$

resulting in 14 multiplications and additions. Benefit is even more noticeable when using large σ values. (e.g. : sigma=5 , 961 operations as opposed to 62)

1.6 Scale space

Scale space is a theory developed while studying signal processing. It was later adapted by many as the only reliable tool for achieving scale invariance in image recognition. Our implementation is based on Lindeberg (1994). Main problem that occurs while studying unknown scene is that we have no prior information about the scale of the image. Consider the following scenario: Two bitmaps are studied for possible feature matches. First image is a photography of a single tree. The other shows whole forest in the distance. There are a lot of fine details in the first image (e.g.: Leaves, branches, bark texture etc.) that vanished from the second projection because of loss of data during encoding process. For the application to produce scale invariant features this has to be dealt with. It would be useless if features points are only matched correctly if taken from same distance. This is where the scale space approximation takes place. In our example we have to remove some of the details from the first image so that feature descriptors would be as close to the ones from the second image as possible. Gaussian smoothing was proved to be the best approximation of scale space representation.

Hopefully this example made things clear enough for the formal encapsulation of the facts to take place. Scale space is the representation of an image L(x,y) as a one-parameter family of smoothed images $L^{new}(x,y,\sigma)$, parameterized by the size of the smoothing kernel ((3) in our implementation). Following fig.2 shows 4 progressively blurred images. Bitmap used in our implementation is black and white representation of well known test image called Lenna. This test image scanned in June 1973 has become standard in testing of many image processing algorithms for over 30 years.

Fig.2: Example of the scale space (σ={1,2,4,8})

2. Feature point detector

2.1 Introduction to features extraction

We already know what feature points are in computer vision, now we are going to take a further look at how to extract them. We want features to be as stable and scale invariant as possible. There is a number of approaches available, corner, edge or blob detection just to name a few. I have chosen local extrema of difference of Gaussian from scale space representation for my project. It has been proved to produce high quality feature points (easily recognizable and stable under variety of changes) while sacrificing computational speed. Corner detection would be better choice if our primary goal was the speed of application. Real-time recognition was not desired in our case so we focused on quality rather than speed. Still we are able to match features from bitmaps of medium resolution (1024 × 768) in around a second. This was achieved by C++ code optimization as well as designing an application to exploit full potential of multi-core CPU (multi-threading). However main purpose of this project was not to achieve best optimization possible, it was created as a step-by-step guideline for developing application while noting potential improvements for the future releases.

2.2 Scale space representation for image recognition

Scale space, briefly described in chapter 1.6 as a representation of image $L(x,y)$as a one-parameter family of smoothed images $L(x,y,\sigma)$, parameterized by the size of the smoothing kernel. It is clear that some sort of approximation have to take place while implementing such phenomenon for computation on hardware with limited performance. Full scale space representation consists of infinite number of images $L(x,y,\sigma)$ as convolved by Gaussian kernels with $\sigma \in \langle 0,\infty\rangle$. In our implementation cascade filtering was chosen to approximate true scale space.

Cascade filtering consists of octaves and scales. Before focusing on it let me show where it originates and why it is not only an approximation but optimization of process as well. Consider simple scale space as seen in Fig.2. This is in fact non cascade representation of consequently blurred images (take note of the parameter choice, σ={1,2,4,8}, as $\sigma_1 = initialsigma$; $\sigma_i = \sigma_{i-1} * k$ where k≥0 is scale-change factor and i >1, we are going to focus on that later on). To make it cascade we need to take advantage of one of the key properties of scale space representation.

Consider image $L(x,y,\sigma)$. By downscaling with a factor of n, parameter sigma becomes:

$$\sigma^{new} = \sigma^{old} * n. \tag{5}$$

Potential for optimization of this property is hidden in chapter 1.5. Gaussian blur is relatively fast to compute for kernel with small σ. In our example Gaussian kernel of image $L(x, y, \sigma^{old})$ has width of $6 * \sigma^{old} + 1$. Therefore image downscaled by factor *n* with kernel width of $(6 * \sigma^{old})/n + 1$ produces close to same results while computational time is reduced radically. It would need some higher mathematics to take place in order to fully understand this property. Rather than that I am going to use empiric approach. For that we need to realize that parameter ***σ*** is nothing more than numeric interpretation of the scale of the scene. Now consider perspective projection. Computer vision tries to copy the way object recognition works in biology. Therefore there is no accident that perspective projection in camera (ignoring calibration) works very similar to the way human eye works. Perspective itself is described as projection from R3 Space to R2 Plane. As opposed to orthographic, it does not maintain ratio for abscissas in different distances. Consequently objects that are further away looks smaller than the ones closer to the encoding device. So ***σ*** of the object in background is bigger than the one in foreground. By downscaling we are meddling with perception of the scale therefore σ is directly affected.

Now we are ready to proceed to cascade filtering. In our implementation first a table of σ is created for octaves. One octave is set of progressively blurred out images with the same width and height. Size of this set is at least 4 while 5 or 6 images in each cascade has proven to suffice for good-enough scale space approximation. σ parameters of these images differs from the previous one by factor *k*. Initial sigma is provided. Assuming that that photography already contains some blur initial sigma is set to $1/\sqrt{2}$ in our algorithm. Best results were achieved by using parameter $\mathrm{k} = \sqrt{2}$, however both initial sigma and parameter *k* have such great effect on the process that it is highly recommended for everybody to experiment a bit, just to see the impact of different values on the feature points. Based on previously alluded information sigma table for first octave is built as follows in Table.1:

Table.1: Example of σ values for one octave.

	σ_1	σ_2	σ_3	σ_4	σ_5
$Octave_1$	$2^{-0.5}$	1	$\sqrt{2}$	2	$2 * \sqrt{2}$

Other octave is created from the initial image by downscaling it into half of its original size. Lowe D. G. (2004) suggested optimization in downscaling process. Since factor of downscaling is always 2 his idea was to leave out every other pixel during the process achieving desired effect. However while experimenting with variety of rotated bitmaps this proved to discard too much information and produce aliasing so in this implementation integral downscaling was used instead. Destination pixel in downscaled image is projected as a square back to the original bitmap where it is used as area of integration. For images of *width=u*2* and *height=v*2* (*v,u* > 0) every destination pixel contributes with 4 floating point additions and multiplications. Otherwise process is a bit more complex but it is only done once for every octave so the computational time/quality ratio is quite good. Table of sigma-s for remaining octaves is built. Table. 2 shows example of σ values for the whole pyramid. These are actual scale space representation parameters. Hopefully it is clear that for Gaussian smoothing every octave is convolved by the kernels created from values of first row (based on (5)).

$$\sigma_{i,j} = \sigma_{\mathrm{initial}} * \mathrm{k}^{\mathrm{i}-1} * \mathrm{j} \tag{6}$$

Table.2: Example of σ values for the whole pyramid.

	$\sigma_{i,j}$ $i = 1$	$\sigma_{i,j}$ $i = 2$	$\sigma_{i,j}$ $i = 3$	$\sigma_{i,j}$ $i = 4$	$\sigma_{i,j}$ $i = 5$
$Octave_j;\ \ j = 1$	0.707	1	1.414	2	2.828

$Octave_j;\ \ j=2$	1.414	2	2.828	4	5.657
$Octave_j;\ \ j=3$	2.828	4	5.657	8	11.314
$Octave_j;\ \ j=4$	5.657	8	11.314	16	22.627

Number of octaves is once again up to programmer. Experimental results proven 4 or 5 octaves to be satisfactory however my algorithm uses simple decision equation to determine desired pyramid "depth".

```
number_of_octaves=log(min(width,height))/log(2)-5;          (1)
```

Now we have reached a point where we are able to built scale space pyramid. After implementing and benchmarking on variety of different images we came to conclusion that speed of the application still wasn't as good as desired. Therefore another optimization was introduced.

2. 2.1Incremental filtering:

One of the properties of optimization is to avoid computing one information multiple times. Just by glancing at Fig.4 it was easily deductable that by convolving initial image over and over again by kernel with increasing σ values we are ignoring that. What happens is that image blurred by kernel with σ=a $\mathrm{a} \in \mathbb{R}; a \geq 0$ is already "included" in the one with σ>a. The relation between these two convolutions is best observable under Fourier transform as suggested by Yu Meng (2006). Convolution becomes multiplication under Fourier transform therefore Gaussian function e^{ax^2} transformed by Fourier transform becomes:

$$F_x\left[e^{ax^2}\right](t) = \sqrt{\frac{\pi}{a}} e^{-\frac{\pi^2 t^2}{a}} \tag{7}$$

For the sake of example consider two images one convolved with initial sigma σ_0 and then by σ_i. Second image is convolved by kernel $\sigma = \mathrm{h} * \sigma_0$ where *h* $h \in \mathbb{R}; h > 1$. We want to calculate *h* so that resulting σ of both images will be the same. After transformation based on (7) and reducing we get:

$$e^{-{\sigma_0}^2 t^2} e^{-{\sigma_i}^2 t^2} = e^{-\mathrm{h}^2 {\sigma_0}^2 t^2} \tag{8}$$

After comparison of coefficients of $-t^2$ in (8) we get:

$${\sigma_i}^2 + {\sigma_0}^2 = \mathrm{h}^2 {\sigma_0}^2 \tag{9}$$

And from (9):

$$\sigma_i = \sigma_0 * \sqrt{k^2 - 1} \tag{10}$$

Now the process could be even more simplified. First we convolve image with kernel of initial sigma and use it as a first image of Gaussian pyramid. Others are then products of the preceding image with sigma for Gaussian smoothing taken from (10). After implementation we get following result in less than 0.1 second. It is recommended to upscale the input image by the factor of 2 to enlarge the number of located future points in following chapters. Bilinear filtering is used for the process. Upscaling more than once is not necessary since it does not provide further improvement. Whole pyramid is illustrated in fig. 3.

Fig.3: Example of complete pyramid.

2.3 Difference of Gaussian

The actual position of feature point in bitmap is described as highly distinctive contrast point that is extractable under variety of different transformations. To achieve best properties possible we are going to create Difference of Gaussian (DOG) pyramid from the cascade filtered image sequence in Fig.3. Feature point location is then located in both maximum and minimum of DOG images. For every DOG image two Gaussian blurred images are required.

$$L(x, y, \sigma) \rightarrow first\ image$$
$$L(x, y, k * \sigma) \rightarrow second\ image$$

Dog image is then given as:

$$DOG(x, y, \sigma) = L(x, y, k * \sigma) - L(x, y, \sigma) \tag{11}$$

Therefore it is possible to obtain *n-1* DOG images from Scale space pyramid with *n* scales in every cascade. Local maximum and minimum of DOG image were proven to be the most stable features by Mikolajczyk (2002). DOG is although approximation of Laplacian of Gaussian (LOG) as studied by Lindenberg (1994). Relation between DOG image and LOG ($\sigma^2\nabla^2 G$) is understood from the heat diffusion equation.

$$\frac{\partial \mathrm{G}}{\partial \sigma} = \sigma \nabla^2 \mathrm{G} \tag{12}$$

From (12) it is deductable that $\nabla^2 G$ can be computed from the finite difference approximation to $\partial G/\partial\sigma$, using the difference of consequent images with $\sigma_0 = \sigma$ and $\sigma_1 = \mathrm{k} * \sigma$.

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, k * \sigma) - G(x, y, \sigma)}{k * \sigma - \sigma} \tag{13}$$

Thus:

$$\mathrm{G}(\mathrm{x}, \mathrm{y}, \mathrm{k} * \sigma) - \mathrm{G}(\mathrm{x}, \mathrm{y}, \sigma) \approx (\mathrm{k} - 1) * \sigma^2 \nabla^2 \mathrm{G} \tag{14}$$

Obviously *(k-1)* is same for every image pair and σ^2 is just scaling that does not affect location of local extrema. Prior theory was not necessary for implementation, however I decided to paraphrase it from Lowe D. G. (2004) just to give reader better understanding of complex mathematics behind scale-invariant feature extraction. Whole process is illustrated

in Fig.4. This image is example of one cascade of DOG as computed from consequent images from scale space pyramid. Output has been altered for better visual interpretation. In our application each bitmap is kept as *n* x *1* array of numbers from <0,1>, therefore difference values of these images are from interval <-1,1>. In general these values are close to zero because of nature of Gaussian smoothing as well as of parameter *k* which is relatively small.

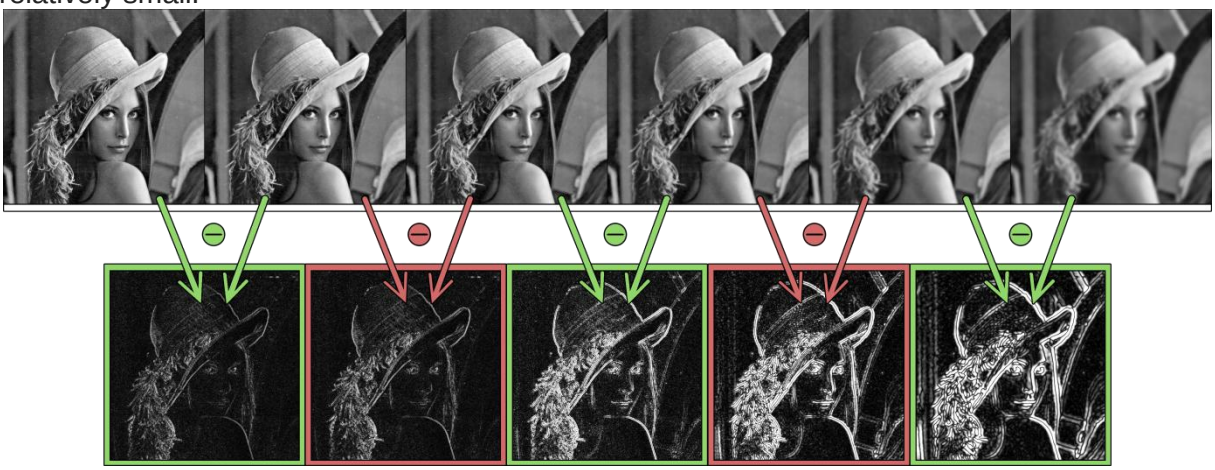

Fig.4: One cascade of DOG images

Every one of these *n-1 DOG images* has potential to produce high quality feature points. One of the possible ways to find local extrema in bitmap is comparing every pixel to its 8-neighborhood. We have successfully located one feature point if the compared pixel´s value is maximum or minimum of surrounding pixels. An improvement have been proposed by Lowe D. G. (2004) that makes located feature points even more stable. Instead of finding extremas in every single image we are going to use three images at once. Therefore first and last image is skipped and features are located as extrema in 26- neighborhood of currently checked pixel (8 pixels from current image and 9 from both preceding and consequent images). Accordingly only *n-3* images are actually used to locate stable feature points. This process is illustrated in Fig.5. This simple concept art shows how feature extraction works. Pixel (symbolized as red cross) is checked to be maximum or minimum of its 26-neighborhood (shown as green circles)

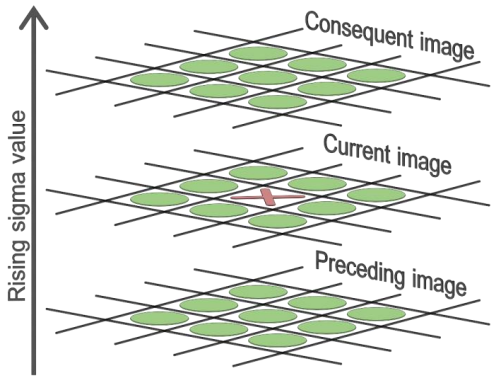

Fig.5: Local extrema of DOG computation

There are 26 floating point comparisons for every pixel and that might seem too much computation. However, bear in mind that majority of pixels are rejected based on first few operations. Process is repeated for every octave. New feature point is created after obtaining extrema. Coordinates of compared pixel becomes coordinates of feature point and sigma parameter is noted as well for later usage during descriptor calculation.

2.4 Sub-pixel accuracy

Section covering sub pixel accuracy was described very poorly in original Lowe D. G. (2004). This is why we spent a lot of time on this part of application. Therefore we decided to provide as much resources as possible. This chapter contains every single equation as well as pseudo code required for successful implementation. Feature point's coordinates extracted from DOG are in fact coordinates of the center of the pixel that represents local extrema. To make our features even more suitable for object recognition or camera resection we might need better definition of its position in R2. Consider R3 point visible in perspective camera. Moreover, it has maximum intensity of all projected points after projected on plane of projection. Plane of projection is in fact our bitmap. Therefore if this point is projected anywhere except center of pixels its value is divided between neighborhood pixels. Probability of projection being located exactly at pixel center is very poor. As a consequence almost every feature point's coordinates gets improved by the following process. Pixel data as encoded by camera are in fact discretized values of real function F(x,y). We can reconstruct this function by fitting some approximation surface to the data surrounding feature point currently in pixel precision. We have chosen Taylor's approximation polynomial in R2 for that purpose. Taylor polynomial of second degree in R1 is given by:

$$T(a) = f(a) + \frac{\frac{\partial f(a)}{\partial x}}{1!}(x-a) + \frac{\frac{\partial^2 f(a)}{\partial x^2}}{2!}(x-a)^2 \tag{15}$$

R2 interpretation is easily derived from (15):

$$T(a,b) = f(a,b) + \frac{\frac{\partial f(a,b)}{\partial x}}{!1}(x-a) + \frac{\frac{\partial f(a,b)}{\partial y}}{!1}(y-b) + \frac{\frac{\partial^2 f(a,b)}{\partial x^2}(x-a)^2 + \frac{\partial^2 f(a,b)}{\partial xy}(x-a)(y-b) + \frac{\partial^2 f(a,b)}{\partial y^2}(y-b)^2}{2!} \tag{16}$$

Sub pixel local extrema is found by differentiating (16) and then setting it to 0:

$$\begin{bmatrix} x_{offset} \\ y_{offset} \end{bmatrix} = -\begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial xy} \\ \frac{\partial^2 f}{\partial xy} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{17}$$

Following pseudo code computes sub pixel's local extrema coordinates offsets from central point position:

```
                dx = (right - left) / 2
                dy = (bottom - top) / 2
            dxx = right + left - 2 * center
            dyy = bottom + top - 2 * center
dxy = (bottom_right - top_right - bottom_left + top_left) / 4
              det= 1 / (dxx*dyy - dxy*dxy)
         x_offset = -1*(dyy*dx - dxy*dy) * det
         y_ offset = -1*(dxx*dy - dxy*dx) * det
```

(2)

Consequent Fig.6 provides even better explanation of how this process works. Pixel data plotted as bar chart in shades of grey. Chrome surface is visualization of Taylor approximation surface. Spheres in the render represents key values in surface approximation. Red sphere - Pixel accuracy maximum, blue sphere -Sub pixel accuracy maximum, green spheres - 8-neighborhood of pixel maximum.

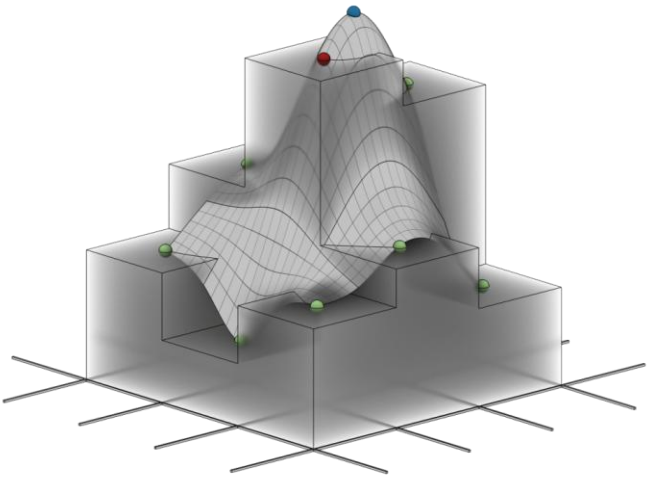

Fig.6: Illustration of sub pixel maximum

2.5 Low contrast feature points rejection

Intensity in DOG image of some feature points will be insufficient to be considered stable therefore we need to discard them. These feature points are in fact located in low contrast areas. We need to interpolate exact value of Taylor polynomial in sub pixel accuracy defined feature points and discard them if:

$$Abs[value] < 0.3 \tag{18}$$

We have to use absolute value because intensity of DOG image is from interval <-1,1>. Interpolating Taylor polynomial is given by (16) where x_offset is used instead of *x* and y is replaced by y_offset. Following pseudo code describes interpolation of Taylor approximation surface in desired point. Variables *dx, dy, dxx, dyy, dxy* are taken from {2}.

```
interpolated_value = center+dx*x_offset+dy*y_offset+0.5*(dxx*
x_offset * x_offset +2*dxy* x_offset * y_offset +dyy* y_offset *
                          y_offset)
```
(3)

2.6 Edges response rejection

Feature points extraction process described in 2.3 has big response in two cases. First case is corner location. Corner in image is area of image where *x* and *y* differences changes drastically under small transitions of feature point. This is in compliance with our desire to produce highly distinctive feature points. Other case is edge response. This area has large intensity variation if feature point is move perpendicularly to the edge and small to none variation if moved along the edge. Feature points produced from edge responses are redundant because they distinctiveness is very poor (huge number of key points are located along the edge but their surrounding is very similar, that makes them hard to be matched correctly). We need to compute principal curvatures of DOG extremas to decide whether located feature points does not lie on the edge. Edge location will have a large principal

curvature across the edge but small one in perpendicular direction. These values are easily computed from the Hessian matrix:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial xy} \\ \frac{\partial^2 f}{\partial xy} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{19}$$

Determinant and trace of the matrix H is used:

$$Tr(H) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \alpha + \beta \tag{20}$$

$$Det(H) = \frac{\partial^2 f}{\partial x^2} * \frac{\partial^2 f}{\partial y^2} - \left(\frac{\partial^2 f}{\partial xy}\right)^2 = \alpha\beta \tag{21}$$

Where α and β are eigenvalues of matrix H. These values are proportional to the curvatures of DOG. Bases on approach proposed by Harris and Stephens (1988), we are only concerned about ratio of α and β rather than about their actual values. That fact speeds the process up remarkably. Let *r* be a ratio between these eigenvalues so that $\alpha = r\beta$. Than the edge response rejection is represented by:

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(r\beta + \beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} \tag{22}$$

Value of (22) depends on ratio of eigenvalues. Therefore we only need to find *r* so that if:

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{23}$$

,feature point is rejected. This parameter *r* was set to 10 in our application based on experiments with different images containing edges. Following figures shows edge and corner DOG function. Their principal values are easily derived from their shape.

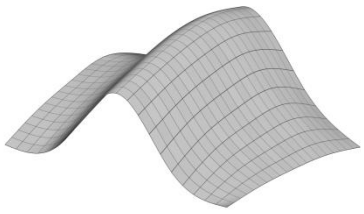

Fig. 7: Example of edge response in DOG

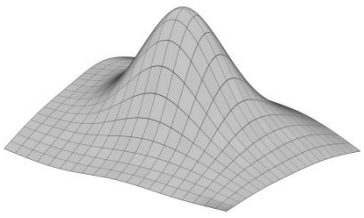

Fig. 8: Example of corner response in DOG

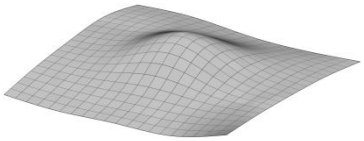

Fig. 9: Example of flat region response in DOG

Fig. 10. shows the response map with notated regions. Dashed lines represents isolines of the feature's response function. Flat region (green) is rejected based on chapter 2.5. Edge regions (grey) are rejected as described in 2.3. Stable feature point have its representation in corner region area (red).

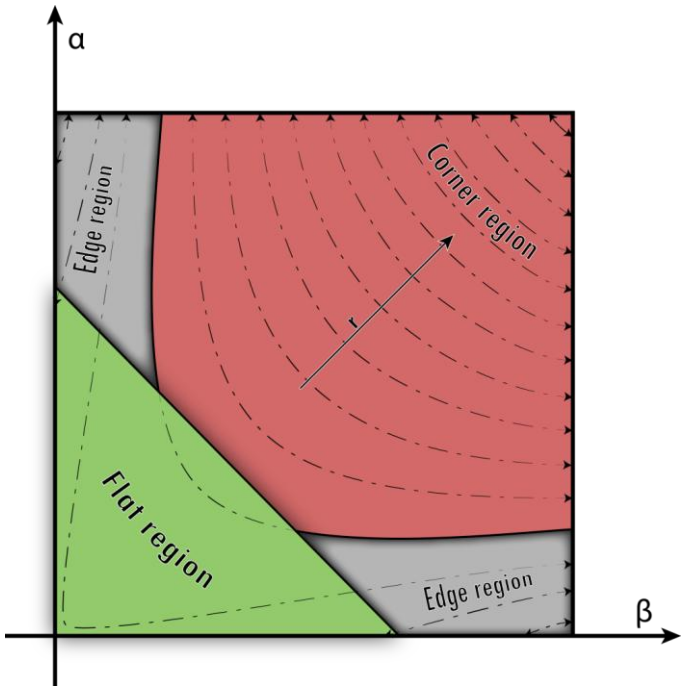

Fig.10: Feature response map with highlighted regions

2.7 Orientation assignment

Assigning stable orientation to the feature point is essential for descriptor to be rotation invariant. Image L(x,y,σ) is chosen from Gaussian pyramid based on feature point's σ. We need to compute magnitude and orientation of data interpolated from this image. These equations are given by:

$$\mathrm{mag}(x,y) = \sqrt{(L(x+1,y,\sigma) - L(x-1,y,\sigma))^2 + (L(x,y+1,\sigma) - L(x,y-1,\sigma))^2} \quad (24)$$

$$\theta(x,y) = \arctan\left(\frac{L(x,y+1,\sigma) - L(x,y-1,\sigma)}{L(x+1,y,\sigma) - L(x-1,y,\sigma)}\right) \quad (25)$$

Square window of width 16 pixels is created around the located feature point and (24) and (25) are computed for all of 256 interpolated values. Small implementation detail has to be figured in this step. Realize that feature point is centroid of this window. This centroid lies in between of the values and because of that actual value representing location of feature point does not have to be interpolated. Therefore we get 256 orientations and appertaining magnitudes. To make orientation as stable as possible we need to weight magnitudes with Gaussian kernel of *σ=1.5*feature_point_sigma*. This operation guarantees that magnitudes closer to the key point are more significant than the ones near the border of the window. 36 bin histogram is created representing 360 degrees of rotation. Therefore one bin represents 10 degrees. Prior computed orientation is used to choose two closest bin. Magnitude belonging to this orientation is then linearly interpolated between these two neighboring bins. Histogram is filled with all 256 magnitudes. Maximum in histogram is located. The actual maximum is than computed by fitting parabola to the surrounding of the bin as proposed by Lowe D. G. (2004). Parabola equation is given as:

$$f(x) = Ax^2 + Bx + C \quad (26)$$

Local extreme is than computed by differentiating and setting equal to zero:

$$0 = 2Ax + B \tag{27}$$

We need to know whether point from (27) is maximum or minimum. We are only interested in maxima which is found if second derivative of (26) is negative in located point. Orientation of this interpolated maximum is assigned to the feature point. For every histogram bin with value larger that 80% of maximum new feature point is created. This feature point's orientation is set by its bin and all the other values are adopted from the original feature point (obviously parabola is fitted again as described in prior section). These additional feature points contribute significantly to the stability of features matching.

2.8 Feature points conclusion

Feature points have been located as extrema of DOG. Taylor expansion up to quadratic terms was used to provide sub pixel accuracy. Than poorly distinctive feature points were removed. Whole process is illustrated in following diagrams:

Fig.11: Feature points extracted from DOG (4509)

Fig.12: Low contrast feature points rejected (1532)

Fig.13: Feature points located on edges rejected (1135)

Fig.14: Feature points with assigned orientation

3. Feature point descriptor

All the feature points are located and orientations are assigned. Last thing to do is to develop way to describe surrounding of the feature point so that it can later be matched against other key points. Descriptor is a vector, mainly from algorithmic side of view. This vector have to be as invariant (all the types of invariance mentioned in the text before) as possible. It is only possible to recognize R2 points representing one R3 point with highest probability if prior property is fulfilled. One way of creating a descriptor is sampling values around the feature point and then using some correlation method do compute probability of feature match. This approach is still used heavily but in our implementation we decided to go other way. As stated in Lowe D. G. (2004):

"A better approach has been demonstrated by Edelman, Intrator, and Poggio (1997). Their proposed representation was based upon a model of biological vision, in particular of complex neurons in primary visual cortex. These complex neurons respond to a gradient at a particular orientation and spatial frequency, but the location of the gradient on the retina is allowed to shift over a small receptive field rather than being precisely localized. Edelman et al. hypothesized that the function of these complex neurons was to allow for matching and recognition of 3D objects from a range of viewpoints. They have performed detailed experiments using 3D computer models of object and animal shapes which show that matching gradients while allowing for shifts in their position results in much better classification under 3D rotation. For example, recognition accuracy for 3D objects rotated in depth by 20 degrees increased from 35% for correlation of gradients to 94% using the complex cell model. "

Following algorithm is based solely on citied text. First, we are going to interpolate values around feature point. This process is similar to the chapter 2.7 where we were computing orientation. The difference is that we need to rotate this 16 x 16 grid in terms of feature point's orientation. Rotation of a point in 2D is given as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{28}$$

Where θ represents feature point's orientation. Keep in mind that feature point is centroid of this grid. Implementation step from 2.7 takes place as well but the location of the grid points are rotated. Values for these grid points are computed and orientations and magnitudes are calculated as well. Magnitudes are than weighted by Gaussian kernel of sigma of half of the descriptor width (8 in our implementation). This grid , or descriptor if you will is then divided into 16 sub regions as illustrated on Fig.15 by black and white hatches. 8 bin histogram is created for every sub region. Magnitudes contributes to the bins the same way as described in 2.7 bearing in mind that one bin no longer represents 10 degrees but 45 instead. In 2.7 we only interpolated orientations in terms of single histogram. Since we want our descriptor to be invariant to small position changes we need to bring this interpolation to the second level. Every magnitude is contributing to the correct bin of all the surrounding sub region histograms. This is ensured by bilinear interpolation. Magnitudes located in the central part of descriptor contributes to 4 histograms while the ones on the border are only divided between 2 "border" sub regions. Values in the corners of the descriptor are weighted by their distance to the closest corner sub region centroid and then assigned to it. Values of histogram bins than becomes descriptor vector. Since we have 16 histograms of 8 bins, computed vector is 128 dimensional. This vector need to be normalized in order to cancel affine changes in illumination. Nonlinear changes have greater effect on matching stability and there is in fact no perfect way to cancel it altogether. Some improvement in that area is achieved by setting values of descriptor that are larger than 0.2 to 0.2 and renormalizing again. Following figure 15 illustrates the descriptor pattern. Red cross represents feature point's location. Green crosses are interpolated values. Blue crosses symbolizes sub regions

centroids as used for bilinear interpolation and hatched black and white areas represents sub regions. Pixels of the studied bitmap are visible in the background with highlighted pixel structure. Arrow shows feature point's orientation.

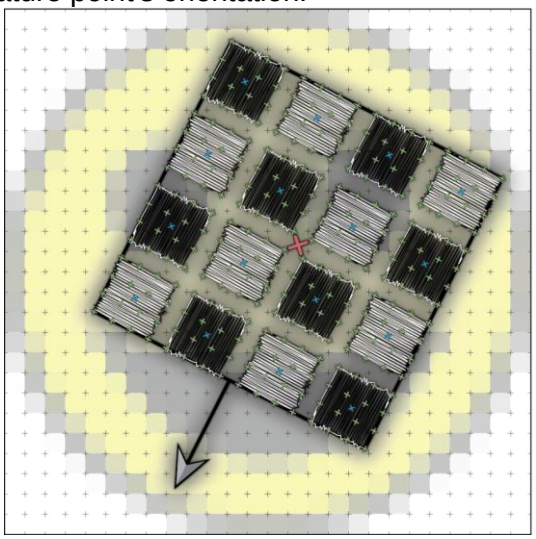

Fig.15: Descriptor illustration.

3.1 Future work with descriptor

After many hours spent on the problem of describing surrounding of the feature point we realized that square descriptor as seen on Fig.15 may not be the best solution. Our yet not implemented proposal for future development of more reliable and faster to compute image descriptor relies in using different pattern for interpolated values. This pattern is shown in Fig.16. It has been computed by complex smoothing function which still need some optimization therefore we decided not to include it in this paper. First idea was to create descriptor pattern from polar grid but that approach was rejected because coordinates got pushed together in the center of descriptor. This is not desired because high density of interpolated values does not provide enough variation for descriptor to be invariant to small changes of position. Therefore, kind of combination between square and spherical descriptor was created. Reason for trying to make descriptor border round is hidden in Gaussian weighting function. Interpolation is rather complex process that requires some computational time (of course computational time is very relative but since for *n* feature points we need to interpolate n*256 values for square descriptor, it is highly advantageous to make as much of it as possible) . Problem with square pattern is that values near to the corners of the descriptor have such small contribution to the descriptor itself because of Gaussian weighting that it is "almost redundant" to compute these values. Odd number of interpolated values is used as well and as a result feature point location and sub regions centorids are sampled as well. For the descriptor in Fig. 16 we used only *209* interpolated values as opposed to 256 in square descriptor created in Lowe D. G. (2004). This improvement was enabled by better distribution of samples in the pattern. Finally our proposal is going to use magnitude-space pattern scaling. This phenomena is represented in Fig. 17. Idea behind it is that feature points's orientation and magnitude could as well be used to calculate eigenvalues of ellipse. We can then use this elipse to form border of descriptor seen in Fig. 16. to create modified pattern in Fig. 17. This pattern might be beneficial in viewpoint changes situations since interesting area might get sampled better. This of course needs a

lot of experimentation. For now it stays in idea domain. We are going to focus on implementing this improvements and comparing it to widely used descriptor in future papers.

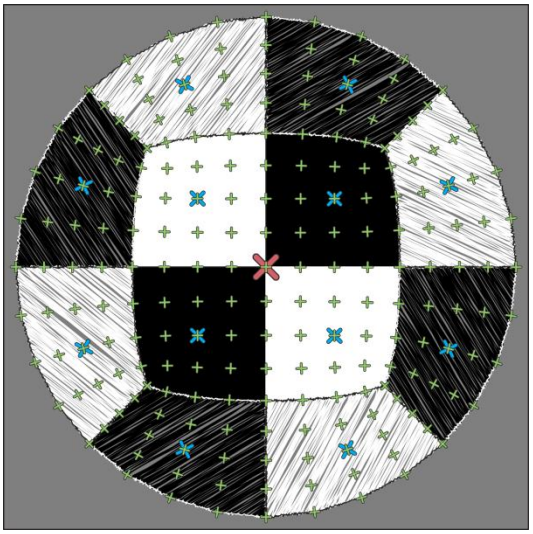

Fig.16: Proposed descriptor pattern.

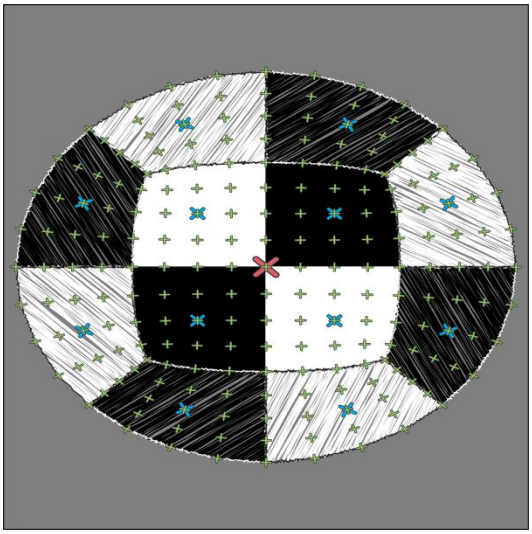

Fig.17: Magnitude affected descriptor pattern.

4. Feature points matching

In chapter 3 we created multidimensional vector that describes feature point based on its close neighborhood. These vector's entries directly represents particular regions of descriptor and since it is rotation invariant the easiest way to correlate two feature vectors is to calculate Euclidean distance. Therefore if we are searching for correspondence of currently checked feature vector we need to calculate this distance to all vectors in database (database might respond to other image or prior computed set get from number of). Closest match is than find as the future vector from database with smallest Euclidean distance to original vector. Match is than noted only If second closest match is outside of 80 percent of closest match. This step take place to cancel indistinctive matches. Unstable matches would be found if this step was left out. Euclidean distance in our implementation is given by.

$$d(checked\_vect, database\_vect) = \sqrt{\sum_{i=0}^{i<128} (checked\_vect_i - database\_vect_i)^2} \tag{29}$$

Computational time of this brute force implementation would be highly affected by database size. For databases larger than 40000 feature vector this computation would take tens of seconds which would be unacceptable. We use approximate nearest neighbor search based on modified *k*-d tree to speed the process up. Feature vectors of database are first sorted in ascending order based on their median values. Modified *k*-d tree is than created with depth.

```
kd_tree_depth=log(database_feature_vector_count)/log(2)
```
(4)

This paper is not focused on *k*-d tree acceleration structures since there are a lot of accessible materials on that subject. Basic idea is described in Fig. 18. Sorted array of medians is divided to halves `kd_tree_depth` times. And median value representing slice is

noted in *k*-d tree structure. This structure is treelike which makes it possible to get approximate nearest neighbor (ANN) in few floating point comparisons (e.g. 40000 feature vectors database only requieres 15 floating point comparisions to get ANN). Since this nearest neighbor is only approximation wee need to check surrounding of the sorted feature vectors database to find exact match. It was proved that as much as 200 points around approximated neighbor is enough to provide over 90 percent chance of finding exact neighbor while improving computational time drasticly. Actual improvement is around 2 orders of magnitude. By comparing all extracted feature points from first image to the ones in second we get matches that can later be used for number of applications. These matches are illustrated in Fig.19. Test images used for this matching were affected by 10 percent Gaussian noise, exposure was decresed by 1.15 and image was rotated by 162 degrees. Non primitive angle (e.g. 180 is primitive angle in this case because no interpolation occurs, therefore image pixels would be the same only their position gets changed) 162 was chosen to introduce even bigger challenge for matching. 318 feature points were matched. As much as 3 are needed to recognize object under affine transformation and at least 8 points are desired to reconstruct camera information. Therefore we consider our implementation to be rather successfull. Ransac could be used in this point to improve results by rejecting false matches.

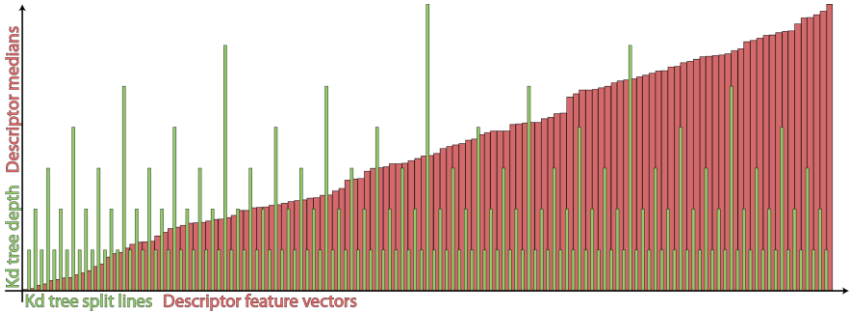

Fig.18: Approximate neighbor search with *k*-d tree split lines included.

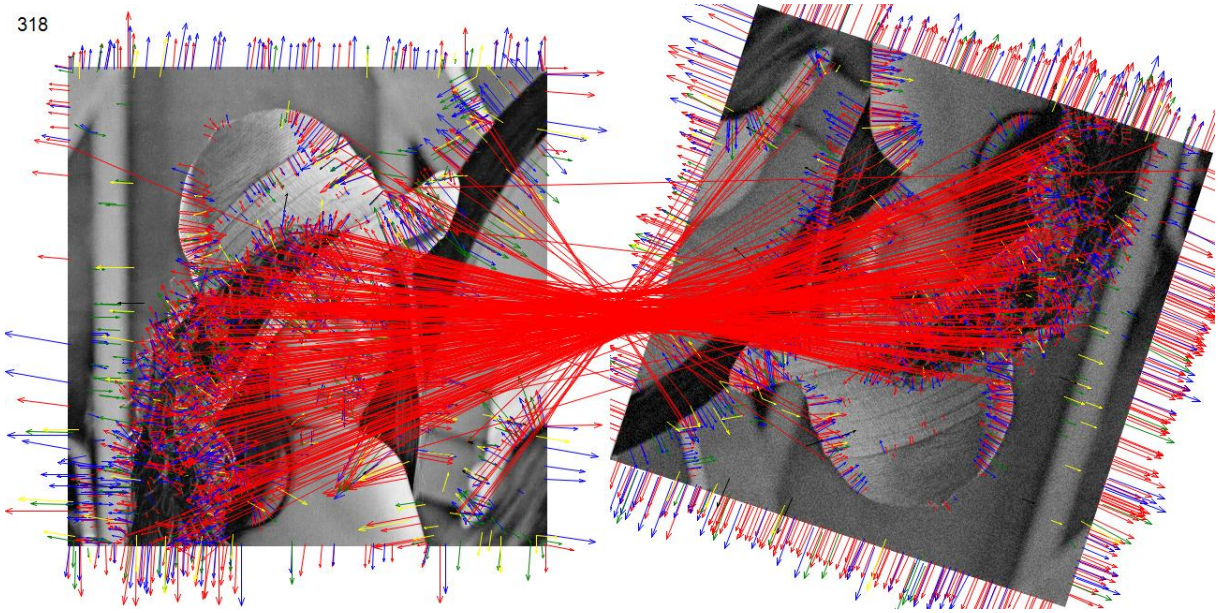

Fig.19: Matched feature points.

Conclusion

This paper provided closer look on implementing sift while partly focusing on optimization of key regions in program. We proposed step by step instructions one might follow to create application of his own. Along with it, equations and pseudo code was provided to ensure even better comprehension. All the illustrations made by us were once again aimed to provide factual and fine-looking illustration of more complex functions and processes. All the figures were created from scratch in order to fully represent our chain of thought during implementation of this algorithm. We would like to build on the foundation created by working with image recognition and exploit its potential to maximum. Our main ambition is to create application for 3D model reconstruction out of 2D projections. This application will rely heavily on terms proposed in chapter 1. Other areas with huge potential for image matching is object recognition, panorama stitching, counting algorithms (traffic cameras for example), face recognition, etc. Computer vision and feature matching in particular provide tools that have revolutionized the world as we know it. It has already found its place in many areas and we believe there is still a lot to come.

References:

Edelman, S., Intrator, N. and Poggio, T. 1997. Complex cells and object recognition. Unpublished manuscript: http://kybele.psych.cornell.edu/~edelman/archive.html

Harris, C. and Stephens, M. 1988. A combined corner and edge detector. In Fourth Alvey VisionConference, Manchester, UK, pp. 147-151.

Lindeberg, T. 1994. Scale-space theory: A basic tool for analysing structures at different scales, Journal of Applied Statistics, 21(2):224-270.

Lowe, D. G. 2004, “Distinctive Image Features from Scale-Invariant Keypoints”, International Journal of Computer Vision, 60, 2, pp. 91-110, 2004.

Mikolajczyk, K. 2002. Detection of local features invariant to affine transformations, Ph.D. thesis,Institut National Polytechnique de Grenoble, France.

Richard Hartley and Andrew Zisserman (2003). Multiple View Geometry in computer vision. Cambridge University Press. ISBN 0-521-54051-8.

YU MENG and Dr. Bernard Tiddeman(supervisor) 2006, "Implementing the Scale Invariant Feature Transform(SIFT) Method ", Department of Computer Science, University of St. Andrews,yumeng@dcs.st-and.ac.uk