Measuring heart rate with a smartphone camera

Posted on Sep 10 2013

32 Comments

Share this page:

See related: **Code** / **Blog posts** / **Projects**

There are some apps out there that can read your heart rate with a smartphone camera. No need for external pulsometers. The procedure is simple: you press the smartphone camera lens gently with your finger and, after some seconds, a reading is shown. I was wondering how these apps were doing their magic and did some experiments to get a similar system working. In this post, a basic signal processing pipeline is discussed to estimate the heart rate evolution over time from a video sequence of a fingertip touching the camera lens. Continue reading to learn more or watch directly a quick video with the result.

It is time to open your photo app and observe what happens when you cover the camera with your finger. The frame is not black but slightly red. The ambient light is traveling through your finger tissues and is reaching the camera sensor. Despite the image looks like an almost static red frame (Fig. 2), the periodic oscillations of the blood flow in the vessels produce weak brightness variations that are undetectable for the naked eye, but can be discovered with signal processing.

The experiments in this post were done offline. I recorded the videos with an iPhone 4S and copied them to my laptop. Then, I processed them with Matlab. The full code is

Ignacio Mellado Almost 13 days ago
Boston Dynamics introduces Spot. Watch this and freak out about terminators *youtu.be/M8YjvHYbZ9w #robotics*

Ignacio Mellado More than 13 days ago
Former Unbounded Robotics CEO *@meloneewise* and core team strike back with Fetch Robotics. Best of luck, guys! *spectrum.ieee.org/automaton/robo...*

Ignacio Mellado Almost 19 days ago
On my way to Cupertino. I won't miss you, London rain.

Join the conversation

- Ultrafast switching between OpenCV versions
- The relative pose problem: A chronology
- Boosting color-based tracking

Go to blog >>

- The Perceptive Portable Device
- Autonomous LinkQuad quadcopter with Computer Vision
- Autonomous Pelican quadcopter with Computer Vision

See all projects >>

available here. Matlab has a very complete signal processing toolbox that makes life easier. However, you can use GNU Octave, too, and the code should be easy to port.

Heart beat monitor

Let's start our quest for the lost signal with a proposal for a simple processing pipeline in Fig. 1.

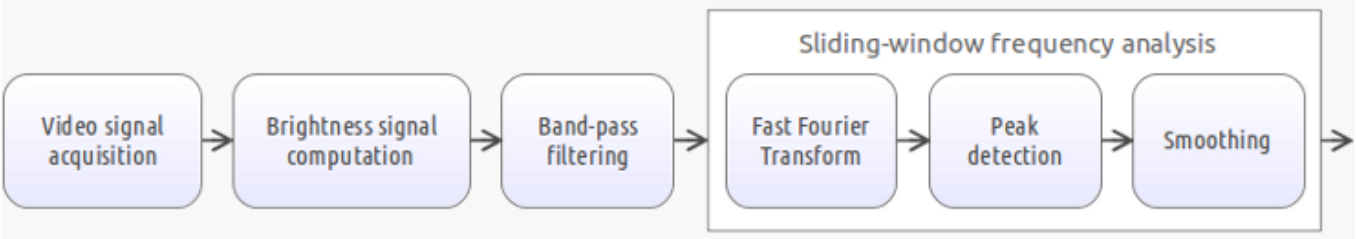


Fig. 1: Simple processing pipeline to extract the heart rate over time from a video sequence of the fingertip skin

1. Video signal acquisition

The first thing to take into account when it comes to sampling a signal is bandwidth. The normal human heartbeat is between 60 and 200 beats per minute (bpm), depending on age, fitness condition and the physical activity that the subject is doing. In my experiments, I have generously assumed that the target signal can be found between 40 and 230 bpm, i.e. 0.667 and 3.833 Hz. According to Nyquist, the sampling frequency should be at least twice the highest value (7.667 Hz) to catch the whole range of

heartbeat frequencies without aliasing. With the iPhone 4S camera, the videos are recorded at 24 or 30 frames per second depending on the shot, which is more than 3 times the minimum.

Fig. 2: When the camera is covered with a finger, the naked eye sees a static red frame, but there are subtle variations caused by the blood flow under the skin that can be recovered processing the video signal.

2. Brightness signal computation

The signal we want to process is the brightness of the skin over time. We cannot ensure that all pixels in the image will contain the brightness variation that we are looking for and we want the rest of the processing pipeline to stay computationally light. Therefore, I chose to combine all pixels into a single average brightness value per frame. On the other hand, as I am thinking about implementing the algorithm in real time, I have skipped the common image brightness computation —combining the red, green and blue planes— in favor of a simple average of all the pixels in the red plane. This is computationally much cheaper and gives very similar results because almost all the image energy is in the red plane. So, the n-th sample of the red brightness function can be expressed as:

$$b[n] = \frac{1}{W \cdot H} \sum_{x}^{W} \sum_{y}^{H} v[n, x, y, 1]$$

Where:
W: width of the image in pixels
H: height of the image in pixels
$v[n, x, y, 1]$: light level of the red plane (index 1) at $[x, y]$ coordinates of frame n in the video signal

In Matlab code, the red brightness signal can be computed with the following code snippet:

```
video = VideoReader('path to your video file here');
brightness = zeros(1, video.NumberOfFrames);
for i = 1:video.NumberOfFrames,
    frame = read(video, i);
    redPlane = frame(:, :, 1);
     brightness(i) = sum(sum(redPlane)) / (size(frame, 1) *
size(frame, 2));
end
```

As the image size is constant over time and we are only

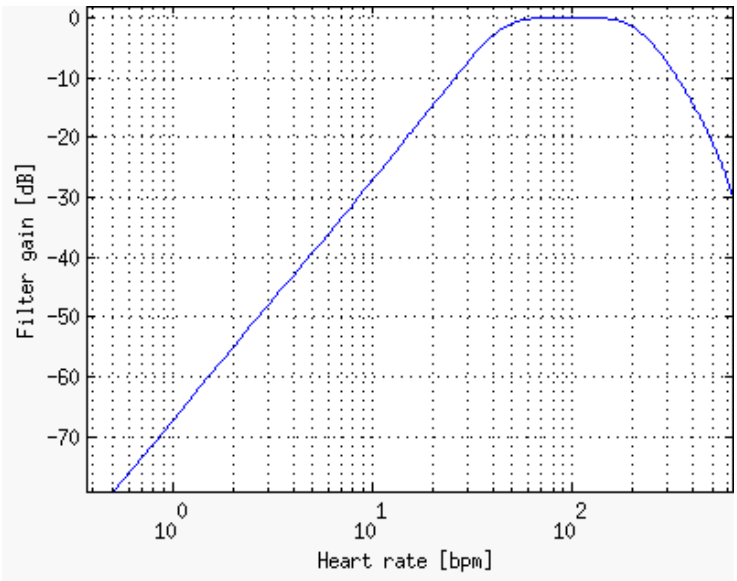
interested in the shape of the signal, not in its amplitude, we could even omit the division by the total number of pixels. I am leaving it like that because it will later help visualizing the different spectrum peaks.

3. Band-pass filtering

After the signal acquisition, a band-pass filter attenuates frequencies outside the interest band. This reduces the noise in later processing steps (peak fine-tuning) and makes the resulting heart rate signal smoother. For our case, a second-order Butterworth filter is designed. The cutoff frequencies have been set to contain our band of interest: 40-230 bpm (Fig. 3). The Matlab code to design and apply the filter is:

```
BPM_L = 40;    % Heart rate lower limit [bpm]
BPM_H = 230;   % Heart rate higher limit [bpm]
FILTER_STABILIZATION_TIME = 1;    % [seconds]
% Butterworth frequencies must be in [0, 1], where 1
corresponds to half the sampling rate
[b, a] = butter(2, [((BPM_L / 60) / v.FrameRate * 2), ((BPM_H
/ 60) / v.FrameRate * 2)]);
filtBrightness = filter(b, a, brightness);
% Cut the initial stabilization time
filtBrightness = filtBrightness((v.FrameRate *
FILTER_STABILIZATION_TIME + 1):size(filtBrightness, 2));
```

Notice how an initial piece of 1 seconds is cut off the filtered signal. It is an approximation of the time it takes for the filter to completely remove the constant signal offset (Fig. 3). If this initial piece is not removed, we might get bad readings of the heart beat during the signal stabilization time.



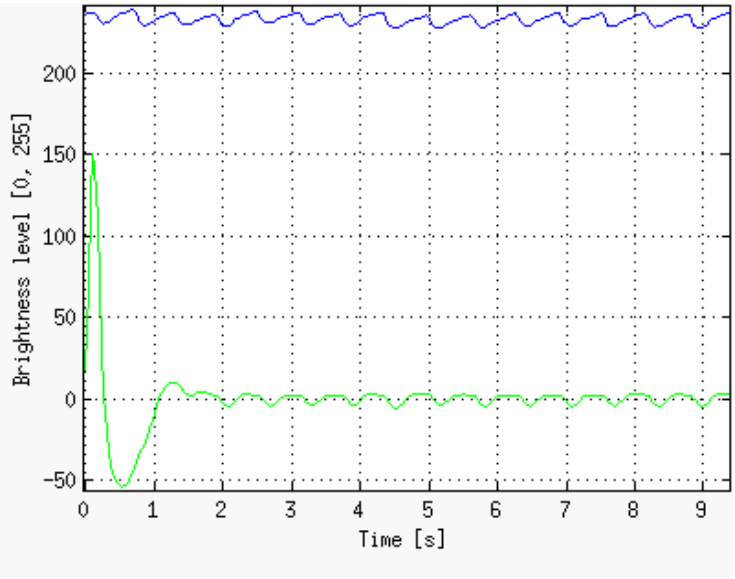


Fig. 3: On the left, the filter frequency response. On the right, original signal (blue) vs filtered signal (green). Notice the transient during the first second.

Other filter types can be tested. I chose Butterworth because:

- It is an IIR filter and the order required for a given bandwidth is much lower than with a FIR filter. Lower order usually means less computations.
- It has flat pass-band and stop-bands compared to other IIR structures that show ripples. This avoids favoring certain frequencies over others in the valid range.

4. Fast Fourier Transform

The Discrete Fourier Transform (DFT) is used to translate the signal from the time domain to the frequency domain. The Fast Fourier Transform (FFT) algorithm was used to save processing time when computing the DFT. While the computational complexity of the DFT is $O(N^2)$ for a set of N points, the FFT gets the same results with $O(N \cdot \log_2(N))$, which means a huge speed-up when N is high.

There is a special command to compute the FFT in Matlab. The FFT of a real signal is a complex signal in which each complex sample represents the magnitude and the phase of the corresponding frequency. In our case, the phase is not needed. The FFT magnitude is easily computed in Matlab:

```
fftMagnitude = abs(fft(signal));
```

Sliding window

In order to give a continuous estimation of the heart rate, the

FFT and the following two steps (peak detection and smoothing) are repeated every 0.5 seconds. This computation is always performed over a window containing the last 6 seconds of signal samples. Virtually, the window moves over the signal and this is why it is called "sliding" or "moving" window.

The 6-second length is not arbitrary. The window length directly affects frequency resolution and, thus, the accuracy of our estimation. The FFT of a signal sampled N times at a sampling frequency F_S is N bins long. All the bins together cover a bandwidth of F_S. So, the frequency difference between two consecutive bins is F_S / N. This is the frequency resolution (F_r). As the sampling frequency can be written as the number of window samples divided by the total time it took to sample them (the window time duration), we can say that:

$$F_r = \frac{F_S}{N} = \frac{\frac{N}{T_W}}{N} = \frac{1}{T_W}$$

Where:
F_r: frequency resolution
F_S: sampling frequency
N: number of window samples
T_W : window time duration

Therefore, the higher the window duration, the better the frequency resolution. The accuracy will be better, too, as it is half the resolution in this case.

However, increasing the window duration decreases the time accuracy. Think about the trivial case in which the whole signal length is picked as the window length. If a peak is detected in the FFT, it is impossible to tell when that tone started within the signal or how long it lasted. Whatever number we give will be a maximum of a window length away from the real value. Another problem of a long window is that it will force the user to wait for an equally long period to get a first reading after starting up the measurements.

In summary, with a 6-second window, we get a tolerable 6-second startup delay that gives a fair time accuracy of 6 seconds and a fair frequency accuracy of 5 bpm (half the FFT resolution). All in all, it looks like a good trade-off.

Why do we move the window in 0.5-second steps? Computing an estimate every 0.5 seconds does not improve

the time accuracy of the output, but it increases the time resolution of the reading. It produces more heart rate output samples per second that will be later smoothed to provide a more continuous and frequent reading. With this, we are incrementing the time resolution of the reading but not its accuracy, which stays limited by the time resolution of the FFT.

Leakage reduction

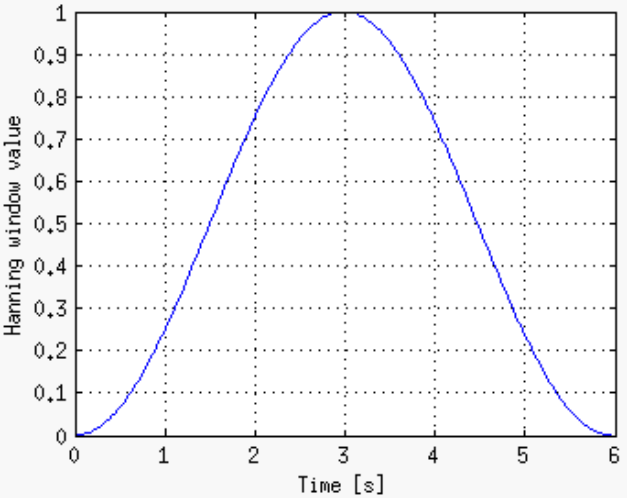


Fig. 4: Hann window for the 6-second sliding window

The DFT works ideally with infinite-time signals. A time-limited signal of length N is equivalent to multiplying its infinite-time counterpart by a rectangular signal of length N and amplitude 1. Frequency-wise, this results in convolving the infinite-time signal spectrum by the rectangular signal spectrum, producing leakage.

In order to reduce leakage, before computing the DFT, the input signal is multiplied by a function whose boundaries are zero. This forces the resulting boundary values to zero. The multiplying function is usually called "window". It must not be confused with the sliding window that we have talked about in the previous section. There are many window functions in the literature, each having their own virtues and disadvantages. I have particularly chosen the Hann window because it offers good resolution and good leakage rejection. Feel free to try others. There is a good comparative analysis of different window functions here.

In Matlab, the FFT magnitude for the sliding window with the Hann window can be computed with the following code. The result generated by the *hann* function is transposed because it is a column vector, while our signal is a row vector:

```
hannWindow = hann(size(slidingWindow, 2));
fftMagnitude = abs(fft(slidingWindow .* hannWindow'));
```

5. Peak detection

Once the FFT is computed for the current sliding window contents, magnitude peaks in the interest band are spotted thanks to the *findpeaks* function in Matlab. A sample is taken as a peak if it is either larger than its two neighbors or equal to infinity. Among the resulting peaks, the highest peak position is sought with the *max* function. Finally, it is translated to the corresponding frequency in the FFT vector:

```
% Translate the frequency range of interest to indices within
the FFT vector
rangeOfInterest = ((BPM_L:BPM_H) / 60) * (size(fftMagnitude,
2) / samplingFrequency) + 1;
% Find peaks in the range of interest
[peaksValues,                peakIndices]                =
findpeaks(fftMagnitude(rangeOfInterest));
% Find the highest peak
[maxPeakValue, maxPeakIndex] = max(peaksValues);
% Translate the peak index to an FFT vector index
bpmFreqIndex = rangeOfInterest(peakIndices(maxPeakIndex));
% Get the frequency in bpm that corresponds to the highest
peak
bpmPeak  =  (bpmFreqIndex  -  1)  *  (samplingFrequency  /
size(fftMagnitude, 2)) * 60;
```

6. Smoothing

At this stage, an approximate location for the most powerful tone in the frequency band has been found, but the possible outcomes are a discrete set in 10 bpm increments because of the frequency resolution produced by the 6-second window. We would like that the heart rate readings look more continuous, with 1 bpm frequency resolution instead. To achieve this, the signal window is correlated with a series of tones in phase and quadrature around the FFT peak in 1 bpm increments. The tones lie in the uncertainty interval around the peak caused by the FFT frequency resolution. The result of each signal-tone correlation is a complex number representing a phase-magnitude pair. The frequency that corresponds to the highest magnitude is taken as the smoothed heart rate:

$$c_k = \sum_{n=0}^{N-1} b_n \, e^{j\,2\pi\,n\,(\,f_p\,-\,0.5\,F_r\,+\,k\,F_r')\,/\,F_s} \qquad \mathbf{U} = \{\,0, \ldots, \left\lceil \frac{F_r}{F_r'} \right\rceil\}, k \in \mathbf{U}$$

$$\mathrm{HR} = f_p - 0.5\,F_r + F_r' \underset{k \in \mathbf{U}}{\arg\max} |c_k|$$

Where:

HR : Smoothed heart rate

b_k : k-th sample of brightness signal

N : Window length in samples

f_p : FFT peak frequency

F_r : FFT frequency resolution

F_r' : Smoothing frequency resolution

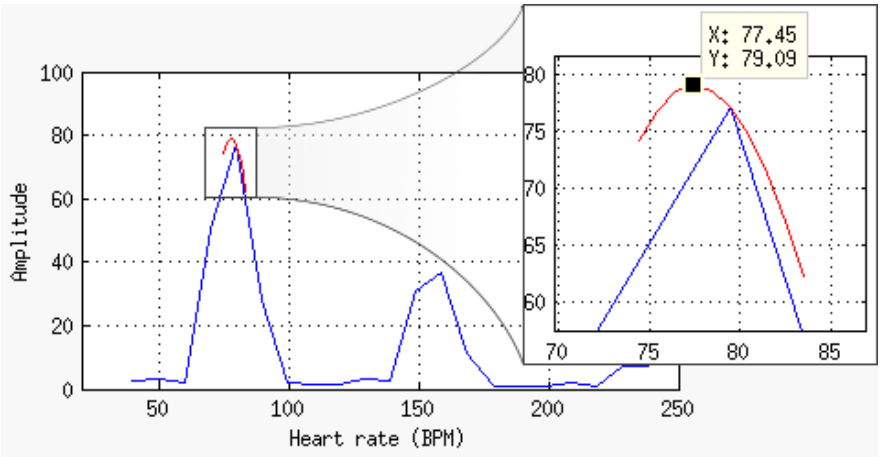


Fig. 5: Smoothing of an FFT peak. The DFT is computed on a zero-padded version of the signal in the window, but only within a range of FFT frequencies around the highest peak. The resulting DFT amplitudes form the red curve above the FFT, in blue. The frequency with the highest DFT amplitude, which is marked with the black square, is chosen as the current smoothed heart rate.

At first, I came up with this method intuitively, thinking of the correlation as a measure of similarity. My goal was to find the tone most similar to the signal in a frequency range around the FFT peak, regardless of the phase, by measuring the similarity (correlation) between the signal and a series of reference complex tones. In fact, it is what the FFT does, but using only orthonormal tones, whose period is an integer number of samples. Since the FFT frequencies are orthonormal, they constitute a base in which all signals can be expressed by linear combination. Correlating against other intermediate frequencies does not give more information because they themselves are formed by linearly combining the orthonormal ones. This is the reason why the result of this method is just smoothed FFT data, instead of higher accuracy extra information. Later, I realized that this method is equivalent to computing the FFT on a zero-padded version of the signal, which is a commonly used technique to smooth the FFT data, and picking only a frequency range. If the original 6-second signal in the window is zero-padded up to 60 seconds, the tone frequencies used in this method are found among the orthonormal frequencies of the 60-second FFT. Then, the method is equivalent to applying the DFT definition on the zero-padded window for certain

frequencies only.

The advantage of this "zero-padding partial DFT" over the "zero-padding FFT" is speed. The FFT computes the result as a whole and cannot be divided to get subresults in a continuous frequency range; therefore, its complexity stays at $O(N_Z \cdot \log_2(N_Z))$, where N_Z is the length of the zero-padded window. On the other hand, the complexity of the proposed method is $O(S \cdot N)$, where N is the window length and S is the number of test tones. So, the complexity reduction factor (CRF) is:

$$\left.\begin{array}{l} S = F_I / F_I' \\ F_I' = F_S / N_Z \\ F_I = F_S / N \end{array}\right] \Rightarrow S = N_Z / N \Rightarrow S \cdot N = N_Z$$

$$CRF = \frac{N_Z \cdot \log_2(N_Z)}{S \cdot N} = \frac{N_Z \cdot \log_2(N_Z)}{N_Z} = \log_2(N_Z) = \log_2 \frac{F_S}{F_I'}$$

Where:
CRF: Complexity Reduction Factor of the partial DFT method over the FFT on the zero-padded signal
F_S: Sampling frequency
F_I: Frequency resolution before smoothing
F_I': Desired frequency resolution after smoothing
N: Window length
N_Z: Zero-paded window length
S: Number of test tones

For a given sampling frequency, the higher the new frequency resolution, the more beneficial will be using the "zero-padding partial DFT" instead of the "zero-padding FFT". In our case, F_I' = 1/60 Hz and F_S = 24 Hz in the worst case. So, the CRF is approximately 10.5, which means that —roughly speaking— the time taken by the proposed method will be something in the order of a 10% of the time taken by the FFT on the zero-padded window.

Here is the Matlab code to implement the signal smoothing:

```
fftResolution = 1 / WINDOW_SECONDS;
lowFreq = bpmPeak / 60 - 0.5 * fftResolution;
smoothingResolution = SMOOTHING_RESOLUTION / 60;
testFreqs = round(fftResolution / smoothingResolution);
power = zeros(1, testFreqs);
freqs = (0:testFreqs - 1) * smoothingResolution + lowFreq;
for k = 1:testFreqs,
    re = 0; im = 0;
    for j = 0:(size(b, 2) - 1),
        phi = 2 * pi * freqs(h) * (j / samplingFreq);
```

```
        re = re + b(j+1) * cos(phi);
        im = im + b(j+1) * sin(phi);
    end
     % Since we only need to find the maximum, we can use
power instead of magnitude and skip sqrt()
    power(k) = re * re + im * im;
end
[maxPeakValue, maxPeakIndex] = max(power);
smoothedBpm = 60 * freqs(maxPeakIndex);
```

Actively increasing the SNR

A good Signal-to-Noise Ratio (SNR) is essential to accurately detect the heart rate signal. It can be improved by either reducing the noise or increasing the signal power. In our case, the noise affecting the measurement comes from three sources basically:

- *Image noise* : It originates in the camera sensor. By averaging all the pixels for the brightness calculation, we are filtering out a big part of its spatial component. The band-pass filter that we are applying to the brightness signal is also rejecting much of its time component. However, there will always be some level of this noise in the band of interest.
- *User behavior* : The pressure variations of the fingertip against the camera lens may show up in the signal. If the user has shaky hands, it is better that she uses a finger of the hand holding the phone.
- *Lighting changes* : Since the finger is lit from behind, any change in the light sources or in the scene reflecting that light towards the camera may introduce noise in our pass band. For instance, moving the phone in the air while recording the video sequence may introduce artifacts if the scene parts in the field of view have very different light intensities.

Whereas the noise sources are obviously difficult to control —especially if the system is operated by someone else—, we can increase the signal level by illuminating the finger with a stable light source. If the smartphone has a torch that can be touched with the finger being recorded, it will light the tissues and improve the signal levels. However, if the torch is too powerful, it will saturate the image sensor and nothing but a pure white image will be recorded. If this is the case, try dimming the torch with your phone controls or covering it with a piece of fabric.

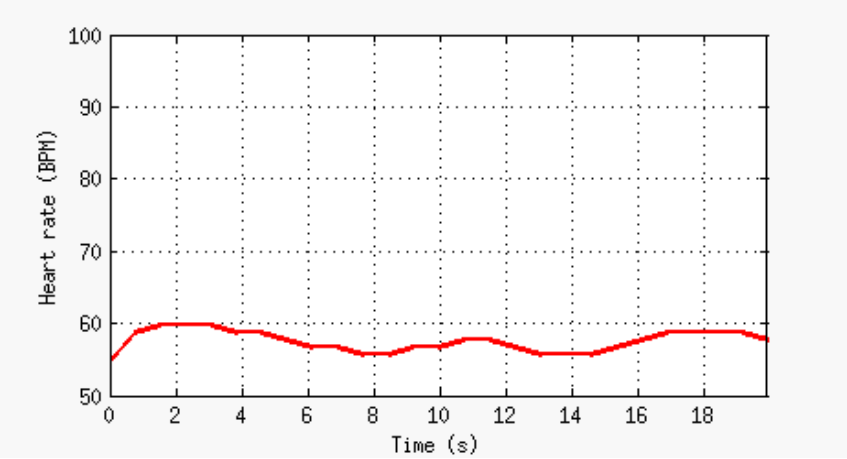
Results

In order to check the system behavior, heart beat measurements with an iPhone 4S camera were performed on

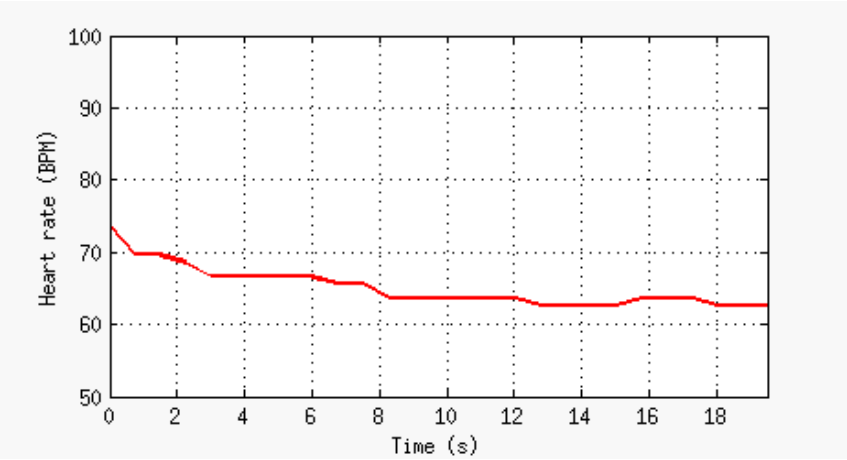
two individuals in two situations: at rest and after exercise (squats). Results are depicted in Fig. 6. Plots 6(a) and 6(c) correspond to a 34-year-old man who exercises regularly, whereas 6(b) and 6(d) belong to a woman of the same age who does not exercise. Besides having a lower resting rate, the active individual recovers faster than the sedentary one: >40 bpm vs <30 bpm decrements after the first minute, respectively.

Heart rates at rest were also estimated manually, by counting heart beats for one minute while videos were being recorded. The manual count results were 58 bpm for plot 6(a) and 67 bpm for plot 6(b), whereas the means computed from the measurements were 57.45 bpm for 6(a) and 65.09 bpm for 6(b).

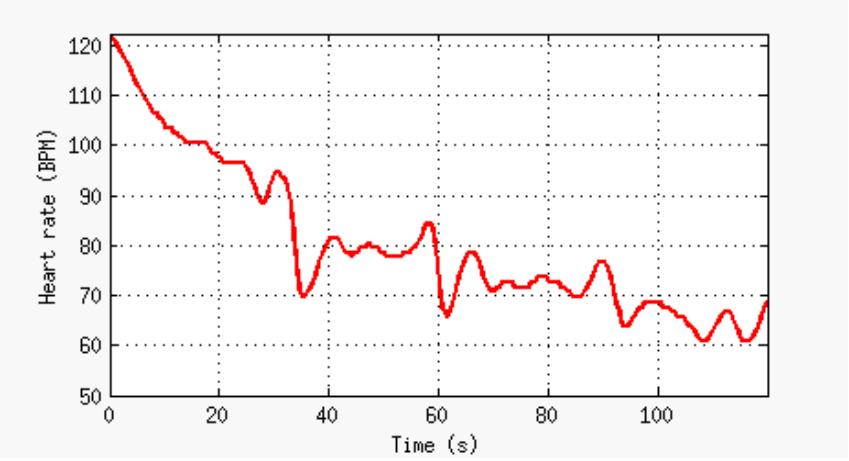
(a)



(b)



(c)



(d)

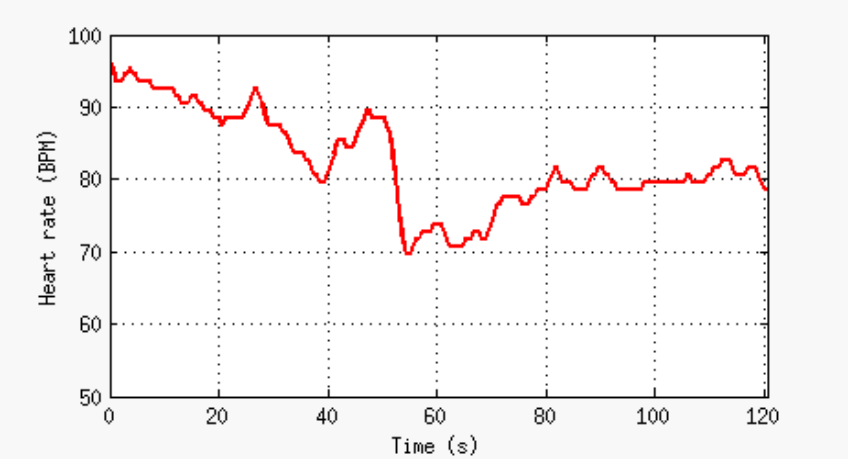


Fig. 6: Heart rate over time for two people at rest (a, b) and recovering after squats (c, d). The curves (a) and (c) belong to a person who exercises often, while (b) and (d) belong to a person who does not. Exercise in (c) was longer and more intense than in (d). Heart rates at rest were manually estimated by counting heart beats for 1 minute while the video was being recorded. The manual count results were 58 bpm for (a) and 67 bpm for (b).

The following video shows an animation in 4x real-time of the case in Fig. 6(c) —active 34-year-old man recovering after intense squats—. Figure captions for Figs. 5 and 6 are applicable to the upper and lower plots in the video, respectively:

Share this page:

Source code

- Heart rate estimation from a video of the fingertip

Related blog posts

- Getting started with Computer Vision on iPhone

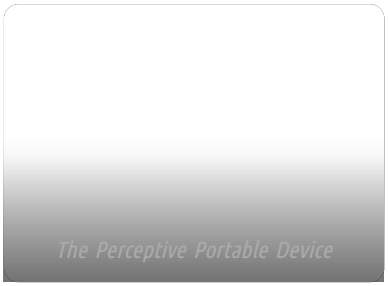
Technologies

iPhone Matlab Fast Fourier Transform

Topics

Computer Vision Tutorials Signal processing

Related projects



Jesses • 10 months ago

Hi Ignacio Mellado,

Thank you for the writeup, I'm working in my project, so it's very useful. But, i had a question: You calculate bpm(i) = (max_f_index-1) * (fps / size(y, 2)) * 60, i don't understand why not bpm(i) = (max_f_index) * (fps / size(y, 2)) * 60 , Why we must subtract 1.

And 1 question: Assume il:ih = 5:25, and fps = 25, size(y, 2) = 151, so bmp(i) only be:{4*25/151*60 = 39.7, 49.7 (max_f_index = 6), 59.6, 69.5, 79.47...} While if you calculate heart rate, assume there are errors, it should be close to a certain value, for example bmp = {67, 65, 68, 66, 70,...}, its adjacent.

I'm looking forward to seeing your reply, i really appreciate that. Thanks

1 • Reply • Share ›

Ignacio Mellado Mod → Jesses • 9 months ago

Thanks for reading, **@Jesses** . Hope my answer helps. The lowest index value for a vector in Matlab is 1. Since max_f_index is such an index, you need to subtract 1 in order to map the lowest index value to 0 bpm.

Regarding your second question, the FFT resolution is limited by the number of samples and the frame rate. With the numbers you mention, it's 9.93 bpm, which is the difference between the consecutive values of your first series. If you'd like something more fine-grained, like your second series, you'd need to use a larger window. Otherwise, you can just smooth the values, as discussed in the blog post. This is later done in the code in bpm_smoothed.

• Reply • Share ›

Jesses → Ignacio Mellado • 9 months ago

Thanks for your answers, @Ignacio Mellado . I really appreciate about that. My first question, according to your reply, i understood.

To be honest, i'm working in my android project. As you know, the camera's phone gets frames from surface view and frames per second is limited by hardware.For example, on my phone i only get 7 frames per second, and it's just enough to proceed in accordance with the formula Nyquist sampling frequency. And you know, after 6 seconds we must display user's heart rate on phone screen. So, i think i can not get a larger window as you mentioned (because users have to wait so long). Can you give me some advices about this problem?

Thanks you and have good day.

Reply • Share ›

Ignacio Mellado Mod ➚ Jesses
• 9 months ago

I would go for a time-domain method, then. Just observe the signal over time, spot the peaks and find the average time between peaks. Good luck with your project!

Reply • Share ›

Jesses ➚ Ignacio Mellado • 9 months ago

Hi, thank you. i'll try.
Ak, i have a problem: When i try to capture video on my phone and I use matlab to find frames per second of this video, i got 250 frames per 10 seconds, it means Fps = 25. But in android app which i'm working , i only get 70 frames per 10 seconds, just 7 fps.
Do you know why?

Reply • Share ›

Ignacio Mellado Mod ➚ Jesses
• 9 months ago

Maybe you have to tune the camera parameters to get 25 fps, or you have some slow operation in your frame capture loop.

Reply • Share ›

Jesses ➚ Ignacio Mellado • 9 months ago

Hi, yes, i fixed it. Now, i can get 25 fps.

Ak, i found out some documents in internet, and i saw some authors use ICA (independent component analysis) algorithms to separate Red, Green, Blue values to orginal values. Can i apply ICA algorithm before i use Band-pass filter? Can it reduce noise when i do like that?
Thank you very much.

Reply • Share ›

Ignacio Mellado Mod ➚ Jesses
• 9 months ago

I'm not very savvy on ICA... Do you mean using ICA to identify other information sources besides the blood flow and discard them from the signal? I never tried that, but it sounds good. It could help getting rid of in-band noise. I would appreciate that you shared your results over here when you try

it. Good luck!

• Reply • Share ›

Jesses ➚ Ignacio Mellado • 9 months ago

Hi,
I read this documents:
http://dspace.mit.edu/handle/1...

I want to use ICA to get a original red values.
You can read the document and give me some comments.
Thank **@Ignacio Mellado**

• Reply • Share ›

Ignacio Mellado Mod ➚ Jesses
• 9 months ago

Very good approach and paper. Thank you for sharing, @Jesses . Little to add on my side, except that both face tracking and heart rate estimation could improve with a Kalman filter. In short periods of time, they both can be modelled as a linear process and, thus, a KF would provide estimates even when there is no direct observation.

On the other hand, they use a threshold of 12 bpm to reject noisy observations. A threshold over the Mahalannobis distance could be applied instead. For this, you would model the bpm signal as a Gaussian distribution and reject those values that are too many standard deviations away from the mean. It is like making their threshold adaptable to the statistical properties of the signal.

• Reply • Share ›

Mohamed S. Hamed • 4 months ago

have any idea about implementing this with using face rec

• Reply • Share ›

Ignacio Mellado Mod ➚ Mohamed S. Hamed
• 4 months ago

Yes, you can use a similar algorithm on a piece of skin from the user's face. Just detect/track the face and always feed the same piece to the algorithm. Please share your experience over here if you try it.

• Reply • Share ›

Steve Schofield • 4 months ago

This is a fantastic piece of research, thanks again for sharing it

This is a fantastic piece of research, thanks again for sharing it. Do you have any recommendations for porting the matlab code into objective C / swift for processing in the app?

^ | v • Reply • Share ›

Ignacio Mellado Mod ➚ Steve Schofield • 4 months ago

Thanks a lot, Steve. I would avoid custom implementations of FFT, correlation and windowing in favor of Apple's vDSP optimized functions. On the other hand, I would play with the window length to give appropriate resolution vs reading delay, depending on user's needs. Please let me know about your project if you go on with it. Happy coding!

^ | v • Reply • Share ›

Roland Harrison • 6 months ago

Putting this research in the public domain is fantastic, did you have anything to do with the apps on the market or was this research purely to understand how it could be done?

^ | v • Reply • Share ›

Ignacio Mellado Mod ➚ Roland Harrison • 6 months ago

Thank you for your appreciation, **@Roland Harrison** . I started this for personal fulfillment, nothing to do with those apps, I promise :)

^ | v • Reply • Share ›

Edward Guo • 8 months ago

Hello Ignacio, I'm a college student and have started my Master study a few month ago, I just decide to focus on the field of Signal Processing. Your writeup is very helpful!! Thank you!

^ | v • Reply • Share ›

Ignacio Mellado Mod ➚ Edward Guo • 8 months ago

Thank you for your kind words, **@Edward Guo** . I'll do my best to keep posting. Best of luck with your Master!

^ | v • Reply • Share ›

Ryan • 8 months ago

Hello. I am using a variation of your code that is everything up to the leakage part and excluding the multiple 6 second segments. Another difference is that rather than using the entire video frame, I am using a cropped portion in the x,y dimensions (not time).

For some strange reason, the signal a.k.a brightness(:), clearly has a pattern of frequency components in the background portion of my image (not on skin). This is both under natural (outside) lighting and fluorescent lighting. I cannot fathom what this could be and was wondering if you came across this? I have used two different cameras and this frequency "pooling" in the background of my image still occurs. I am afraid it is corrupting my heart rate

of my image still occurs. I am afraid it is corrupting my heart rate measurement.

I have attached the plots for camera 1 under natural light and camera 2 under fluorescent lighting.

If you could help me in any way I would greatly appreciate the

see more

• Reply • Share ›

Ignacio Mellado Mod Ryan • 8 months ago

Hi **@Ryan** . The code looks fine and I cannot see any clear error. Just to check that I understand what you mean:

- The plots you attached belong to an image region with no skin in it
- You found those peaks within the heart rate frequency band regardless of the specific non-skin image region that you pick

If this is the case, it might be ambient noise, as you say. Fluorescent lights could have produced those undesired

<< Back to all blog posts

"If you want to walk fast, walk alone. If you want to walk far, walk together."

African proverb