1 Introduction

The main goal of our project is to estimate the frequency at which an object is oscillating in a silent video. Extracting such a property can lead to applications in non-invasive respiratory rate estimation [6], human blood flow detection [1], and sound extraction from vibrating objects [3].

Our project while inspired by [3] takes a more intuitive and simpler approach. By modeling the movement of tracked features (e.g. corners) on a video we can infer some important behavior of a rhythmic object, e.g, the frequency of a tuning fork or the respiratory rate of a sleeping person.

We started our project with frequency estimation of a tuning fork from a mute video and its audio retrieval and then extended the concept to respiratory rate and heart rate estimation. For this project, we had to record most of the videos in our own data set and we took advantage of the slow-motion and timelapse features of our cellphones camera.

2 Our Method

In our method, we track the rhythmic motions of interesting features frame by frame to determine the frequency of object of constant frequency in the video. Figure 1 depicts the entire workflow of our method.

The first step is temporal filtering [6]. We induce temporal filtering by using videos that were captured at a frame rate which is in the range of the frequency of the object (2x to approx 20x frequency of video). In case we have a video of frequency much higher than the frequency of the object, we dont consider all frames and skip a certain constant number of frames (n) which essentially reduces the frame rate by a factor of n. This temporal filtering is necessary to filter out higher frequencies. However, it is important to note that the frame rate of the video has to be at least twice the frequency of the object.

We use the Shi-Tomasi corner detector [5] to decide which features to track. To track all features we use Lucas-Kanade Optical Flow algorithm [4]. We group frames into windows and track features within these windows of frames. Also, these windows are contiguous...
and have an equal number of frames. The motion of these feature points in the $y$-axis is in a rhythmic pattern of a frequency same as that of the object over different windows of frames. Hence, we record the position of features in the $y$-direction in a window and use it to compute the number of peaks for each feature. In a window, we determine the frequency of each feature using the number of peaks, frame rate, and window size as seen in Figure 2. For each window of frames we determine the mode frequency and show it as a "temporal result".

Finally, a histogram containing all the frequencies estimated at each point is computed, showing the number of times a particular frequency value was assigned to a point.

We filter out the points that do not belong to the objects movement. We assume that after temporal filtering, the features what have higher displacement belong to the object of
interest in the video. Hence, we filter out outliers that have low displacements.

Finally for every window, we calculate the frequency of object by a voting technique. This technique considers the frequencies of all features within the frame window and each feature votes for a frequency. A histogram is generated and the overall frequency of the object is determined by the mode.

### 3 Feature Extraction

In our approach, we only track interesting points and calculate frequency at these points. To extract these interesting points or features, we use the Shi-Tomasi corner detector developed by [5]. This method made a modification of Harris Corner Detector and is known to give better results with optical flow algorithm which is explained in the next subsection (enter number here).

Like Harris detector, the image is converted to grayscale and the local derivative is calculated at each pixel in figure 3. To do so a window $W$ over the pixel $(x, y)$ is considered and moved by $(u, v)$ for all pixels in the grayscale image. Then the sum of squared differences ($SSD$) between these two windows is used to approximate the partial derivatives along $x-axis$ and $y-axis$ as follows.

$$SSD = [u \ v] H \begin{bmatrix} u \\ v \end{bmatrix}$$

$$H = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$$

$$A = \sum_{(x,y) \in W} I_x^2$$

$$B = \sum_{(x,y) \in W} I_x I_y$$

$$freq_p = \frac{revolutions \times fps}{window \ size}$$
\[ C = \sum_{(x,y)\in W} I_g^2 \]

On eigendecomposition of matrix \( H \), we get eigenvalues \( \lambda_{\text{max}} \) and \( \lambda_{\text{min}} \).

On eigendecomposition of matrix \( H \), we get eigenvalues \( \lambda_{\text{max}} \) and \( \lambda_{\text{min}} \).

The Shi-Tomasi corner detector differs from Harris after this point. The corners selected using a greedy method as follows:

1. Select points with \( \lambda_{\text{min}} \geq \) threshold in list of possible interesting points, \( L_1 \).
2. Sort the list of possible interesting points, \( L_1 \), by \( \lambda_{\text{min}} \).
3. Select point \( P \) with highest \( \lambda_{\text{min}} \) in \( L_1 \) and put it in list of final interesting points, \( L_2 \), and remove all points in \( L_1 \) within neighborhood of \( P \).
4. Continue till \( L_1 \) is empty.

### 4 Optical Flow

In order to calculate the frequency at the interesting points we found previously, we need to find their location at each frame. To achieve this, we use the Lucas-Kanade tracking algorithm with Pyramidal representation proposed in [4].

Basically, this algorithm tries to find the new location \( v \) in image \( J \) of a point \( u = [u_x \ u_y]^T \) from image \( I \) such that \( v = u + d = [u_x + d_x \ u_y + d_y]^T \), where \( J(v) \) is "similar" to \( I(u) \). The vector \( d \) is also known as the optical flow vector, which describes the image movement at a point \( x = [x \ y]^T \). The similarity is measured on an image neighborhood (integration window) of size \((2w_x + 1)\(2w_y + 1\). With all this being said, the vector \( d \) is defined as the vector that minimizes the following function:

\[
\epsilon(d) = \epsilon(d_x, d_y) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) - J(x + d_x, y + d_y))^2
\]

In order to tackle possible tracking sensitivity to camera motion, changes of lighting or image size, [2] proposed a pyramidal implementation for the original optical flow algorithm.
This pyramid representation is built in a recursive fashion, by resizing the image width and height in \( n \) levels, where at each iteration the height and width are divided in 2. This will allow the algorithm to handle large pixel motions, even larger than the integration window defined previously.

This algorithm works as follows: first, the optical flow for a point \( x \) is computed at the lowest level \( L_n \), then this result is propagated to level \( L_{n-1} \), and so on until you reach the original image (level \( L_0 \)). In other words, equation 1 is calculated at every level. Then, the final \( d \) optical flow vector is defined by equation 2. This process can be seen in detail in figure 4.

\[
d = \sum_{L=0}^{L_n} 2^T d^T
\]

Figure 4: Pyramidal Representation of KLT algorithm. Image extracted from http://www.mathworks.com/help/vision/ref/vision.pointtracker-class.html

5 Experiments and Results

In order to test our approach, we chose to deploy three different experiments: Tuning fork frequency estimation, Heart Rate estimation, and Respiratory rate estimation. Hopefully, these experiments will give us a notion of how accurate our system is and what are the constrains that have to be taken into account for each case scenario.

5.1 Tuning Fork Frequency Estimation

We started with this particular experiment since tuning forks have a constant frequency along time, and it was easier for us to fine tune our code. We borrowed some tuning forks of different frequencies from Jacobs School of Music and from the Physics Lab from Indiana University in order to build our own data set. Most of the videos where recorded with out cell phone’s camera.

The frequency of the tuning fork should be at least half of the sampling frequency of the camera that is recording the video, otherwise we wont be able to see the tiny vibrations of the tuning fork. Figure 5 shows an example of a tuning fork of 125Hz and a frame rate of
1200 fps. This video was collected from YouTube and the tracked points are shown in green.

![Video Sequence Output](image)

Figure 5: Video Sequence Output

As you can see, it is necessary to remove outliers, like the points on the person’s hand, if not, the frequency of these points will be considered as well and will affect the final estimation. This is done by filtering the points based on their displacement in the $y$ direction. We only consider the point with a displacement greater than half of the maximum displacement ever recorded within a particular window frame. This points are then marked in red, as shown in figure 1.

Because of the lack of high speed cameras, we were only able to record videos with our cell phones’s camera, which have a max fps of 240 with the slow motion feature. This will restrict the amount of experiments possible with tuning forks, since the maximum frequency that we can detect is less that 125Hz. We also tried to record a video of a slow motion video in order to augment the fps range, but it didn’t show good results because of the uncertainty of the actual sample frequency of the video.

The results we got for a video of a tuning fork of 125Hz, and 12000 fps are as shown in figure 6.

As you can see from figure 6a, the first result we obtained was not as accurate as the second one. This is because at the beginning, all points are being considered for the estimation, thus, the points on the person’s hand give a biased result. Later, these points are eliminated and more points that belong to the tuning fork are taking into account for the frequency estimation. Figure 7 shows the histogram of all the frequencies ever calculated for all the filtered points along the video. The window size we used for this experiment was 200 frames. The final histogram showing all the values estimated by the system is shown in figure 7. The partial result we obtained at each frame window were: 120, 120, 126, 126, 126, 126, 126, 126, 126. This means that 9 windows were taken into account and for each of them, those values were the more prominent ones.

We tried the same experiment but with a video we recorded with the slow motion feature of our cell phone’s camera and the results are shown in figure 8.

For this particular case, a little fine tuning took place, due to a sampling frequency
limitation. The maximum frame rate we have in our cameras is of 240 fps, and the tuning fork is of 125Hz, which means that we need at least 250fps in order to capture the vibrations. Because of this, we used in our system as sampling frequency 250fps instead of 240. This gave us better results, but not as accurate as in the previous example. Figure 9 shows the final results we collected.

The window size for this experiment is the same as the previous example, 200. And the temporal frequency values are as follow: 117.5, 120, 118.75, 118.75, 118.75, 118.75.

5.2 Respiratory Rate Estimation

This experiment was about Respiratory Rate Estimation, how many breaths per minute a person takes. We thought of trying this because we wanted to try experiments which can work in conditions of lack of high speed camera and with real applications. For this experiment we recorded a person sleeping and we measured the breath taken while sleeping. We used a blanket with enough details so it can be easily tracked.
The temporal filtering here was done with the timelapse video of our cellphone which has a frame rate of 2fps. This is how we learnt the importance of temporal filtering and how to do it. We understood that the frame rate of the video should be within the range of the expected object’s frequency. Results are shown in figure 10.

The window size for this experiment is the same as the previous example, 30. And the temporal frequency values are as follow: 0.333333, 0.333333, 0.333333, 0.333333. Thus, $0.33 \times 60 = 19.8$ breaths per minute and the expected result was 21 breaths per minute. The final histogram is shown is figure 11.

This could be applied for baby cam, for example, as a way of monitoring babies in a non-intrusive way.

### 5.3 Heart Rate Estimation

Finally, in order to estimate the heart rate of a person based on the involuntary movement of the head when the heart pumps blood to our brain, as shown in [1], we used the same system. Our first attempt was to drew crosses in somebody’s face to find more corner features, like shown in figure 12. However, we found out this was not necessary, instead we took a close up video of a face.

With a close up video of a face we also took care of feature outliers that were not on
the face. The only difference for this experiment in comparison with the Tuning fork one, was that in order to filter points temporarily, we considered a frame every 5 frames. This is because the video frame rate has to be within the range of the expected frequency, so instead of having a 30 fps video, we lowered that value to 6 fps by only calculating the peaks every 5 frames. The results are shown in figure 13.

The window size for this experiment was 50. And the temporal frequency values are as follow: 1.56, 1.44, 1.56, 1.56, 1.44, 1.44, 1.44, 1.44, 1.44, 1.44, 1.44, 1.44. The final histogram is shown in figure 14.

The expected result for this experiment was a heart rate of 90 beats per minute, so if we multiply our estimated frequency $1.44 \times 60 = 86.4$ bpm.

We also tried taking pulse by taking a video of the veins in our hands. Since we couldn’t make the veins evident enough we couldn’t detect any movement nor corners. Additionally, we tried to take a video of the veins in our necks but that experiment failed because, again, no corner points.

We think that as a future work, we could, instead of trying to detect corners, just detect faces and apply our algorithm to the bounding box of the face or just detect face features so we do not need to filter any points or capture the person’s face only.
6 Conclusions

As a conclusion of this project we can say that we successfully achieved the expected results and we replicated experiments proposed in papers with much less equipment, although some constrains.

Our project uses a very intuitive way for calculating the frequencies and it works with scenes that are not that controlled. We didn’t use any kind of tripod, o a very controlled environment.

For the tuning fork experiments, we were able to reconstruct the sound with a Matlab script which is attached to this project.
7 Code Usage

We used the OpenCV library version 2.4.12. To compile the code just run `make`. To run experiments, the code expects the following parameters:

```
./tracking <video-path> <frame-rate> <frame-window> (optional, default = 100)
```

If you want to skip frames, like we did in section 5.3, you have to remove the comments of a few lines from line 146 in the code.

If you want to replicate the sound of the tuning fork, you have to save to console’s output in a text file and make sure that the Matlab script take that file as input; yes, Matlab is required to run the script.

Example: 

```
./tracking tuning-fork.avi 250 > output.txt
```

Please notice that if we want to output the video with the tracked point and temporal frequency values, the code starts running very slow. We activate or deactivate this function by setting the variable `save_output_video` value in 1 or 0 respectively.
References


