Object Detection using SIFT

Yuan Tjiam

August 15, 2022

Abstract

Surgical teams use instrument counts to prevent leaving unintended objects in patients. This is done manually, but could potentially be done through computer vision software. This paper presents a proof of concept for detecting instruments in the operating room with the Scale Invariant Feature Transform (SIFT). The SIFT algorithm is explored and tested on a variety of household appliances to substitute medical instruments. The algorithm responds differently to metal objects compared to matte objects and has room for many improvements. Further research on run time and multi object images is necessary. The proof of concept is considered successful when not taking run time into account.

Keywords— Scale Invariant Feature Transform, Computer vision, Instrument detection, Operating room

Acknowledgements

I want to thank my Professor John van den Dobbelsteen for giving me another chance. And I want to thank my supervisor Rick Butler for our illuminating weekly discussions. Without those I would not have been able to finish. Finally I want to show my appreciation to my parents and my girlfriend for their endless support.

Contents

1 Introduction

The operating room (OR) is by its nature a hazardous place. Many things can and do go wrong, resulting in increased cost and reduced quality of care. One method of preventing mistakes in the OR is to have checks and redundancy checks. These can vary from pre-procedure anaesthetic and allergy checks to instrument counts [9].

Leaving an unintended sponge, needle or instrument in a patient is a mistake that occurs with a frequency ranging from 1 in 5000 to 1 in 7000 [1, 3]. The instrument count is a standard procedure that takes place pre- and post- surgery. It has been standardized by the World Health Organization [9]. Sponge, needle and instrument counts are done at all procedures that risk the possibility of an unintended foreign object being retained in the patient. These objects can lead to inflammation, obstruction, perforation sepsis and death [11]. The risk of foreign object retention increases for emergency operations, a sudden change in surgical procedure and patients with a high body-mass index [4].

The count takes several minutes and is done multiple times, often by multiple members for redundancy. Reconciling miscounts is done via X-rays and results in increased OR time and costs.

If the count could be automated in such a way that it is both fast and reliable, it frees up multiple members of the team and OR time decreases along with cost. Automating the count could be done using Radio Frequency IDentification (RFID) tracking [11], which requires trackers on all objects entering the body. Another method would be to use computer vision software to track objects without adjusting instruments and with minimal change to operating room procedure.

There are different options for computer vision software. In this paper we chose to work with the Scale Invariant Feature Transform (SIFT) by Lowe [7]. Similar software such as Oriented Features from Accelerated Segment Test and Rotated Binary Robust Independent Elementary Features (ORB) is faster but focuses on the center of the image. Software Speeded Up Robust Features (SURF) outperforms SIFT only in noisy images, but is less accurate when it comes to clear images [2]. Another option would be to use deep learning. Deep learning emulates the working of the human brain. It requires a lot of data to train the software. After it has been trained the inner mechanisms are often difficult to understand. In this thesis, we focus on accuracy and reliability over speed and we use sharp and clear images. We also want to be able to understand what effect different variables have on our outcome. For these reasons, we chose to work with SIFT [5, 6, 7, 8].

SIFT compares keypoints in images and matches these to a database. This software is invariant to scale, translation, rotation and illumination. Additionally, it can detect objects rotated in depth up to 20 degrees. These properties make SIFT suitable as instruments are not necessarily placed neatly on a tray under the same lighting conditions pre- and post- operation.

The research question of this thesis is:

”Can the Scale Invariant Feature Transform (SIFT) algorithm reliably detect objects in an operating room setting?”

This thesis will serve as a proof of concept.

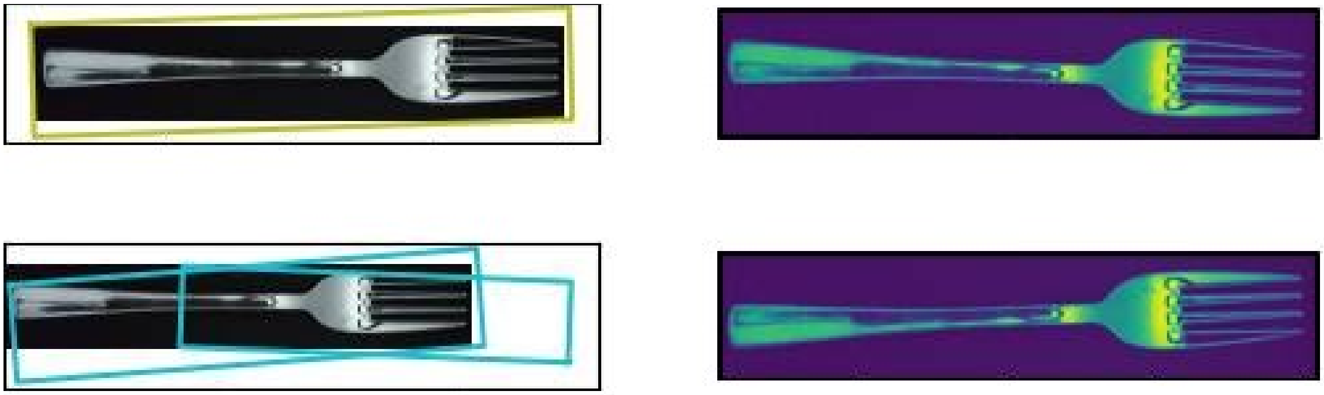

Figure 1: An example of a true positive identification (upper) and a false positive identification(lower). A false negative identification is the same but without any colored boxes.

2 Method

In an operating room setting, a health practitioner would take a picture of all instruments on a surface at various stages of surgery, but at least before and after. These are the scene images. The algorithm would identify the number and type of all objects in a scene image from a previously defined database of instrument images. Non-detections and false positives increase the likelihood of harm to a patient. Which is why we define a correct identification as the detection of the correct type of instrument with no false positives and a 70% overlap of area between the scene and instrument photo. This 70% was chosen because it seemed reasonable.

Positive identifications are shown by colored boxes around the detected instrument in the scene image as in figure 1, upper left. In the case of comparing single instrument scene images, if there are multiple boxes per image, it's determined to be a false positive. All detections are verified by eye.

In an operating room setting, run time would be essential to consider as well. In that setting, the run time of the program should be no more than the current count procedure time. However, there are too many variables influencing run time of the program to consider the run time with regards to the research question.

2.1 Multiple images per instrument

Single image comparison often would not yield positive identifications. If objects are rotated too much along the wrong axis, the algorithm loses the ability to detect it. As instruments are not always neatly placed in real life situations, there was a need for a comprehensive way of detecting instruments regardless of lighting or angle. We decided to make a map of images per instrument that vary in several parameters.

These parameters are instrument rotation, camera angle, lighting angle and lighting intensity. All images per instrument were added to an instrument folder. Then every instrument in a scene has multiple different instrument images to compare to.

The program is also not mirror-invariant. To solve this, images of instruments were mirrored and both versions of the instrument image were used.

2.2 Generating results

The program was tested on household appliances that are similar to surgical tools in shape and size shown in table 1. Preliminary tests of SIFT showed a large difference in outcome between metal and matte objects. This is why we divided the instruments into two categories shown in table 1.

Table 1: Objects that were used to test the program.

Glare	Matte
Fork	Wooden spatula
Butter knife	Black spatula
Spoon	Black soup spatula
Ice cream scoop	Blue spatula
Whisk	

2.3 Camera setup

The objects were photographed in a dark room where light sources were controlled. Each object was placed individually underneath the camera on either a white or black background, depending on the colour of the object. The light never shone directly on the objects, but always on a white wall to imitate omnidirectional homogeneous lighting. True homogeneous lighting would decrease the variability of the results and we expect it would decrease number of false positives, as even slight errors in light direction might lead to different keypoints. The light we used was a Dörr SLR-16 Bi-Color Selfie Ring Light, which allows for multiple angles on lightning and camera angle. The camera is from a Huawei P20 Lite phone, with 16 megapixel camera, f/2.2 lens and a 2 megapixel depth sensor. In figure 2 we see the default configuration of the set up. The camera is parallel with the horizontal of the table. The object is placed directly beneath it and parallel with the alignment of the camera. The light points up toward the white ceiling.

For each object, depending on the size of the object, we varied the distance of the camera to the surface to capture the entire object. We then cycled through all light intensities. We took the lowest intensity that was still clear and sharp because over-lighting the image could result in a surplus of keypoints. From this default position we varied light intensities, the object rotation, the orientation of the light source and the tilt of the camera. For each variation in orientation of light and camera tilt, we rotated the object. Table 2 shows the object rotation, light orientation and camera tilt variations. Per object there are 54 photos, 62 including the light intensity variations.

2.4 Running SIFT

We took the default configuration of the camera and position of the object as the scene image. All other images from an instrument were put into a folder and served as the instrument images. We ran the program for three variables in two settings resulting in eight configurations. Table 3 shows the type of transform, the cluster threshold and the probability threshold. These are found in section 3 The cluster threshold was chosen to be three and four instead of the minimum because

Table 2: Rotation angles of the object, light and camera.

Object rotation	Light orientation	Camera tilt
0°	Up	0°
7.5°	180°	10°
17.5°	90°	27.5°
45°		-12.5°
90°		
135°		
180°		
-7.5°		
-17.5°		

Table 3: Settings for SIFT

Transform	Cluster threshold	Probability threshold
Similarity	3	0.98
Similarity	3	0.99
Similarity	4	0.98
Similarity	4	0.99
Affine	3	0.98
Affine	3	0.99
Affine	4	0.98
Affine	4	0.99

preliminary tests showed an overabundance of false positives when the cluster threshold was two.

All instrument images of every object in table 3 were compared to their scene images. Photos were manually checked whether it was a true positive, a false negative or a false positive. False positives were defined as having multiple identifications within the same image.

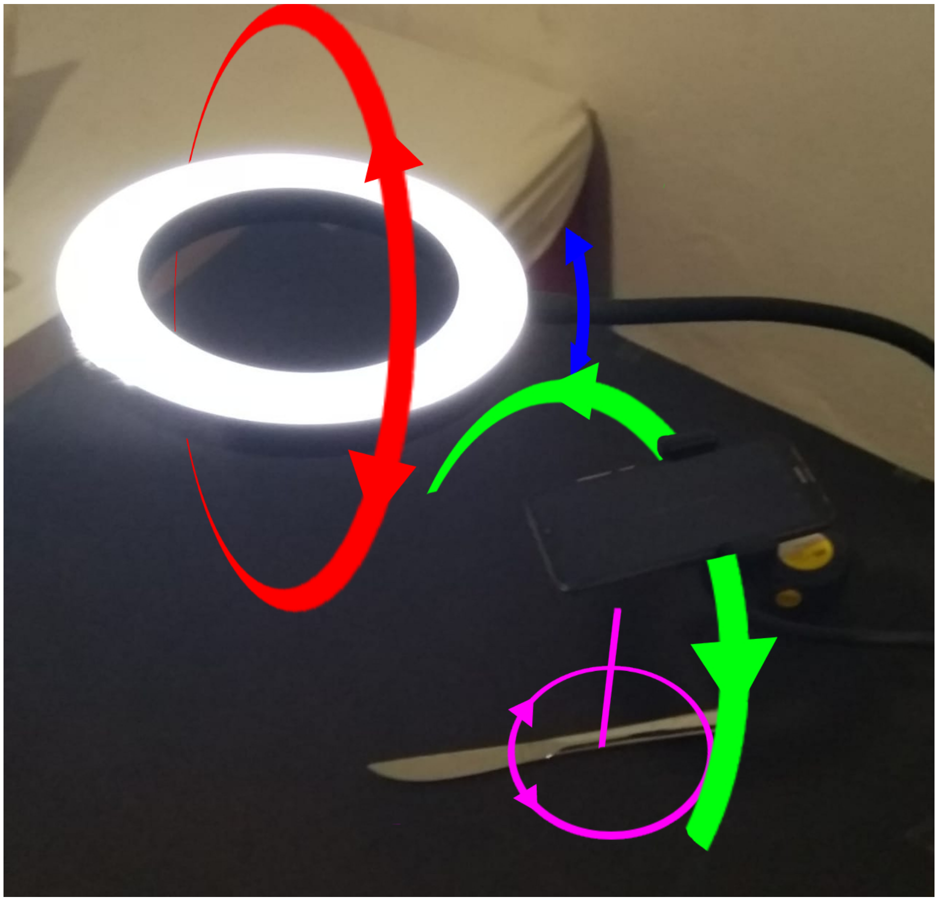

Figure 2: The setup used to take pictures of instruments. The ring light can rotate (red) and tilt (blue). The camera can be tilted so perspective (skewed) photos can be taken (green). Finally the object itself can be rotated (pink). In this local reference system the direction of the knife is 0° for the object rotation and light orientation. A light orientation towards the ceiling is defined as up. The angle for the tilt is defined as 0° directly above the object, with -90° and 90° when the camera hits the table along the green arrows.

3 Theoretical Framework

The algorithm used in this paper is the SIFT algorithm based on Lowe's work [7][6]. Generally, SIFT compares 2D images and detects objects they share. In this application of SIFT, one scene image is compared to many instrument images. An instrument image contains the object that must be detected. The scene image potentially contains it as well. For each unique instrument we use a range of reference images to increase the probability of detection of the instrument in the scene image. The algorithm is coded in Python using integrated development environment Spyder version 5.

Many of the values chosen for the parameters in this section are experimentally determined by Lowe [7, 6]. Many of these could be chosen to have a different value. An oversight of these with more in depth explanations can be found in appendix A. We decided to focus on cluster threshold and probability threshold as the parameters to tune the algorithm with. We chose the cluster threshold because it has a straightforward effect and because the minimum step difference of the cluster threshold is already significant on the performance of the algorithm. We chose the probability threshold because it is one of the final tuning opportunities. It also has a straightforward effect.

SIFT consists of a number of steps, to be repeated for every instrument and scene image. The goal of the algorithm is to create a singular model for each instrument present in the scene that transforms the coordinates of the instrument image into scene image coordinates.

To clearly explain SIFT, example instrument image of figure 3 will be matched onto example scene image of figure 4.

3.1 Overview

This is a very short overview of how the algorithm works. Sections 3.2 through 3.12 explain it more in depth.

First the scene and instrument images are loaded. After loading, every subsequent step builds on the previous one. First keypoints are evaluated for both the instrument and scene image based on their position. Data based on the immediate area around every keypoint is added to these keypoints. Keypoints between the instrument and scene are matched into keypairs. Similar keypairs are clustered. From every cluster, a model is computed. A model maps the instrument onto the scene image. The models are evaluated and filtered. The remaining models are the detections in the scene.

Figure 5 shows the steps in order. The middle blocks are the steps. On the right these are summarized. On the left the information that's passed on from step to step is shown. The purple blocks are the contents of the pink blocks.

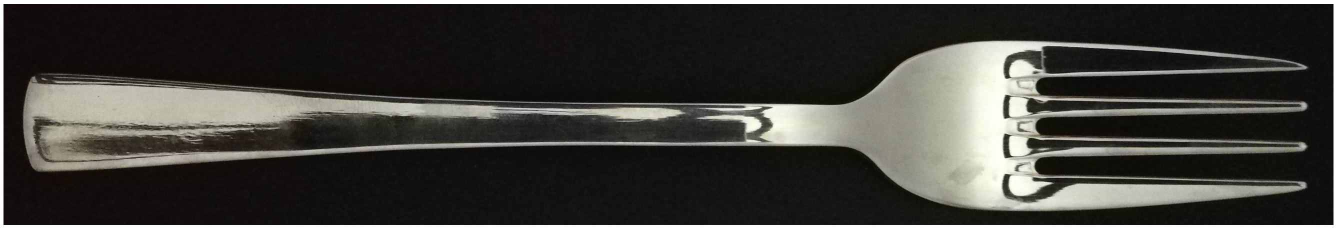

Figure 3: This image of a fork will be the example instrument image to illustrate the steps of SIFT.

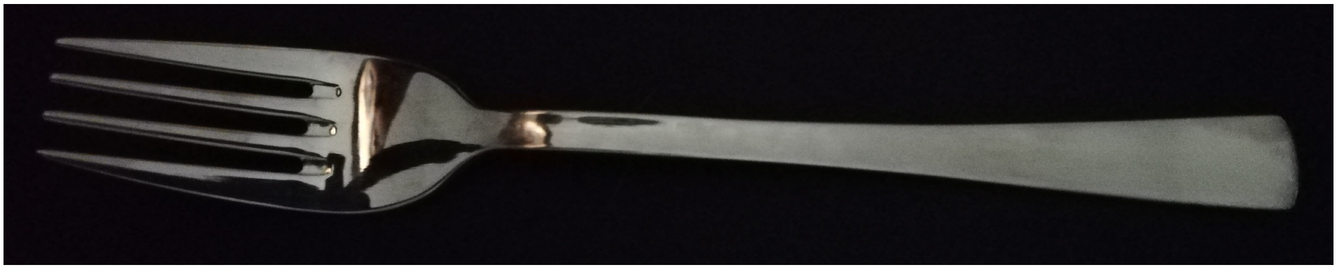

Figure 4: This image of a fork will be the example scene image to illustrate the steps of SIFT

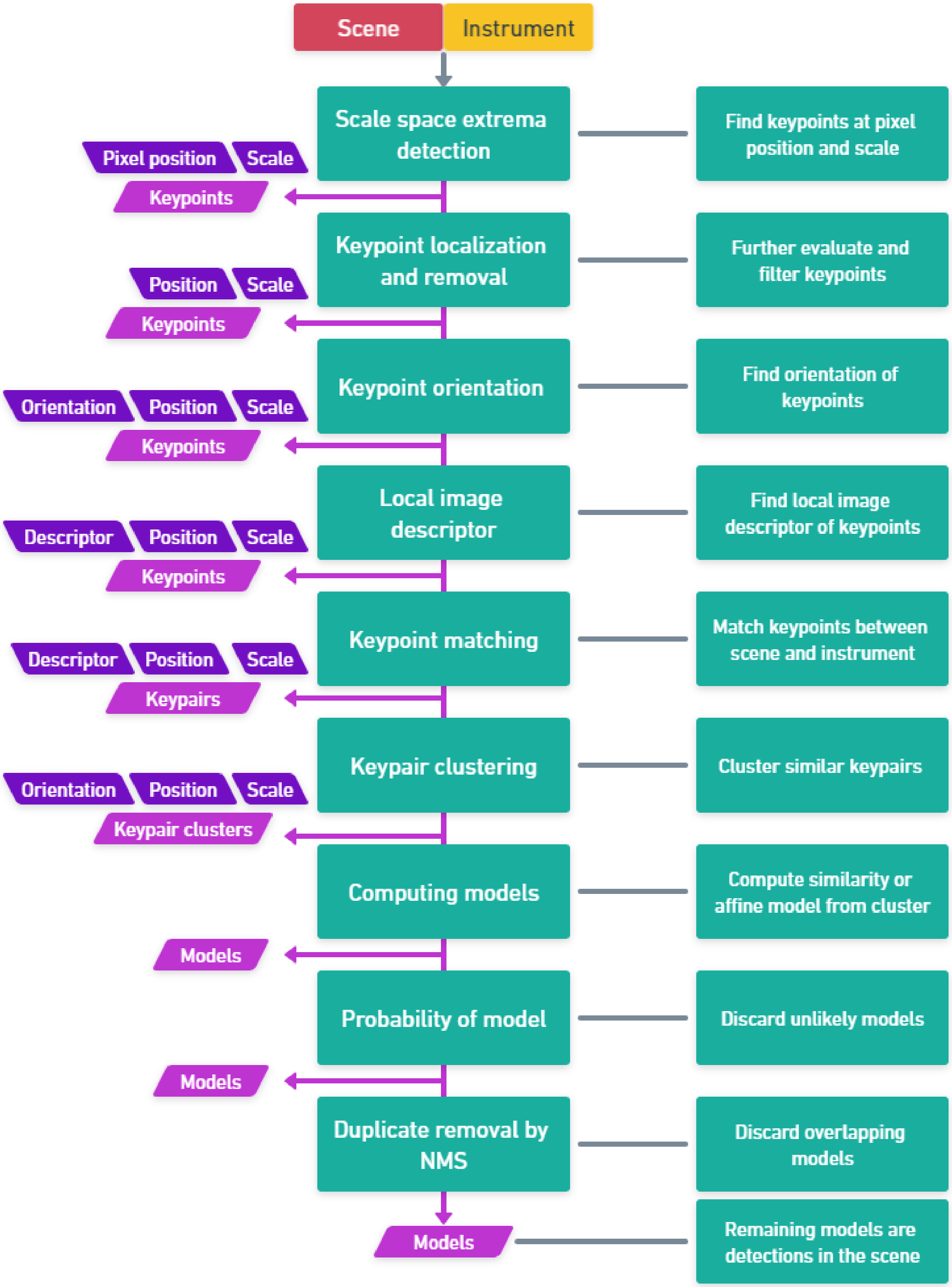

Figure 5: A general overview of SIFT. A scene image and instrument image are loaded into the program at the top. If done correctly the final product is a model that maps the instrument image onto the scene image.

3.2 Scale space extrema detection

SIFT uses extrema detection to define keypoints because this method is rotation invariant and is highly efficient [5]. The image is blurred by convolving it with Gaussians of different standard deviation values σ. The blurred images are called scales. The different σ values are separated by a constant factor k. The Gaussians have standard deviations σ_0, $\sqrt{2}\sigma_0$, $2\sigma_0$, etc.

As this trend continues, more computational time is needed as a higher σ means that a larger area in the image must be evaluated for blurring per individual pixel. However, k is chosen to be $\sqrt{2}$. Because of this, every fourth scale can be replaced by a downsampling of the image by a factor of 2 (i.e. taking every second pixel of an image). As shown on the left side in figure 6, this downsample is original of the second scale. This saves on computational time and has the same accuracy [7]. A set of scales before downsampling is called an octave because it is done with a factor of 2. Lowe recommends 4 octaves [7], which is what we will be using.

In each octave the scales are stacked on top of another. Adjacent scales are then subtracted from each other, with 5 scales giving 4 Difference of Gaussians (DoGs) as shown on the right side of figure 6. This is done for each scale. Of these 4 DoGs, two are the middle layers. Every pixel on these two middle layers except for the edge pixels have 26 neighbors, 9 on the layer above them, 8 on their own layer and 9 on the layer below. Figure 7 shows the pixel marked 'x' as an example. If the value of this pixel is larger or smaller than all its neighbours, it is a local maximum or local minimum and defined as a keypoint.

3.3 Keypoint localization and removal

Now that we have a set of keypoints, we want to make them more descriptive to be able to match them later

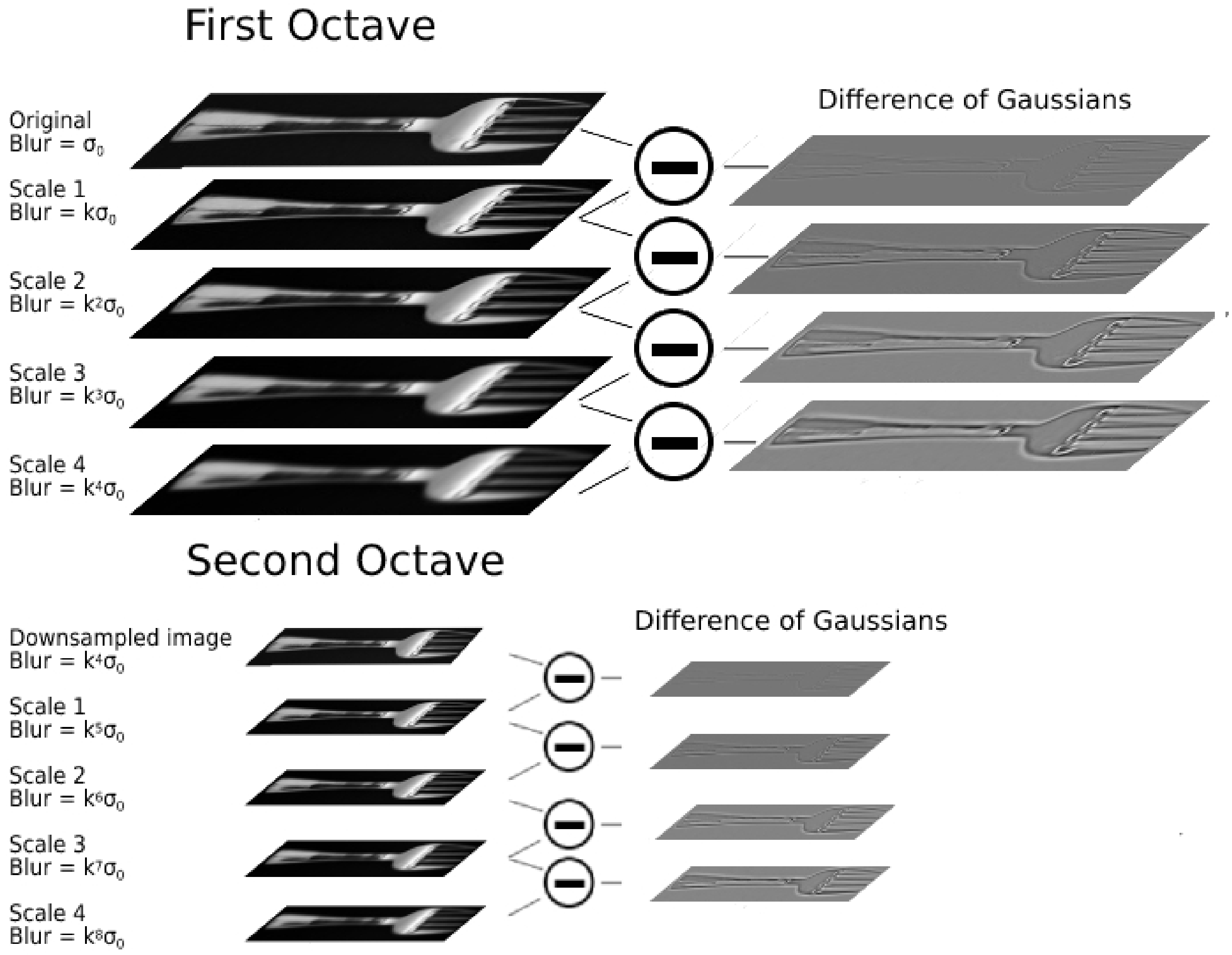

Figure 6: The scales are images convolved with $k\sigma_0$. Every second scale is downsampled to become the first of the next octave.

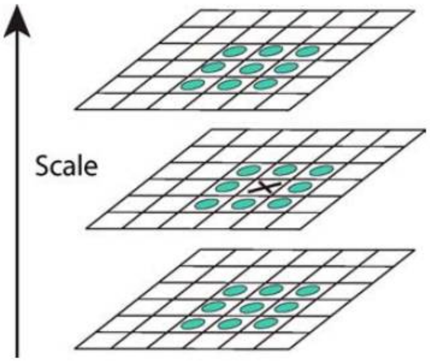

Figure 7: The middle of the middle layer is evaluated. If all 26 surrounding pixels are collectively darker or brighter, the pixel is defined as a keypoint. Image taken from Lowe, 2004 [7].

on whilst removing keypoints that are not robust across images due to, for example, noise. Now, the keypoints possess a discreet location and a scale.

Pixels exist in discrete space. However, the positions of local maxima and minima of the pixels can be evaluated in continuous space. This makes them more descriptive. We can find these positions by evaluating the derivative in every direction, where a higher derivative means that the extrema leans toward that direction. Additionally, it allows for a value at the place of the extrema, giving an opportunity to filter keypoints based on this value compared to a threshold value. Low values are extrema with low contrast which are sensitive to noise [8][7]. This value is discarded for all keypoints as it will no longer be of use.

Keypoints at the edges of the image are common. But as it is the edge of the image, local extrema are much more likely and much less reliable. These are eliminated by evaluating the derivative in the x direction and contrasting it with the y direction. At the edge, one of these is going to be big whilst the other will be small. If the ratio between the two is smaller than the edge ratio (appendix A), the keypoint is discarded.

3.4 Keypoint orientation

So far, each keypoint has a scale, a continuous location and a value. Now we assign an orientation. The closest Gaussian smoothed image to the scale of each keypoint is taken. Selecting for scale in this manner approaches scale-invariance. With this scale, the magnitude m and orientation θ are computed using pixel differences as in formula 1 and 2.

$$m_{x,y}=\sqrt{(L_{x+1,y}\text{-}L_{x\text{-}1,y})^2+(L_{x,y+1}\text{-}L_{x,y\text{-}1})^2} \quad (1)$$

$$\theta(x,y) = \arctan\frac{(L_{x,y+1}-L_{x,y-1})}{(L_{x+1,y}-L_{x-1,y})} \quad (2)$$

These formula are applied to each pixel in the image for this scale. Then, for every keypoint, we create an orientation histogram with 36 bins over 360 degrees. The histogram is filled by taking the orientation and magnitude of the pixels within a 1.5σ area around the keypoint and adding these to the histogram. The farther away a pixel, the less weight it carries in the histogram. The peak position is then approximated more accurately by fitting a parabola on the two histogram values closest to each peak and taking the position of the maximum. If the second highest peak is within 80% of the highest peak, the keypoint is duplicated with this orientation. Figure 8 shows all 3134 keypoints for our instrument image. In figure 9, the largest circle clearly shows two orientations.

3.5 Local Image descriptor

Each keypoint now has an continuous location, a scale and an orientation. Around every keypoint location, for the scale of that keypoint, the orientations and gradient magnitudes of all sample points in that region have been computed in section 3.4. These are rotated relative to the keypoint orientation and weighted according to distance from the keypoint location with a Gaussian weighting function σ_w equal to one half the width of the descriptor window (appendix A). The sample points come from a 16x16 area around the keypoint with respect to the rotation. This 16x16 area is divided into 4x4 regions each with a histogram in 8 directions as shown in figure 10.

To avoid boundary effects, the orientations are trilinearly interpolated across adjacent histogram bins to ensure continuity across bins. The histograms can be

Figure 8: The keypoints of the instrument image 3. There are 3134 keypoints in this image with 3134 corresponding descriptors. Every green circle is a keypoint

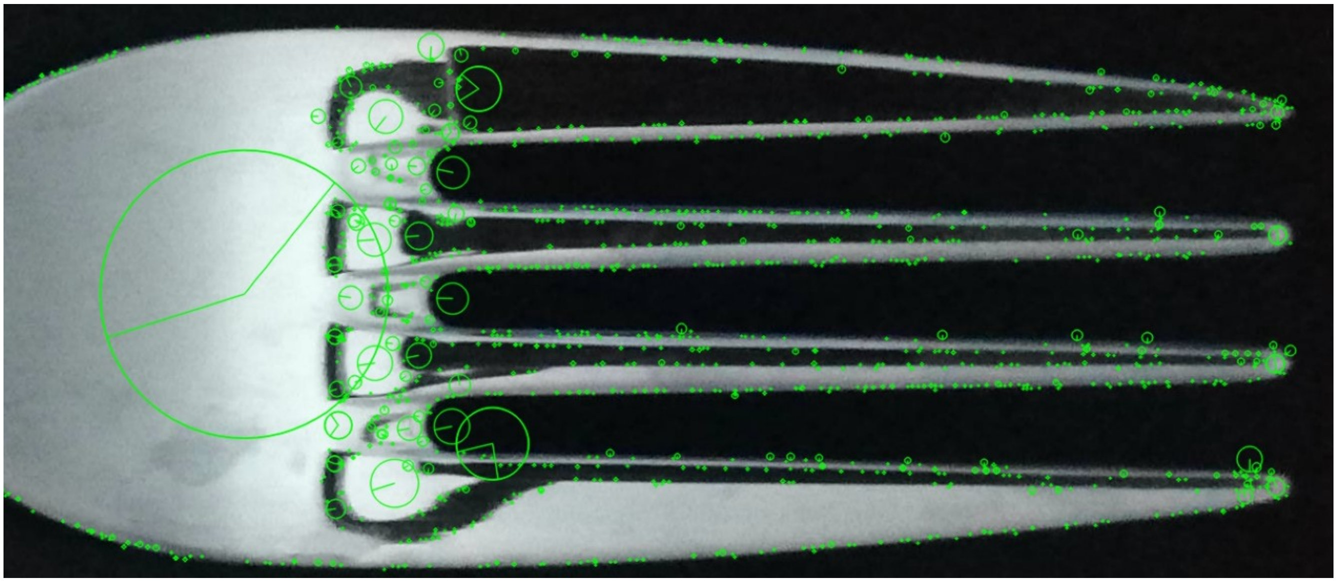

Figure 9: The tip of the fork of image 8 is enlarged to show the keypoints. Every circle is a keypoint where the center indicates position and the radii the orientation. Note that there can be up to 2 radii in a keypoint, depending on whether they satisfy the 80% histogram requirement. Higher octaves and scales (i.e. blurrier images) produce less keypoints.

summarized in a $4x4x8 = 128$ vector for each keypoint. This is the local image descriptor. This vector is then normalized to unit length, which keeps its value if all pixels' illumination increase by the same amount. This makes the local image descriptor illumination invariant, provided the illumination is constant over all pixels in the region around the keypoint. If that is not the case, gradient magnitudes are disproportionally affected with respect to gradient orientations. By capping the gradient magnitudes and renormalizing the vector, emphasis is placed on orientation.

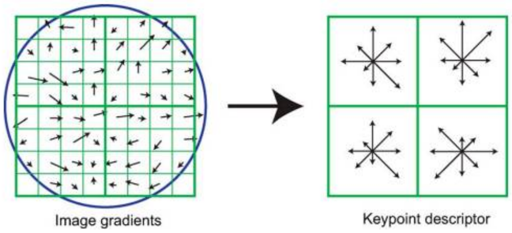

Figure 10: This example image uses a 8x8 set of samples from which a 2x2 descriptor array is computed. Image taken from Lowe, 2004 [7].

See the following example in equation 3, where the vector is 5 dimensional. The vector in this example is normalized, capped at 0.5 and renormalized. The actual value of the cap is experimentally determined to be 0.2 (appendix A[7]).

$$\begin{bmatrix}7\\3\\2\\1\\1\end{bmatrix} \rightarrow \begin{bmatrix}7/8\\3/8\\2/8\\1/8\\1/8\end{bmatrix} \rightarrow \begin{bmatrix}4/8\\3/8\\2/8\\1/8\\1/8\end{bmatrix} \rightarrow \frac{1}{\sqrt{31}} \begin{bmatrix}4/8\\3/8\\2/8\\1/8\\1/8\end{bmatrix} \tag{3}$$

3.6 Keypoint matching

Each keypoint possesses a continuous location, a scale, an orientation and a local image descriptor.

The local image descriptor is used to match keypoints of one picture to another. A match of keypoints is a keypair. The minimum Euclidian distance of the descriptor vector is found by brute-force comparisons. These distances can be globally thresholded, but this method performs poorly because Euclidian distance isn't a great predictor of distinctiveness of the descriptor. For that, relative Euclidian distance between the closest neighbor and second-closest is used. If the closest match is much closer than the second-closest match (appendix A), then the descriptor is distinctive. We use a brute force approach to match the keypoints. Lowe [7] uses the Best-Bin-First algorithm, which is faster. We use the brute force method because it is more accurate. Figure 11 shows keypoint matching between images without and with the Euclidian distance requirement.

3.7 Keypair clustering

We are now working with keypairs. Every keypair has a pair of continuous position, a pair of scales a pair of orientations and a pair of local image descriptors. The local image descriptor is no longer needed and is discarded.

Keypairs indicate where the instrument should be placed in the scene. It can be likened to using thumbtacks to overlay images on a scrap board. The more thumbtacks are present, the surer one can be that the image is correctly placed. However, this does not mean

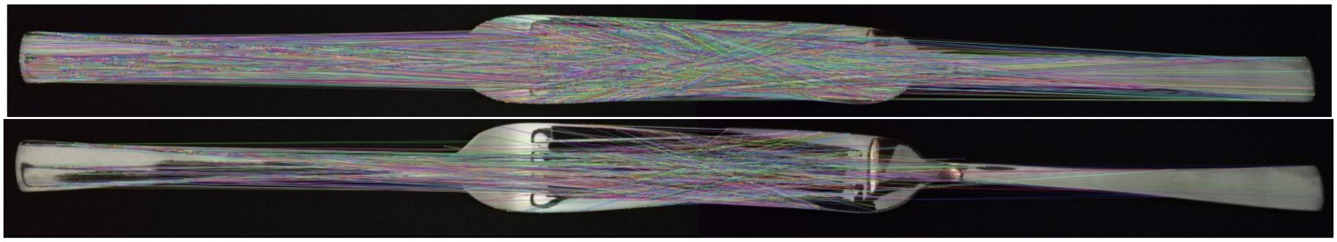

Figure 11: The keypoint matching algorithm applied to the instrument (upper left) 3 and scene (upper]right) 4 image. Every line matching these is a keypair. The bottom is the same but with the Euclidian distance requirement

all keypairs can be used in this manner because not all keypairs indicate true positive instrument detections within the scene. This is why the keypairs are grouped together based on how much they are alike.

A keypair has the following information in both the instrument and scene image: the position, orientation and the scale. Grouping keypairs together must be done based on invariant parameters. In this case the difference between the positions, orientations and scales. This gives a translation, rotation and scaling. These are computed in following section 3.8. They can then be grouped according to these differences.

3.8 Keypair parameters

A keypair has the x position, the y position, the orientation θ and the the scale s for the instrument and the scene image. From these the following parameters can be computed: the difference in x coordinates d_x, the difference in y coordinates d_y, the scale ratio d_{Scale} and the difference in orientation d_θ. The parameters d_{Scale} and d_θ are straightforwardly computed in equations 4 and 5.

$$d_{Scale} = \frac{scale_{Scene}}{scale_{Instr}} \tag{4}$$

$$d_{Angle} = \theta_{Scene} - \theta_{Instr} \tag{5}$$

Parameters d_x and d_y are more complicated. The coordinates from the instrument image must be mapped to the scene image first. Instrument image coordinates must be adjusted for scale and rotation. The order is relevant, but all orders are possible. In this paper we do scaling first, rotation second. The scaling is relative to the center of the instrument image. See equations 6 and 7.

$$x_{scaledInstr} = (x_{Instr} - \frac{xlength_{Instr}}{2})d_{Scale} \tag{6}$$

$$y_{scaledInstr} = (y_{Instr} - \frac{ylength_{Instr}}{2})d_{Scale} \tag{7}$$

After scaling, the scaled instrument positions are rotated based on the difference in orientation d_θ to find the x and y positions in the scene $x_{InstrScene}$ and $y_{InstrScene}$. This is done with a rotation matrix shown in equation 8.

$$\begin{bmatrix} x_{InstrScene} \\ y_{InstrScene} \end{bmatrix} = \begin{bmatrix} \cos d_\theta & \text{-}\sin d_\theta \\ \cos d_\theta & \sin d_\theta \end{bmatrix} \begin{bmatrix} x_{scaledInstr} \\ y_{scaledInstr} \end{bmatrix} \tag{8}$$

Now the position of the instrument keypoint can be located in the scene. Finding parameters d_x and d_y is a simple subtraction. See equations 9 and 10.

$$d_x = x_{Scene} - x_{InstrScene} \tag{9}$$

$$d_y = y_{Scene} - y_{InstrScene} \tag{10}$$

3.9 Hough Transform

Each keypair now has the four known parameters d_x, d_y, d_{Scale} and d_θ. These describe how a keypoint in the instrument image is placed in the scene image. From these keypairs a model can be computed that maps all the pixels from the instrument image onto the scene image.

One keypair is sufficient to transform the instrument into the scene with the proper rotation. However, the second keypair determines the size of the image. The model that can be computed from two keypairs is called the similarity transform. The instrument or the scene image can also be taken at an angle, making it necessary to skew the instrument image. For this a third keypair is needed. The model computed from three keypairs is called the affine transform. Not just any three keypairs can be considered to find an instrument in the scene. The keypairs must be sufficiently similar. We use the Hough transform to cluster the keypairs.

The Hough transform considers the keypoints in parameter space, creating a 4 dimensional box known as a bin where the 4 parameters are enclosed by the 4 dimensions. Figure 12 shows the principle in 3 dimensions.

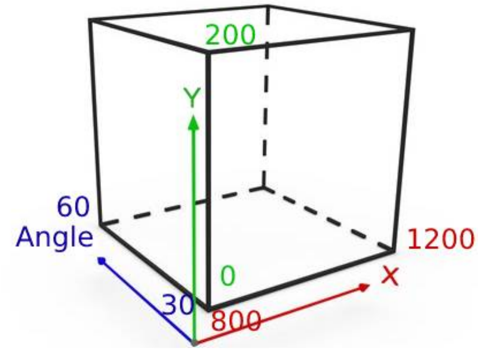

Figure 12: This example box is illustrated in 3 dimensions. Every keypair with an angle difference between 30-60 degrees, a y difference between 0-200 pixels and an x difference between 800-1200 pixels would fall in this box. For illustrative purposes the boundary problems are not shown, but can be viewed as one point falling into two adjacent boxes.

Each parameter is one dimension of the bin. The bin sizes per dimension are taken from Lowe [7]. They are 30 degrees for orientation difference, factor 2 for scale ratio and the maximum projected training image dimension for d_x and d_y is 0.25 (appendix A). As the entire parameter space must be accounted for, larger bins mean less bins and vice versa.

The voting is done by assigning a unique number to each box. Table 4 is an example of a keypair with four parameter values. For angles, 30 degrees lead to 12 bins, values 0-11 in python. The scale ratio of 2 include a number of scales, and can be assigned a number by taking the $\log_2$(scale ratio). For d_x and d_y, the ratio between d_x and the size of the scene image in the x dimension times the number of bins assigns a number. Rounding up and down yields whole numbers, two for every dimension for each keypair.

Table 4: Table of example parameter values for difference in orientation, scale ratio, difference in x and difference in y. Parameters d_x and d_y are dependent on image size, so normalized values are arbitrary. These are normalized and given a lowerbound and upperbound number.

	d_θ	d_{Scale}	d_x	d_y
Parameter value	45	8.0	489	9312
Normalized value	1.5	3	8.6	4.5
Lowerbound number	1	3	8	4
Upperbound number	2	4	9	5

In this example, there are four sets of two values. These are added together like letters, to assign a number to a bin. The bin numbers for this example would be 1384, 1385, 1394, 1395, 1484, 1485, 1494, 1495, 2384, 2385, 2394, 2395, 2484, 2485, 2494 and 2495. Each of these bins gets one vote from this keypair. Other keypairs might vote for different bins, which will be generated. Multiple votes end up in the same bin only if their parameter values fall in the same bin. A bin needs at least two votes to compute a similarity model and three votes for an affine model. These are explained in section **??**. Every filled bin is called a cluster.

Boundary problems are prevented by adding the keypair match to the two closest bins for each parameter. For four parameters, this leads to 16 boxes with a vote for each keypair. This also increases total number of votes for bins by a factor 16, increasing the chance that a bin receives multiple votes.

3.10 Computing models

Both the similarity and affine transform are models that transform instrument image coordinates into scene image coordinates. They both have the same basic formula 11[7].

$$\begin{bmatrix} x_{Scene} \\ y_{Scene} \end{bmatrix} = \begin{bmatrix} m_1 \; m_2 \\ m_3 \; m_4 \end{bmatrix} \begin{bmatrix} x_{Instr} \\ y_{Instr} \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{11}$$

The similarity transform model takes into account scale, rotation and translation. The affine model takes into account scale, rotation, translation and skew. These are hidden in the m parameters from formula 11.

The m parameters from the similarity transform are found by multiplying the rotation with the scale. The scale is a scalar s. The rotation is defined by a standard rotation matrix R of rotation θ. The m parameter matrix for the similarity transform is found in equation 12.

$$m = sR = s \begin{bmatrix} cos(\theta) & \text{-}sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{12}$$

The affine transform adds another element to the m parameters: the shear matrix C 13.

$$m = s\ R\ C = s \begin{bmatrix} cos(\theta) & \text{-}sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & c_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ c_y & 1 \end{bmatrix} \tag{13}$$

The m parameters can be found by taking formula 11 and using a least squares method. The similarity transform angle θ and scale s can then be calculated from the m parameters in a straightforward manner as in equations 14 and 15.

$$\theta = \arctan \tfrac{m_2}{m_1} = \arctan \tfrac{-s \sin \theta}{s \cos \theta} \tag{14}$$

$$s = \sqrt{m_1^2 + m_2^2} = \sqrt{(s \cos \theta)^2 + (-s \sin \theta)^2} \tag{15}$$

The affine m parameters are more difficult to decompose into angle, scale and shear. We used Maple for the calculations. These can be found in appendix B.

The translation, scale and rotation of the model is then checked against the original keypairs. The model must be closer to the keypairs than half the original bin sizes of the Hough transform (appendix A). If this is not the case, the offending keypairs are removed from the cluster and the model is recomputed without those keypairs until the model is accepted. If the number of keypairs is insufficient, the entire cluster is discarded.

3.11 Probability of model

The candidate models can be rank ordered and thresholded by assigning a probability [6]. This is the probability that model m exists given the keypair set f. Which is the probability that the keypairs are present if the model is present divided by the probability that the keypairs are present in general. This probability can be approximated using equation 16 (appendix C).

$$P(m|f) \approx \frac{P(m)}{P(m)+P(f|-m)} \tag{16}$$

If this probability is larger than a certain threshold, the model is accepted. Lowe recommends a probability threshold of 0.98 (appendix A).

3.12 Duplicate removal by NMS

The method of applying the SIFT algorithm as described in the previous section 2.1 leads to more positive identifications. However, as each instrument photo is treated by the algorithm as a new possible instrument, the same instrument can be found in the same position multiple times as shown in figure 13 (left).

These duplicates in combination with the possibility of multiple detections per instrument image distorted the results of the algorithm. Every duplicate is a false positive and throws the count off. To remove the duplicates we used the Non-Maximum Suppression algorithm (NMS).

The NMS algorithm was used to identify duplicates and remove them [12]. It functions by ranking the proposed models by probability, taking the model with the highest probability of being a true positive. This model was then compared with all other models by calculating the intersection of the bounding boxes and dividing by the union of the bounding boxes. This is the Intersection over Union (IoU). If this is larger than 0.9 (appendix A, that model is removed. This threshold is chosen to be high to ensure that only very similar models are removed. If the model is not removed, it is retained as a separate detection and matched to all remaining models.

The remaining models are the detected instruments. Figure 13 (right) shows a successful removal of duplicates leaving only one positive identification.

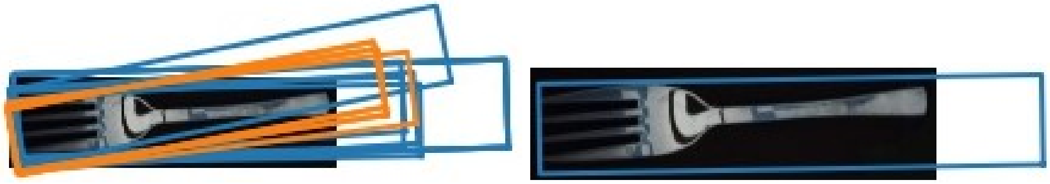

Figure 13: There are 14 detections before the NMS algorithm (left). Afterwards, only 1 detection remains (right).

4 Results

The results displaying the differences between glare and matte objects and the differences between the affine and similarity transform can be found in table 5. Table 6 shows the effect of increasing the cluster threshold for both glare and matte objects. The difference between regular and mirrored images for glare, matte and their average is found in table 7. Finally, table 8 shows the difference between the similarity and affine transform for both regular images and images taken by a tilted camera along the green axis of figure 2.

Additionally, as an example we applied the program on a scene with multiple instruments with an instrument map of 28 instrument images. This took 40 seconds. Figure 14 shows what a real life application of the program could look like. Of the 4 objects, only two were detected. These were detected with no false positives.

Tables 5, 6, 7 and 8 show a significant difference between the performance of the glare and matte objects in terms of both precision and recall. Tables 5 and 8 show that affine transform performs worse than the similarity transform in terms of both precision and recall.

Figure 14: An example of a scene image with multiple instruments detected. There are 2 false negatives and 0 false positives. The instrument map here contains 28 images, 7 per instrument.

Table 5: Comprehensive results displaying the true positive, false positive, false negative, precision and recall of glare and matte objects for similarity (Sim) and affine (Aff) transform and their average (Avg).

	Glare Sim	Glare Aff	Mat Sim	Matte Aff	Glare Avg	Mat Avg
True positive	20%	14%	52%	39%	17%	46%
False positive	73%	67%	26%	26%	70%	26%
False negative	7%	20%	22%	34%	13%	28%
Precision	0.21	0.17	0.67	0.60	0.19	0.64
Recall	0.74	0.41	0.70	0.53	0.55	0.62

Table 6: Difference between glare and matte for cluster thresholds 3 and 4.

	Glare Cluster 3	Glare Cluster 4	Matte Cluster 3	Matte Cluster 4
Precision	0.14	0.23	0.61	0.67
Recall	0.51	0.56	0.67	0.59

Table 7: Difference of precision and recall between regular (Reg) and mirrored (Mir) images for the glare, matte and their average.

	Glare Reg	Glare Mir	Matte Reg	Matte Mir	Avg Reg	Avg Mir
Precision	0.24	0.18	0.81	0.74	0.57	0.49
Recall	0.66	0.50	0.57	0.47	0.59	0.47

Table 8: Difference between tilt and regular for affine and similarity, for glare and matte objects.

Tilt	Sim Glare	Aff Glare	Sim Mat	Aff Mat	Avg Sim	Avg Aff
Precision	0.13	0.16	0.67	0.58	0.34	0.32
Recall	0.87	0.54	0.79	0.61	0.81	0.59
Regular	**Sim Glare**	**Aff Glare**	**Sim Mat**	**Aff Mat**	**Avg Sim**	**Avg Aff**
Precision	0.20	0.16	0.71	0.70	0.41	0.39
Recall	0.49	0.25	0.72	0.60	0.59	0.40

5 Discussion

5.1 Matte and metal

In columns 5 and 6 of table 5 the difference between glare and matte objects is clear. Glare objects have a much higher false positive percentage and a much lower true positive percentage than matte objects. Tables 7 and 8 show the difference between in terms of glare and precision.

Table 6 concretely shows a lower precision and recall for glare objects across cluster thresholds 3 and 4. Most notably the precision for glare objects is much lower. This is because the number of false positives is higher in glare objects.

This difference is probably due to the unpredictability of light reflection in metals. In the first step of SIFT, scale space extrema detection, keypoints are generated by local minima and maxima. Metal surfaces have specular reflection, while matte surfaces have diffuse reflection. A specular reflection can have many extrema and therefor many keypoints, depending on what it reflects. Additionally, the variability of these keypoints between images can differ vastly due to the specular reflection of metal surfaces.

On the other hand, diffuse reflections give a surface a constant color. This uniform reflection has fewer local extrema, and therefor fewer keypoints than a metal surface. For matte objects, this means non-detections are more prevalent.

Variables such as image quality, lighting, background and object size and complexity being equal, metal objects generate more keypoints and these have a higher variability.

The SIFT program should then account for what type of instrument it is detecting, and adjust parameters accordingly. For glare objects, the number of accepted models should be reduced by increasing parameter thresholds.

5.2 Similarity and affine transform

Columns 1 through 4 of table 5 show the difference between the similarity and affine transform. The similarity transform performs better in both true positives and false negatives. False positives remain about equal, with some variance below 10%. This indicates that the similarity transform performs better than the affine transform across the board. The nature of the affine transform isn't congruent with these results: it should have fewer false negatives at least because it should be able to fit models onto the scene more easily.

This not being the case indicates that there is some problem with the affine transform. The affine transform uses a complicated equation (appendix B) to decompose the m parameters into scale, skew and rotation. However, for some affine model computations, these formula attempt to take the square root of a negative number. These potential models were skipped as candidate models, which makes it likely that an indeterminate number of true and false positives were lost. Possible solutions can be to find a different mathematical way of decomposing the m parameters or to use the similarity transform for these instances.

However, the affine transform is comparatively better when looking at skewed images. Comparing the similarity transform with the affine transform from table 8, the average precision for glare and matte stays quite even but recall drops significantly. For regular photos the recall drops 8% more than for tilted photos. When comparing skewed and regular images, the skewed images have an increase in false negatives for both the affine and similarity transform. However, the increase of the affine transform is smaller than the increase of the similarity transform. This indicates that the affine transform performs comparatively better than the similarity transform for skewed images.

So there is evidence that the affine transform can be more effective at comparing depth rotated objects. This is only relevant if the affine transform can be made to work properly, as it now performs worse in both precision and recall than the similarity transform.

5.3 Mirrored images

Table 7 shows that the regular images perform better. This is expected because SIFT is not a mirror-invariant program. This means that copying and mirroring images contributes to a higher positive detection rate, as objects that would have been mirrored physically can now be successfully detected.

The average difference found in columns 5 and 6 show a difference of 14% in precision and 20% in recall. However, copying and mirroring every instrument image also nearly doubles computing time.

5.4 Hyper parameters

The two hyper parameters that we varied were the cluster threshold and the probability threshold.

The first two columns of table 6 show that the precision and recall of glare objects increases with increased cluster threshold. The stricter threshold predictably reduces false positives and increases false negatives. This is paired with an increase in true positives, as it reduces false positives more than it reduces false negatives. For the final two columns, matte objects show the same effect on false positives and negatives, though false negatives increase more than false positives decrease, which means there are less true positives. This indicates, all other things being equal, that the cluster threshold of 4 is more suited to glare objects while a threshold of 3 to matte objects.

For matte and glare objects, the best cluster thresholds seem to be 3 and 4 respectively. A cluster threshold of 3 means that there are 3 keypairs required to compute a model. Going from 3 to 4 means that every model that would have been accepted with 3 keypairs is now discarded. However, these are often models that are similar to the best model.

Changing the cluster threshold is more impactful than it appears when solely looking at precision and recall. When increasing the threshold, the number of accepted models decreases. However, many of these models are similar and would have been removed anyway due to the NMS algorithm. This means that the effect of the cluster threshold change on precision and recall is not as noticeable. However, it means that any effect on precision and recall is based solely on the models the NMS algorithm could did not catch.

The cluster threshold of 2 in this configuration of SIFT generated too many false positives and a cluster threshold of 5 generated many false negatives. However, depending variables such as the size of the instrument map, higher cluster thresholds could be useful.

Varying the probability threshold from 0.98 to 0.99 proved too small of a step. We still believe that the probability threshold is a logical choice to tune the algorithm. It is near the end of the algorithm, which makes it a very intuitive variable to tune. Model probabilities often reached 0.999999, so further testing should be done keeping a such a threshold in mind compared to the 0.98 recommended by Lowe.

In the SIFT algorithm, there are many other variables that can be used to tune the program. An overview of these and what would happen to the program if you changed them can be found in appendix A. Many of these were experimentally determined by Lowe [7]. These can be tuned as well but we believe that the cluster and probability threshold parameters should be researched more thoroughly. The variables should be considered first are the cluster threshold, the probability threshold and the Hough transform bin size. These are straightforward in their effect: reduce the number of positive detections by increasing thresholds or reducing the bin size. A more fundamental approach would be to reduce the number of keypoints detected in any given image. However, it is hard to predict what discarding information at such an early stage would do.

5.5 Precision and Recall

The results are largely measured by precision and recall. These are measured over categories. The precision shows the ratio of true positives and total positives, that is what fraction of all detections is true. Recall tells us the ratio between the true positives and the sum of the true positives and false negatives, that is what fraction of all relevant detections is detected. In this subsection we explain how results should be interpreted when taking into account recall and precision as separate.

We want to test he program in such a way that that quantitative data is produced that sheds light on the functioning of the program. This was done through the matching of individual photos, which isn't the way the program would function in an OR setting. The program takes an entire map of photos of an instrument and tries to locate that instrument in the photo, for each photo.

Here we have tested individual images. We do this because testing the entire map on a picture can give qualitative results, but would generate little data that indicate how the program would react to different configurations. This means that the results must be interpreted whilst taking the individual nature of the test into consideration.

For the precision this has little bearing; the higher the precision, the better the program functions. However, recall is another matter. A large number of false negatives will decrease recall, but a that isn't necessarily a bad thing. Each false negative is just one undetected instrument from an instrument map with potentially hundreds of photo's, which means false negatives aren't a problem so long as one of the other images in the map finds a true positive, thereby detecting the instrument.

F score is meant to be a measure of accuracy and often goes hand in hand with precision and recall [10]. In this thesis we disregard F score as its relationship with accuracy is skewed due to the individual nature of the tests contrasted with the wholesale nature of the intended program.

A high recall means only that there are few false negatives. It must be evaluated in concert with the precision. Table 9 shows how the configuration of the program should be altered for different situations. If there's a high precision and a high recall, then the program is working as intended. If there is high precision and low recall, the program might be too severe in its boundaries. It could also be just fine, depending on whether the map of instruments can generate at least one true positive. For low precision and high recall, the program needs to be stricter to increase the precision, even though recall might fall. And for low precision and low recall, the program needs to be stricter to increase precision and perhaps more photos are to be added to the map to increase the chance of finding a true positive.

Table 9 shows how the algorithm should be configured when

Table 9: Precision and recall for a map of images.

	Low precision	High precision
Low recall	↑ Threshold & images	↓ Threshold
High recall	↑ Threshold	Good

Using an entire map of photos allows us to prioritize precision over recall, which means making the program stricter is preferable to reduce false positives, as false negatives can be compensated by adding more photos of instruments to maps. This will however increase computational time.

5.6 Requirements and limitations

The main limitations are practical. The program cannot detect what it cannot see. Therefor, the instruments have to be placed on a table in such a way that every object is at least partly visible. Also, the program would work best when the objects are placed on a contrasting background. These requirements would probably require a human touch or at least a deviation on current procedure.

Pictures have to be taken. This can be done automatically with some sort of camera set up or by a medical practitioner. Additionally, a good user interface is essential. If the medical team can easily input what instrument the program missed and draw a box around the instrument, the database can be filled quickly. On the other hand, if some instrument is supposed to be missing, that too should be easily entered in the user interface.

5.7 Future research

This algorithm needs more testing. Individual image tests were promising, but more cluttered images need to be tested. The NMS algorithm has not been tested in a setting where multiple objects are stacked on top of each other. The threshold for removing duplicates should be strict enough to ensure that doesn't happen.

Lowe [6] also integrates multiple views of an object. He takes an initial view of the object and uses the Hough transform [6] to add subsequent views from different angles. This creates a larger set of keypoints with which to match an image. This increases the chance of matching the model to an image. This method uses the similarity transform instead of the affine because the latter "provides a poor approximation for rotation in depth of more complex 3D objects" [6]. We believe this approach is suited to this application. Instead of comparing multiple models per training image for multiple images in our image map, the relevant keypoints can be combined into one model. Additional training image keypoints can then be compared to that current model. If it matches, those keypoints can be added to the model. If it doesn't match, a new model can be made. This increases robustness of the program [6]. More specific to our implementation, not just 3D depth rotation could be varied. If the lighting changes per training image, unique keypoints that are generated by arbitrary specular reflection can be compared to the model and be discarded. Additionally, keypoints from the model can be weighted according to how often they appear in training images.

Run time has not been the focus of this research. For figure 14, the run time was 40 seconds on my laptop. and had 28 instrument images across 4 objects. In an OR setting, more images should be used per face of the instrument, increasing run time. However, the computer on which the program is run could be much faster instrument images can be loaded before every operation. Additionally, in section 3.6 we choose to use a brute force approach to matching, instead of the Best-Bin-First algorithm. This would make the program faster. There are many factors which can influence run time, and future research should make evaluate the run time thoroughly in an operating room setting.

Cluttered image tests should be done with images from actual surgeries, where the instrument count is done with SIFT and compared to a hand count. As with any practical application, testing in the field is preferable. In this case it is necessary, as it concerns technology where errors might result in harm to patients.

6 Conclusion

Glare objects generate more keypoints and more false positives than matte objects. The SIFT algorithm should be run on different settings for both. The cluster threshold should be 3 for matte objects and 4 for glare objects in the current configuration.

Positive detections will increase when instrument images and their mirrors are used. It is unclear whether that is worth the increased run time.

The similarity transform performs better across all metrics, probably because the affine transform does not function as expected. The affine transform, when working correctly, is expected to be able to be better able to deal 3D depth object rotation than the similarity transform. If any improvements are made to this program, fixing the affine transform should be high priority.

The probability threshold was varied too slightly between 0.98 and 0.99 to be used as a tuning variable. It could still function as such, given more research.

Run time is a factor to be considered when implementing the program in an operating room setting.

Can the Scale Invariant Feature Transform algorithm reliably detect objects in an operating room setting? Yes, it can. Conceptually, it could detect instruments with 100% accuracy, provided all instruments are visible in the scene image.

A Variables and constants

There are many variables and constants I've used in section 2. In this appendix, they are displayed in table 10. Many of these constants are tested or assumptions by Lowe. We do not test most of these, but they are choices that could be adjusted to improve the algorithm.

Table 10: Variables, constants and choices made in this application of SIFT.

	Parameter Value	Section
k	$\sqrt{2}$	3.2
σ_0	1.6	3.2
Extrema threshold	0.03	3.3
Edge ratio	12.1	3.3
Gaussian window σ	1.5 scale	3.5
Keypoint descriptor area	16x16 pixels	3.5
Histogram vector	4x4x8=128	3.5
Gradient magnitude cap	0.2	3.5
Euclidian distance ratio	0.8	3.6
Min # keypairs sim	2	3.7
Min # keypairs aff	3	3.7
Degrees per bin	10	3.9
Bin size for d_x, d_y	10	3.9
Bin size for scale	10	3.9
Bin size for orientation	10	3.9
Bin size for verification	Half bin size	3.9
Probability threshold	0.98	3.11
IoU threshold	0.9	3.12

Sigma multiplier k is chosen as is because the chosen value allows for a downsampling of the image with a factor 2, saving computational time. A greater σ_0 improves repeatability of keypoint detection, but increases computational time. At the 1.6, the repeatability is close to optimal.

The Taylor fit allows a value to be placed on the extremum peak. If extrema are lower than the given value, they are removed because these are unstable extrema with low contrast.

To find and remove extrema on the edge, compute the sum of the straight derivatives over the diagonal derivative around the extremum. The ratio between these can be thresholded [7].

In 3.4, 10 degrees per bin was chosen. Fewer degrees per peak lead to more peaks, but lower. This increases precision but also computational time. Fewer degrees per bin would lead to more duplicates, which means more keypoints. On the other hand, more keypoints also mean more false positives.

The Gaussian weighting window σ is chosen to be 1.5 times scale of the keypoint. Larger windows could improve accuracy, as more information is taken into account. However, it could also decrease accuracy, as at a certain pixel distance useful information carry over is zero.

If peaks are within an 80% threshold, the keypoint is duplicated and given this secondary orientation. This increases the number of keypoints. However, it also increases the number of false positives. At 80%, 15% are duplicated.

The Gaussian weighting window σ in 3.5 is half the descriptor window, which is 8 pixels in our case. Its purpose is to remove is to reduce boundary issues by adding a gradual decline of effectiveness of pixels farther away from the descriptor position. Increasing σ increases the effect of faraway pixels within the window, making the boundary issues more prominent. Decreasing σ also increases boundary effects, as the gradual decline is lessened.

The keypoint descriptor area is chosen to be 16x16 pixels. This can be bigger, but again, it would take into account pixels that wouldn't contribute positive information. It could be smaller, but pixels could be missed that do contribute positive information. The area allows for 4x4 descriptors.

The histograms have 8 pixels, leading to 4x4x8 = 128 element feature vectors. Larger feature vectors can decrease matching by making the histograms too sensitive to distortion. Smaller vectors perform worse as well in terms of matching, though computational time decreases. The descriptor is sensitive to affine change, falling below an 80% matching repeatability for a viewpoint angle difference of 40 degrees or more.

Lowering the cap decreases gradient emphasis. The goal is to make sure gradient contribution isn't overbearing, but also not so low that it's irrelevant. The gradient magnitude cap was determined experimentally. The Euclidian distance ratio of section 3.6 is set to 0.8. This eliminates 90% of false matches whilst discarding less than 5% of true matches.

In subsection 3.9 two keypoint pairs are needed for the similarity transform model and three for the affine transform model.. However, the minimum could be increased to a number greater than 3. This would lead to models being accepted, but also a lower false positive rate.

The bin sizes are chosen for the similarity transform, not the affine transform. This means they are quite broad to catch non-rigid transformations. Decreasing bin size would lead to less models, but a lower false positive rate.

In subsection 3.10, half the bin size of section 3.9 is chosen to test the matches against. This is to remove any outliers that aren't within the scope of the model. A stricter test would remove more potential models, decreasing the false positive rate but increasing the false negative rate. A looser test does the opposite.

In subsection 3.11, the main variable is the probability threshold. This is set at 0.98 by Lowe. Varying this threshold is an excellent way of testing the algorithm. If the threshold increases, and false positives are removed, then the threshold should be higher. The value at which correct identifications are removed is too high. However, if the threshold increases and correct identifications are removed before the false positives, then something else is wrong.

The duplicate removal threshold is set to 0.5. This application of the algorithm, with its many potential instrument identifications per instrument folder, needed a strict duplicate removal threshold. The disadvantage of this is that overlaying instruments have a higher chance of being removed. To counteract this, the instrument photos need to be cropped as close to the instrument as possible, as to decrease the area of the bounding boxes.

B Affine

$$s = \sqrt{m_1 m_4 - m_3 m_2} \tag{17}$$

$$D = m_1 m_4 - m_3 m_2 \tag{18}$$

$$C = m_2^2 + m_4^2 \tag{19}$$

$$R = sqrts(s - Col) \tag{20}$$

$$Arc_1 = \left(\frac{R(m_1 m_4^2 - m_2 m_3 m_4)}{CsD} + m_2^2 m_3 - m_1 m_2 m_4\right) \tag{21}$$

$$Arc_2 = \left(\frac{R(m_1 m_2 m_4 - m_2^2 m_3)}{CsD} + m_1 m_4^2 - m_2 m_3 m_4\right) \tag{22}$$

$$\theta = \arctan \frac{Arc_1}{Arc_2} \tag{23}$$

$$c_x = \frac{R}{D} \tag{24}$$

$$c_y = \frac{1}{C}\left(-m_1 m_4 \frac{R}{D} + m_2 m_3 \frac{R}{D} + m_1 m_2 + m_3 m_4\right) \tag{25}$$

C Probability

$P(m)$ is the chance that a single keypair is correct, which is the ratio of correctly matched image features to all matched features in a typical image. This is about 0.01. See appendix A. The probability of matched keypairs f given that m is not present can be computed with the binomial distribution 26.

$$P(f|-m) = \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} \tag{26}$$

Here the probability p is given by the following equation 27.

$$p = dlrs \tag{27}$$

Where d is the probability of accidentally selection a database match to the current model, which is the matches of the current model divided by all the matches. l is the probability of accidentally satisfying the location constraints, which is the bin size in both dimensions multiplied, which is 1/16. r the probability of accidentally satisfying the orientation restrains given our bin sizes, which is 30/360 = -0.85 and s the probability of accidentally satisfying the scale constraints, which is 0.5.

D Code

```python
import numpy as np
import scipy as sp
import scipy.special
import shapely.geometry
from shapely. ops import cascaded_union
from shapely.geometry import Polygon
import math
from math import pi
import matplotlib
matplotlib.rcParams['backend'] = 'tkagg'
import matplotlib.pyplot as plt
import matplotlib.gridspec as gs
import cv2
from tqdm import tqdm

import warnings
warnings.filterwarnings("error")

import sys, os
from datetime import datetime

autofill = True

## Constants
binSizeAngle = 30
binSizeScaleFactor = 2
binSizeXFactor = 0.25
binSizeYFactor = 0.25
polygonThreshold = 0.25
transformChooser = False
# transformChooser = True
clusterThreshold = 2
```

```

probL = binSizeXFactor*binSizeYFactor
probR = binSizeAngle/360
probS = 1/binSizeScaleFactor
probM = .01
probThreshold = 0.98

showBoxPreNMS = True
showBoxPostNMS = True

'''
logBase: x is the input of the logarithmic function, base the type of logarithm
'''
def logBase(x, base=2):
    return np.log(x)/np.log(base)

## Functions
'''
'''
def betai(a, b, x):
    '''
    Calculate incomplete beta function I_x(a,b), taken from paragraph 6.4 of Press, W.H. et al. - Numerical Recipes in C (second edition)
    '''
    if x < 0 or x > 1: raise ValueError("betai:␣x␣must␣be␣0␣<=␣x␣<=␣1")
    if x in (0, 1): bt = 0
    else: bt = np.exp(gammln(a+b)-gammln(a)-gammln(b)+a*np.log(x)+b*np.log(1-x))
    if x < (a+1)/(a+b+2): return bt*betacf(a, b, x)/a
    else: return 1-bt*betacf(b, a, 1-x)/b

def betacf(a, b, x):
    '''
    Calculate continued fraction for incomplete beta function, taken from paragraph 6.4 of Press, W.H. et al. - Numerical Recipes in C (second edition)
    '''
    maxit = 100
    eps = 3.e-7
    fpmin = 1.e-30

    qab = a+b
    qap = a+1.
    qam = a-1.
    c = 1.
    d = 1.-qab*x/qap
    if np.abs(d) < fpmin: d = fpmin
    d = 1/d
    h = d

    for m in range(1, maxit+1):
        m2 = 2*m
        aa = m*(b-m)*x/((qam+m2)*(a+m2))
        d = 1+aa*d
        if np.abs(d) < fpmin: d = fpmin
        c = 1+aa/c
        if np.abs(c) < fpmin: c = fpmin
        d = 1/d
        h *= d*c
        aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2))
        d = 1+aa*d
        if np.abs(d) < fpmin: d = fpmin
        c = 1+aa/c
        if np.abs(c) < fpmin: c = fpmin
        d = 1/d
        de = d*c
        h *= de
        if np.abs(de-1) < eps: break

    if m > maxit: raise ValueError("betacf:␣a␣or␣b␣too␣big,␣or␣maxit␣too␣small")
    return h

def gammln(xx):
    '''
    Calculate natural logarithm of gamma function, taken from paragraph 6.1 of Press, W.H. et al. - Numerical Recipes in C (second edition)
    '''
    cof = [76.18009172947146, -86.50532032941677, 24.01409824083091, -1.231739572450155, 0.1208650973866179e-2, -0.5395239384953e-5]
    y = x = xx
    tmp = x+5.5
    tmp -= (x+.5)*np.log(tmp)
    ser = 1.000000000190015
    for j in range(6):
        y += 1
        ser += cof[j]/y
    return -tmp+np.log(2.5066282746310005*ser/x)

'''
instrumentCounter: keeps track of which instrument is being considered at any given iteration of i
photoCount tracks which photo is being looked at according the the directories, whatIsDetected gives the name according to the directories, i is instrument index
returns the idintifier for the instrument
'''
def instrumentCounter(photoCount,whatIsDetected,i):
    # photoCountSum = 0
    photoCountSum = 0
    for instrument in range(len(whatIsDetected)):
        photoCountSum += int(photoCount[instrument])
        if np.floor(i/2) <= photoCountSum:
```

```
            instrumentIdentifier = whatIsDetected[
                 instrument]
            return instrumentIdentifier

'''
keypointDifference: calculates the differences
     between keypoints.
q is the for loop that iterates over the number of
      matches. featMatches is a list of all the
     matches between keypoints.keyPointsImg is a
     list of all the keypoints of the image.
keyPointsImgsInstruments is the same, but for all
     the keypoints of a given instrument of i. i is
      the index going through all the instrument
     images
Returns the difference in angle, in scale, in x
     position, in y position and the indices of the
      image and instrument keypoints.
'''
def keypointDifference(q,featMatches,keypointsImg,
     keyPointsImgsInstruments,i):

    indexImgKP = featMatches[q,0].queryIdx
    indexInstrumentKP = featMatches[q,0].trainIdx

    angleImgKP = keypointsImg[indexImgKP].angle*2*
         pi/360
    angleInstrumentKP = keypointsImgsInstruments[i
         ][indexInstrumentKP].angle*2*pi/360

    scaleImgKP = keypointsImg[indexImgKP].size
    scaleInstrumentKP = keypointsImgsInstruments[i
         ][indexInstrumentKP].size

    xImgKP, yImgKP = keypointsImg[indexImgKP].pt
    xInstrumentKP, yInstrumentKP =
         keypointsImgsInstruments[i][
         indexInstrumentKP].pt

    dangle = angleInstrumentKP - angleImgKP
    dangle %= 2*pi

    dscale = scaleImgKP/scaleInstrumentKP


    dx = (xInstrumentKP-imgsInstruments[i].shape
         [1]/2) * dscale
    dy = (yInstrumentKP-imgsInstruments[i].shape
         [0]/2) * dscale
    dx, dy = (np.array([[np.cos(-dangle), -np.sin
         (-dangle)], [np.sin(-dangle), np.cos(-
         dangle)]]) @ np.array([[dx], [dy]]))[:,0]
    dx, dy = xImgKP-dx, yImgKP-dy
'''
hashTableFunc stores all keypair contributions to
     the hough transform in a hashTable.
imgGray is the grayed version of the image.
     imgsGrayInstruments[i] is the same for every
     instrument image i. binSizeXFactor is the size
      of the bin defined in variables.
Same for binSizeAngle and binSizeScaleFactor.
Doesn't return, but updates the hashTable
'''
def hashTableFunc(imgGray,imgsGrayInstruments,
     binSizeXFactor,binSizeYFactor,binSizeAngle,
     binSizeScaleFactor,dx,dy,dscale,dangle,
     indexImgKP,indexInstrumentKP,hashTable,i):
    xSizeImg = imgGray.shape[1]
    ySizeImg = imgGray.shape[0]
    xSizeInstrument = imgsGrayInstruments[i].shape
         [1]
    ySizeInstrument = imgsGrayInstruments[i].shape
         [0]
    maxSizeInstrument = max(xSizeInstrument,
         ySizeInstrument)

    binSizeX, binSizeY = np.array([binSizeXFactor,
          binSizeYFactor])*maxSizeInstrument*dscale
    nrXBins = np.ceil(xSizeImg/binSizeX)
    nrYBins = np.ceil(ySizeImg/binSizeY)

    angleHash = (dangle*360/(2*pi))/binSizeAngle
    scaleHash = logBase(dscale, base=
         binSizeScaleFactor)
    xHash = dx*nrXBins/xSizeImg
    yHash = dy*nrYBins/ySizeImg

    for iAngle in int(np.floor(angleHash))+np.
         arange(2):
        for iScale in int(np.floor(scaleHash))+np.
             arange(2):
            for iXHash in int(np.floor(xHash))+np.
                 arange(2):
                for iYHash in int(np.floor(yHash))+
                     np.arange(2):
                    key = str(iAngle)+str(iScale)+
                         str(iXHash)+str(iYHash)
                    if key in hashTable.keys():
                        hashTable[key] += [[
                             indexImgKP,
                             indexInstrumentKP,
                             dangle, dscale, dx, dy,
                             binSizeX, binSizeY]]
                    else:
                        hashTable[key] = [[
                             indexImgKP,
                             indexInstrumentKP,
                             dangle, dscale, dx, dy,
                             binSizeX, binSizeY]]
    return

'''
hashTableClusterRemover removes clusters of
     keypairs if they are less than 3 keypairs or
     if the model they produce gives an error.
hashtableItems is a listed accessible version of
     the hashTable, q iterates over every cluster
```

```
in hashTable items, clusterThreshold is defined in variables and reset is used to reset the while loop this function is in.
returns the hashTableItems and a reset. Also updates the hashTable.
'''
def hashTableClusterRemover(hashTableItems,q,clusterThreshold,reset):
    nInstrumentKeypoints = len(np.unique([hashTableItems[q][1][s][1] for s in range(len(hashTableItems[q][1
    if nInstrumentKeypoints < clusterThreshold or reset == 'remove':
        keyToRemove = hashTableItems[q][0]
        hashTable.pop(keyToRemove)
        del hashTableItems[q]
        reset = True
    return hashTableItems, reset

'''
similarityTransform gives a model by checking if clusters from the hashTable are within the bins as defined in variables. This transform takes into account rotation and scale.
All inputs are found in descriptions above.
returns modelMat, the model that transforms the instrument coordinates to the image coordinates. modelParametersMat gives the rotation and scale in one matrix.
modelParametersTrans gives the translation separately in one vector. reset, again gives the option to reset if one of the keypairs in the cluster didn't fall within certain parameter bounds
'''
def similarityTransform(i,q,clusterThreshold,hashTableItems,keypointsImg,keypointsImgsInstruments,dangle,dscale,dx,dy,reset):
    affineA = np.zeros((2*(len(hashTableItems[q][1])), 4))
    affineB = np.zeros((2*(len(hashTableItems[q][1])), 1))
    keypointIndices = []
    for r in range(len(hashTableItems[q][1])):
        [indexImgKP, indexInstrumentKP] = hashTableItems[q][1][r][:2]
        xKeypointImg = keypointsImg[indexImgKP].pt[0]
        yKeypointImg = keypointsImg[indexImgKP].pt[1]
        xKeypointInstrument = keypointsImgsInstruments[i][indexInstrumentKP].pt[0] - imgsInstruments[i].shape[1]/2
        yKeypointInstrument = keypointsImgsInstruments[i][indexInstrumentKP].pt[1] - imgsInstruments[i].shape[0]/2

        affineA[2*r, :] = np.array([xKeypointInstrument, -yKeypointInstrument, 1, 0])
        affineA[2*r+1, :] = np.array([yKeypointInstrument, xKeypointInstrument, 0, 1])
        affineB[2*r] = xKeypointImg
        affineB[2*r+1] = yKeypointImg
        keypointIndices += [indexImgKP, indexInstrumentKP]

        modelParameters = np.linalg.lstsq(affineA, affineB, rcond=None)[0][:,0]

        modelParametersMat = np.array([[modelParameters[0], -modelParameters[1]], [modelParameters[1], modelParameters[0]]])
        modelParametersTrans = np.array([[modelParameters[2]], [modelParameters[3]]])
        affineAngle = float(2*pi-np.arctan2(modelParameters[1], modelParameters[0]))%(2*pi)
        affineScale = float(np.sqrt(modelParameters[0]**2 + modelParameters[1]**2))
        affineX = float(modelParameters[2])
        affineY = float(modelParameters[3])
        reset = False

    for t in range(len(hashTableItems[q][1])):
        dangle, dscale, dx, dy, binSizeX, binSizeY = hashTableItems[q][1][t][2:]
        binSizeScale = 2**(np.floor(np.log2(dscale))+1)-2**(np.floor(np.log2(dscale)))
        if (np.abs(dangle-affineAngle) <= 0.5*binSizeAngle*2*pi/360) and (np.abs(dscale-affineScale) <= 0.5*binSizeScale) and (np.abs(dx-affineX) <= 0.5*binSizeX) and (np.abs(dy-affineY) <= 0.5*binSizeY):
            R = np.array([[np.cos(2*pi-affineAngle), -np.sin(2*pi-affineAngle)], [np.sin(2*pi-affineAngle), np.cos(2*pi-affineAngle)]])
            modelMat = affineScale*R
            pass
        else:
            keyToChange=hashTableItems[q][0]
            valuesToChange=hashTable.pop(keyToChange)
            del valuesToChange[t]
            if len(valuesToChange) != 0:
                hashTable[keyToChange]=valuesToChange
            reset = True
            return 0, 0, 0, 0, reset
```

```
    return modelMat, modelParametersMat, modelParametersTrans, keypointIndices, reset
'''
affineTransform gives a model in the same way as the similarityTransform, only adding an affine element.
returns the same only modelParameters also incorporates the affine element in the matrix.
The reset can be set to 'remove', which removes an entire cluster instead of one keypair if the found model gives an error.
'''
def affineTransform(i,q,clusterThreshold,hashTableItems,keypointsImg,keypointsImgsInstruments,dangle,dscale,dx,dy,reset):
    affineA = np.zeros((2*(len(hashTableItems[q][1])), 6))
    affineB = np.zeros((2*(len(hashTableItems[q][1])), 1))
    keypointIndices = []

    for r in range(len(hashTableItems[q][1])):
        [indexImgKP, indexInstrumentKP] = hashTableItems[q][1][r][:2]
        xKeypointImg = keypointsImg[indexImgKP].pt[0]
        yKeypointImg = keypointsImg[indexImgKP].pt[1]
        xKeypointInstrument = keypointsImgsInstruments[i][indexInstrumentKP].pt[0] - imgsInstruments[i].shape[1]/2
        yKeypointInstrument = keypointsImgsInstruments[i][indexInstrumentKP].pt[1] - imgsInstruments[i].shape[0]/2

        affineA[2*r,:] = np.array([xKeypointInstrument, yKeypointInstrument, 0, 0, 1, 0])
        affineA[2*r+1,:] = np.array([0, 0, xKeypointInstrument, yKeypointInstrument, 0, 1])
        affineB[2*r] = xKeypointImg
        affineB[2*r+1] = yKeypointImg
        keypointIndices += [indexImgKP, indexInstrumentKP]

    modelParameters = np.linalg.lstsq(affineA, affineB, rcond=None)[0][:,0]
    modelParametersMat = np.array([[modelParameters[0], modelParameters[1]], [modelParameters[2], modelParameters[3]]])
    modelParametersTrans = np.array([[modelParameters[4]], [modelParameters[5]]])

    m = modelParameters
    detM = m[0]*m[3]-m[2]*m[1]

    if detM<=0:
        reset = "remove"
        return 0, 0, 0, 0, reset, q

    col2lenM = m[1]**2+m[3]**2
    rootMtemp = -detM*(detM-col2lenM)

    if abs(rootMtemp) < 0.01:
        rootMtemp = 0

    if rootMtemp < 0:
        reset = "remove"
        return 0, 0, 0, 0, reset, q

    rootM = np.sqrt(rootMtemp)

    affineScale = np.sqrt(detM)
    arctan2Part1 = 1/(col2lenM*affineScale)*(rootM/detM*(m[0]*m[3]**2-m[1]*m[2]*m[3])-m[0]*m[1]*m[3]+m[1]**2*m[2])
    arctan2Part2 = 1/(col2lenM*affineScale)*(rootM/detM*(m[0]*m[1]*m[3]-m[1]**2*m[2])+m[0]*m[3]**2-m[1]*m[2]*m[3])
    affineAngle = (2*pi-np.arctan2(arctan2Part1, arctan2Part2))%(2*pi)
    affineAngleCheck = (2*pi-np.arctan2(arctan2Part1,-arctan2Part2))%(2*pi)
    affineSkewX = rootM/detM
    affineSkewY = 1/col2lenM*(-m[0]*m[3]*rootM/detM+m[1]*m[2]*rootM/detM+m[0]*m[1]+m[2]*m[3])
    affineX = modelParameters[4]
    affineY = modelParameters[5]

    for t in range(len(hashTableItems[q][1])):
        dangle, dscale, dx, dy, binSizeX, binSizeY = hashTableItems[q][1][t][2:]
        binSizeScale = 2**(np.floor(np.log2(dscale))+1)-2**(np.floor(np.log2(dscale)))

        if (np.abs(dangle-affineAngle) <= 0.5*binSizeAngle*2*pi/360) and (np.abs(dscale-affineScale) <= 0.5*binSizeScale) and (np.abs(dx-affineX) <= 0.5*binSizeX) and (np.abs(dy-affineY) <= 0.5*binSizeY):
            # Construct model matrix
            Cx = np.array([[1, affineSkewX], [0, 1]])
            Cy = np.array([[1, 0], [affineSkewY, 1]])
            C = Cx@Cy
            R = np.array([[np.cos(2*pi-affineAngle), -np.sin(2*pi-affineAngle)], [np.sin(2*pi-affineAngle), np.cos(2*pi-affineAngle)]])
```

```
            modelMat = affineScale*R@C

            return modelMat, modelParametersMat, modelParametersTrans, keypointIndices, reset, q
        else:
            keyToChange=hashTableItems[q][0]
            valuesToChange=hashTable.pop(keyToChange)
            del valuesToChange[t]
            if len(valuesToChange) != 0:
                hashTable[keyToChange]=valuesToChange
            return 0, modelParametersMat, modelParametersTrans, keypointIndices, reset, q

'''
probability checks whether a found model is likely to be a correct model, and removes those not able to make the threshold.
keypointsImgCoords is the location of any given keypoint in the image. probL,R,S are the probabilities that
returns the output that can be used to calculate bounding boxes
'''
def probability(modelMat,modelParametersMat,keypointsImgCoords,modelParametersTrans,imgsInstruments,i,probL,probR,probS,hashTableItems,probThreshold,instrumentIdentifier,output):
    keypointsImgModelCoords = np.linalg.inv(modelParametersMat)@(keypointsImgCoords-modelParametersTrans)
    keypointsImgModelCoordsInds = np.where((keypointsImgModelCoords[0,:] >= -imgsInstruments[i].shape[1]/2) & (keypointsImgModelCoords[0,:] < imgsInstruments[i].shape[1]/2) & (keypointsImgModelCoords[1,:] >= -imgsInstruments[i].shape[0]/2) & (keypointsImgModelCoords[1,:] < imgsInstruments[i].shape[0]/2))[0]
    keypointsImgModelCoords = keypointsImgModelCoords[:,keypointsImgModelCoordsInds]
    n = keypointsImgModelCoords.shape[1]

    # 2) Calculate probability of accidentally matching a single image feature to the current model pose
    probD = len(keypointsImgsInstruments[i])/np.sum([len(keypointsImgInstrument) for keypointsImgInstrument in keypointsImgsInstruments])
    p = probD*probL*probR*probS

    # 3) Calculate probability that feature matches were accidental, given that the model is not present (binomial probability CMF, equation 6.4.12 from Press, W.H. et al. - Numerical Recipes in C (second edition))
    probMatchesNoModel = betai(len(hashTableItems[q][1]), n-len(hashTableItems[q][1])+1, p)

    # 4) Calculate probability that the model is present, given the feature matches
    probModelGivenMatches = probM/(probM + probMatchesNoModel)
    # 5) Accept or reject model
    if probModelGivenMatches >= probThreshold:
        output[i].append([modelMat, modelParametersTrans, keypointIndices, instrumentIdentifier, probModelGivenMatches])
    return


'''
boundingCalculator finds the bounding boxes in the scene image according to the instrument images. output is an collection of the relevant modelParameters,
the keypointIndices, the probability and the type of instrument.
returns a list of bounding boxes, and plots them too if desired.
'''
def boundingCalculator(imgsGrayInstruments,imgsInstruments,output):
    colours = matplotlib.rcParams['axes.prop_cycle'].by_key()['color']
    nInstrumentCols = int(max(1, np.floor(np.sqrt(len(imgsGrayInstruments)))))
    nInstrumentRows = int(np.ceil(len(imgsGrayInstruments)/nInstrumentCols))
    listPointsModel = []
    for indImg in range(len(imgsGrayInstruments)):
        y = int(np.floor(round(indImg/nInstrumentCols, 1)))
        x = indImg%nInstrumentCols

        # Show instrument
        imgInstrument = imgsInstruments[indImg].copy()
        imgInstrument = cv2.cvtColor(imgInstrument, cv2.COLOR_BGR2RGB)

        ax = fig.add_subplot(nInstrumentRows, nInstrumentCols*2, nInstrumentCols*(y+1) + x+1 + y*nInstrumentCols
        ax.imshow(imgInstrument)
        ax.axis('off')
        rectPoints = np.array(np.meshgrid([0, imgInstrument.shape[1]], [0,
```

```
            imgInstrument.shape[0]])).T.reshape((-1,2))[[0,1,3,2],:]
        rectPoints = np.append(rectPoints, rectPoints[0,None,:], axis=0)
        ax.plot(rectPoints[:,0], rectPoints[:,1], colours[indImg%len(colours)])
        ax.set_ylim([imgInstrument.shape[0]+1, -2])
        ax.set_xlim([-2, imgInstrument.shape[1]+1])

        # Show detections
        currentOutput = output[indImg]
        modelCounter = 0
        for o in currentOutput:
            modelParametersMat = o[0]
            modelParametersTrans = o[1]
            rectPointsModel = rectPoints - np.array([imgInstrument.shape[1]/2, imgInstrument.shape[0]/2])
            rectPointsModel = modelParametersMat @ rectPointsModel.T + modelParametersTrans
            rectPointsModel = rectPointsModel.T
            listPointsModel.append([rectPointsModel,o[3],o[4],indImg,modelCounter])
            modelCounter +=1
            axImg.plot(rectPointsModel[:,0], rectPointsModel[:,1], colours[indImg%len(colours)]) # Plot bounding rectangle
    return listPointsModel

'''
duplicateRemover finds duplicates according to a threshold and removes these with a Non-Maximum Suppression algorithm.
PolygonThreshold is the chosen threshold at which duplicates are removed.
returns a list of non-duplicate bounding boxes and a list of the removed duplicates. The latter one is used to remove duplicates from output as well.
'''
def duplicateRemover(listPointsModel, polygonThreshold):
    polygonList = [] #make polygonlist including instrument and probability
    for model in range(len(listPointsModel)):
        polyCoordinates = ((listPointsModel[model][0][0][0],listPointsModel[model][0][0][1]),(listPointsModel[model][0][1][0],listPointsModel[model][0][1][1]),(listPointsModel[model][0][2][0],listPointsModel[model][0][2][1]),(listPointsModel[model][0][3][0],listPointsModel[model][0][3][1]))
        currentPolygon = Polygon(polyCoordinates)
        polygonList.append([currentPolygon, listPointsModel[model][1], listPointsModel[model][2], listPointsModel[model][0], listPointsModel[model][3], listPointsModel[model][4]])
    ## NMS
    polygonProposal = []
    nextBestProposal = 0
    negativeProposal = []
    while len(polygonList) > 0:
        maxProbPolygon = 0
        if len(polygonList) == 1:
            bestPolygon = polygonList.pop(maxProbPolygon)
            polygonProposal.append(bestPolygon)
            break

        for polygons in range(len(polygonList)-1):
                if polygonList[maxProbPolygon][2] > polygonList[polygons+1][2]:
                    maxProbPolygon = maxProbPolygon
                else:
                    maxProbPolygon = polygons+1

        bestPolygon = polygonList.pop(maxProbPolygon
        polygonProposal.append(bestPolygon)

        toBePopped = []
        for props in range(len(polygonList)):
            polygonU = [polygonProposal[nextBestProposal][0], polygonList[props][0]]
            union = cascaded_union(polygonU)
            intersection = polygonProposal[nextBestProposal][0].intersection(polygonList[props][0])
            IoU = intersection.area/union.area
            if IoU > polygonThreshold:
                toBePopped.append(props)
        toBePopped.reverse()
        for poppers in range(len(toBePopped)):
            thePopped = polygonList.pop(toBePopped[poppers])
            negativeProposal.append(thePopped)
        nextBestProposal += 1

    return polygonProposal, negativeProposal

## Ask path to image to load
if autofill:
    print("Autofill␣enabled")
    pathImgSrc = "YourImagePath"
else:
    pathImgSrc = 0
    while pathImgSrc == 0 or not os.path.isfile(pathImgSrc) or not pathImgSrc[-4:].lower() in (".jpg", ".png"):
        print("Paste␣the␣path␣to␣the␣image␣to␣detect␣instruments␣in␣below:" if
```

```
            pathImgSrc==0 else "The␣specified␣path␣doesn't␣point␣to␣an␣existing␣image.␣Please␣try␣again:")
        pathImgSrc = input()
        if pathImgSrc[0] == "&":
            pathImgSrc = pathImgSrc[1:]
            pathImgSrc = pathImgSrc.strip()
        if (pathImgSrc[0] == "\"" and pathImgSrc[-1] == "\"") or (pathImgSrc[0] == "\'" and pathImgSrc[-1] == "\'"):
            pathImgSrc = pathImgSrc[1:-1]
            print()


## Ask path to directory of directories with instruments to detect
if autofill:
    pathInstrumentsSrc = "YourInstrumentPath"

else:
    pathInstrumentsSrc = 0
    while pathInstrumentsSrc == 0 or not os.path.isdir(pathInstrumentsSrc):
        print("Paste␣the␣path␣to␣the␣directory␣with␣instrument␣ground-truth␣images␣below:" if pathInstrumentsSrc==0 else "The␣specified␣path␣doesn't␣point␣to␣an␣existing␣directory.␣Please␣try␣again:")
        pathInstrumentsSrc = input()
        if pathInstrumentsSrc[0] == "&":
            pathInstrumentsSrc = pathInstrumentsSrc[1:]
            pathInstrumentsSrc = pathInstrumentsSrc.strip()
        if (pathInstrumentsSrc[0] == "\"" and pathInstrumentsSrc[-1] == "\"") or (pathInstrumentsSrc[0] == "\'" and pathInstrumentsSrc[-1] == "\'"):
            pathInstrumentsSrc = pathInstrumentsSrc[1:-1]
            print()
    if pathInstrumentsSrc[-1] != "\\": pathInstrumentsSrc += "\\"


# Retrieve filenames for ground-truth images
pathsInstrumentsSrc = []
photoCount = []
whatIsDetected = []
pathDirectories = os.listdir(pathInstrumentsSrc)
for e in range(len(pathDirectories)):
    while 1==1:
        print()
        print('Detect',pathDirectories[e],'?')
        isInstrumentPresent = input("Enter␣[y]␣or␣[n].")
        if isInstrumentPresent == "y":
            print('Scanning␣for␣', pathDirectories[e])
            break
        elif isInstrumentPresent == "n":
            break
        else:
            print("Please␣enter␣[y]␣or␣[n].")
    if isInstrumentPresent == "y":
        pathInstrumentDirectory = pathInstrumentsSrc + '/' + pathDirectories[e]
        pathsInstrumentsSrc.extend([os.path.join(pathInstrumentDirectory, f) for f in os.listdir(pathInstrumentDirectory) if os.path.isfile(os.path.join(pathInstrumentDirectory, f)) and f[-4:].lower() in (".jpg", ".png")])
        photoCount.append(len(os.listdir(pathInstrumentDirectory)))
        whatIsDetected.append(pathDirectories[e])

## Load image
print("Loading␣image␣from␣file..")
img = cv2.imread(pathImgSrc)
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

## Load instrument images
print("Loading␣ground-truth␣images␣from␣directory..")
imgsInstruments = [cv2.imread(f) for f in pathsInstrumentsSrc]
imgsInstrumentsMirror = [np.flip(imgInstrument,0) for imgInstrument in imgsInstruments]
imgsInstruments += imgsInstrumentsMirror
imgsGrayInstruments = [cv2.cvtColor(imgInstrument, cv2.COLOR_BGR2GRAY) for imgInstrument in imgsInstruments]

## Apply SIFT
print("Initiating␣SIFT␣algorithm..")
sift = cv2.SIFT_create()


keypointsImg, descriptorsImg = sift.detectAndCompute(imgGray, None)
keypointsImgCoords = np.array([kp.pt for kp in keypointsImg]).T

print("Applying␣SIFT␣to␣ground-truth␣images..")
keypointsImgsInstruments = []
descriptorsImgsInstruments = []
for imgInstrument in tqdm(imgsGrayInstruments):
    keypointsAndDescriptorsImgInstrument = sift.detectAndCompute(imgInstrument, None)
    keypointsImgsInstruments += [keypointsAndDescriptorsImgInstrument[0],]
    descriptorsImgsInstruments += [keypointsAndDescriptorsImgInstrument[1],]


## Match features (detect instruments)
```

```python
bf = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck=False)

# Iterate over objects to detect
output = [[] for _ in range(len(imgsInstruments))]
allFeatMatches = [[] for _ in range(len(imgsInstruments))]
instrumentTable={}
for i in tqdm(range(len(imgsInstruments))):

    # Match features
    featMatches = np.asarray(bf.knnMatch(descriptorsImg, descriptorsImgsInstruments[i], k=2))
    imgsInstruments[i]
    instrumentIdentifier = instrumentCounter(photoCount,whatIsDetected,i)

    # Ratio test
    featMatches = featMatches[[featMatches[n,0].distance <= .99*featMatches[n,1].distance for n in range(featMatches.shape[0])]]
    allFeatMatches[i] = featMatches
    numberOfMatches = featMatches.shape[0]
    hashTable={}

    for q in range(featMatches.shape[0]):
        dangle, dscale, dx, dy, indexImgKP, indexInstrumentKP = keypointDifference(q,featMatches,keypointsImg,keypointsImgsInstruments,i)
        hashTableFunc(imgGray,imgsGrayInstruments,binSizeXFactor,binSizeYFactor,binSizeAngle,binSizeScaleFactor,dx,dy,dscale,dangle,indexImgKP,indexInstrumentKP,hashTable,i)

    # Geometric verification through Least Squares to obtain affine projection
    hashTableItems = list(hashTable.items())
    q = 0
    while q < len(hashTableItems):
        reset = False
        hashTableItems,reset = hashTableClusterRemover(hashTableItems,q,clusterThreshold,reset)
        if reset == True:
            continue
        if transformChooser == False:
            modelMat, modelParametersMat, modelParametersTrans, keypointIndices, reset = similarityTransform(i,q,clusterThreshold,hashTableItems,keypointsImg,keypointsImgsInstruments,dangle,dscale,dx,dy,reset)
            if reset == True:
                continue
        elif transformChooser == True:
            modelMat, modelParametersMat, modelParametersTrans, keypointIndices, reset, q = affineTransform(i,q,clusterThreshold,hashTableItems,keypointsImg,keypointsImgsInstruments,dangle,dscale,dx,dy,reset)
            if reset == 'remove':
                hashTableItems,reset = hashTableClusterRemover(hashTableItems,q,clusterThreshold,reset)
            if reset == True:
                continue
        probability(modelMat,modelParametersMat,keypointsImgCoords,modelParametersTrans,imgsInstruments,i,probL,probR,probS,hashTableItems,probThreshold,instrumentIdentifier,output)
        q+=1


## Show original image and ground-truth images with features drawn
## Show original image with boxes around detected instruments

if showBoxPreNMS == True:
    fig = plt.figure()
    axImg = fig.add_subplot(1, 2, 1)
    axImg.imshow(img)
    axImg.axis('off')
listPointsModel = boundingCalculator(imgsGrayInstruments,imgsInstruments,output)
polygonProposal, negativeProposal = duplicateRemover(listPointsModel, polygonThreshold)

# Count instruments
instrumentCount = 0
for iden in range(len(whatIsDetected)):
    for detection in range(len(output)):
        instrumentCount +=len(output[detection])
print("Showing",instrumentCount, "detected instruments")

#Remove duplicates from output
negativeProposal.sort(key=lambda x: (x[4],x[5]), reverse = True)
for q in range(len(negativeProposal)):
    del output[negativeProposal[q][4]][negativeProposal[q][5]]

# Count instruments
instrumentCount = 0
for iden in range(len(whatIsDetected)):
    for detection in range(len(output)):
```

```
        instrumentCount +=len(output[detection])

print("Showing",instrumentCount, "detected instruments")

# Show bounding boxes after duplicate remover.
if showBoxPostNMS == True:
    fig = plt.figure()
    axImg = fig.add_subplot(1, 2, 1)
    axImg.imshow(img)
    axImg.axis('off')
listPointsModel = boundingCalculator(imgsGrayInstruments,imgsInstruments,output)
```

References

[1] Bani-Hani, K.E., Gharaibeh, K.A., Yaghan, R.J.: Retained surgical sponges (gossypiboma). Asian Journal of Surgery **28(2):109-15** (April 2005). https://doi.org/10.1016/s1015-9584(09)60273-6

[2] Ebrahim Karami, S.P., Shehata, M.: Image matching using sift, surf, brief and orb: Performance comparison for distorted images. S2015 Newfoundland Electrical and Computer Engineering Conference (2015)

[3] Egorova, N.N., Moskowitz, A., Gelijns, A., Weinberg, A., Curty, J., Rabin-Fastman, B., Kaplan, H., Cooper, M., Fowler, D., Emond, J.C., Greco, G.: Managing the prevention of retained surgical instruments: what is the value of counting? Annals of Surgery **247(1):13-8** (January 2008). https://doi.org/10.1097/SLA.0b013e3180f633be

[4] Gawande, A.A., Studdert, D.M., Orav, E.J., Brennan, T.A., Zinner, M.J.: Risk factors for retained instruments and sponges after surgery. New England Journal of Medicine **348(3):229-35** (January 2003). https://doi.org/10.1056/NEJMsa021721

[5] Lowe, G, D.: Object recognition from local scale-invariant features. Proceedings of the Seventh IEEE International Conference on Computer Vision **pp. 1150-1157 vol.2** (1999). https://doi.org/10.1109/ICCV.1999.790410

[6] Lowe, G, D.: Local feature view clustering for 3d object recognition. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition **Vol. 1. IEEE, 2001** (2001)

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

[8] Lowe, G, D.: Invariant features from interest point groups. BMVC **Vol. 4. 2002** (September 2002)

[9] Organization, W.H., Safety, W.P.: Implementation manual surgical safety checklist (first edition) (May 2008)

[10] Powers, D.M.W.: Evaluation: From precision, recall and f-factor to roc, informedness, markedness correlation. School of Informatics and Engineering **Technical Report SIE-07-001** (2007)

[11] Rivera, N., Mountain, R., Assumpcao, L., Williams, A.A., Cooper, A., Lewis, D.L., Benson, R.C., Miragliotta, J.A., Marohn, M., Taylor, R.H.: Assist - automated system for surgical instrument and sponge tracking. IEEE International Conference on RFID **pp. 297-30** (2008). https://doi.org/10.1109/RFID.2008.4519358

[12] Sambasivarao, K.: Non maximum suppression, a technique to filter the detections of object detectors. Towards Data Science, 2019 (October 2019)