Signal Processing in Bioengineering
Instituto Superior Técnico, Lisboa
Lab 3 – QRS complex and Resampling
March 22, 2021

João Saraiva 86449 | João Fernandes 87223 | Iara Ravagni 101090

All code was developed in a Python equivalent manner to MATLAB and is available at `https://github.com/jomy-kk/PDSBLabs` in `lab03` directory.

Similar to the previous lab work we may read and write EDF/EDF+ files using the `pyedflib` package. The `lab03/py_students/get_edf.py` provides a translation of the faculty file `getedf.m`. To do signal processing we may use the MATLAB equivalent `scipy.signal` package or the `biosppy` package that was designed with the particular purpose of processing physiological signal.

1 QRS complex in ECG

Using the `get_edf` function we can extract directly from `data/plm3_r_ECG_PPG_1h.edf` the signals of channels ECG1-ECG2 and PLETH and their respective headers, which correspond to an acquired ECG signal and a PPG signal. A plot for each in the interval [6, 306] second (s) is provided in Figure 1. Since both signals came vertically mirrored (i.e. inverted amplitudes), their data array was multiplied by -1 in order to undo that.

The procedure `scipy.signal.find_peaks` was used on the ECG and the PPG signal to find the timepoints of the R-peaks and the P-peaks, respectively. To evaluate their accuracy manually, four time envelopes were selected as metrics to look if the peaks detection corresponded to true positives (TP). For the ECG, those are – Envelope 1: [30, 40]s, Envelope 2: [105, 125]s, Envelope 3: [240, 260]s and Envelope 4: [3200, 3220]s. For the PPG, those are – Envelope 1: [395, 420]s, Envelope 2: [545, 575]s, Envelope 3: [2620, 2640]s and Envelope 4 is the same as the ECG's.

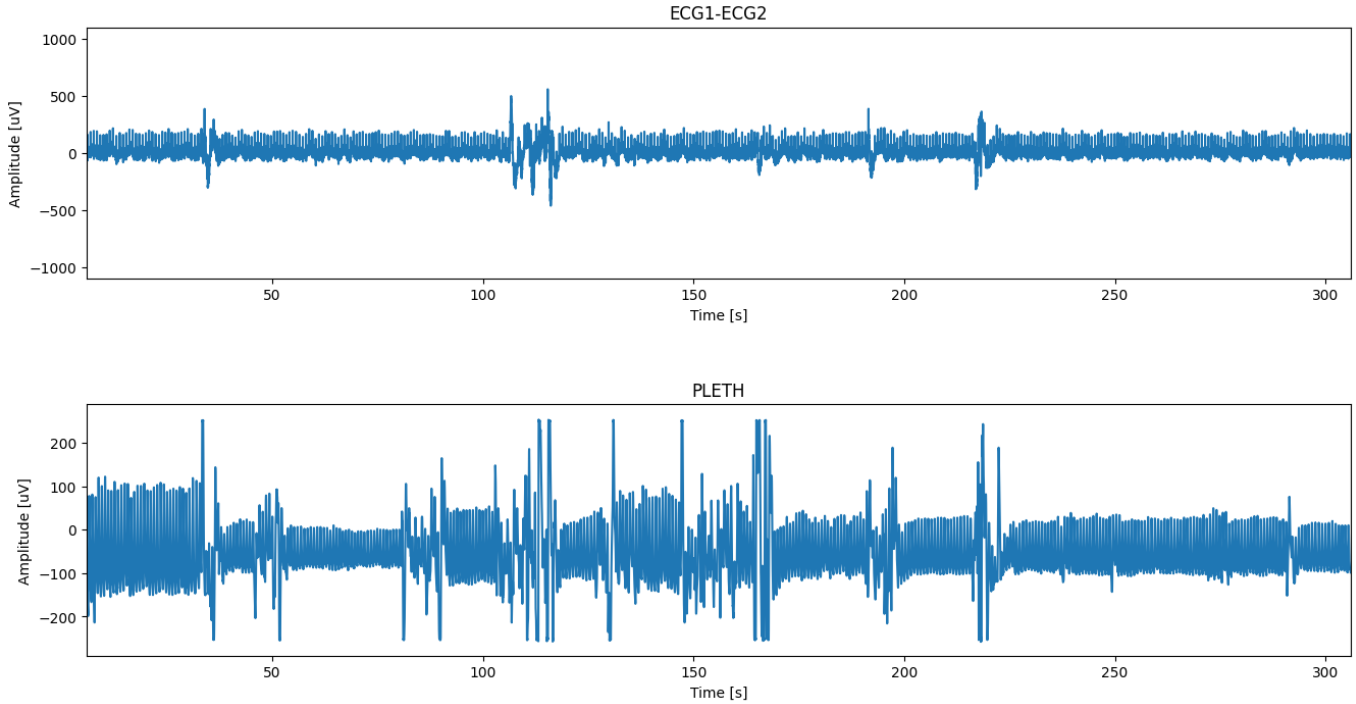

Figure 1: Channels ECG1-EC2 and PLETH of `data/plm3_r_ECG_PPG_1h.edf` from 6 to 306 s.

Let us start evaluating the `find_peaks` accuracy on the ECG – Figure 2. The signal was pre-filtered prior to application of the algorithm (more on that later). The parameters given to the algorithm were:

- prominence = 120
- distance = 150 (to allow a minimum distance between the QRS complexes)
- maximum width = 280 (to not count flat peaks as R-peaks)

We can notice two false positives (FP) in envelope 2 and three FPs in envelope 4. All other peaks in these envelopes were correctly pointed. It is important to mention that the prominence parameter was decreased to 50 just for the interval [2100:] s; otherwise only two peaks would be detected on envelope 4 and its neighborhood. But we had to do it manually. This is an indicator that adaptive parameters are important in this task (more on that later).

The R-R peak interval (RRI) can be simply computed by `numpy.diff` applied on the detected R-peak timepoints. With these found timepoints a mean of 0.689 s for the RRI and a standard deviation of 3.684 s were computed. This is depicted in Figure 3. As one can notice, there is too much variance between these intervals. But we will see later that there are more suitable algorithms for this task.

About pre-filtering the signal, a major difference can be seen if we apply the algorithm to the raw ECG signal. If we would do that, a mean RRI of 1.145 s would be found, which is highly unlikely. So, we used an already implemented procedure to filter the signal that can be found in `biosppy.ecg.ecg`. The ECG signal was filtered by a FIR bandpass filter of order 153 from 3 Hz to 45 Hz, allowing for two main improvements: (i) the removal of power line noise at 50 Hz, and (ii) the removal of respiration and baseline wander artifacts at approximately 0.05 - 0.1 Hz.

Regarding adaptive thresholds, there are some interesting different strategies to do this task. Let us now evaluate other peak finding algorithms – these ones engineered to specifically find R-peaks:

- **Christov segmenter** [1]
  Removes power line noise and electromyogram noise using two different moving-average (MA) filters and uses an adaptive threshold, combining a steep-slope threshold, an integrating threshold, and a beat expectation threshold. No parameters.

- **Engzee segmenter** [5]
  Differentiates the signal and applies a low-pass filter to the result. It also uses an adaptive threshold that when surpassed, it searches to see if the signal keeps lower than the threshold for a defined period of time *to the right* of that timepoint.
  Parameters:
  `threshold=0.48`

- **Hamilton segmenter** [4]
  Forces a distance of 200 ms for all large peaks and a distance of 360 ms for all confirmed R-peaks. It works with the time elapsed since the last detection and the average RRI of the last 8 detected R-peaks. Here the detection threshold is a function of the average noise and the average QRS peak values of the last 8 detected R-peaks.
  No parameters required.

- **Gamboa segmenter** [3]
  Normalizes and derivates the signal. It also uses the time elapsed since the last detection to confirm an R-peak, but here the threshold for detection is fixed.
  Parameters:
  `tolerance=0.01`

- **Slope Sum Function (SSF) segmenter** [6]
  Enhances the upslope of the signal, making the detection easier, derivates it and applies a weighted MA function. It also uses the time elapsed since the last detection to confirm an R-peak. The threshold is also fixed.
  Parameters (best compromise found):
  `threshold=50, before=0.5, after=0.7`

- **Elgendi segmenter** [2]
  Uses two MAs to segment the signal into blocks containing potential QRS complexes, the first with a window of 120 ms to match the approximate duration of a QRS complex, and the second with a wider window of 600 ms to match the approximate duration of a complete heartbeat. Sections where the amplitude of the first MA is higher than that of the second are marked and detections which follow the previous one by less than 300 ms are removed.
  No parameters required.

Each of these algorithms filters the signal in a different way. An already established implementation of these QRS segmenters can be found in the package `biosppy.ecg` released by Instituto de Telecomunicações and Elgendi's is available here. Following the same methodology as before, Table 1 shows the mean RRI and corresponding standard deviation of each algorithm, as well as a *pseudo*-accuracy of each, calculated by the fraction of true positives (TP) and the total positives evaluated only on the four mentioned envelopes. Figures 4, 6, 8, 10, 12 and 14 depict the detected R-peaks for each algorithm. And Figures 5, 7, 9, 11, 13 and 15 plot the computed RRI of each algorithm.

From Table 1 we can notice that `scipy.signal.find_peaks` shows the highest *accuracy* in detecting R-peaks, but we should remember there were given two different prominences, 120 and 50, to the first and second part of the signal, respectively. Also, Elgendi's and Hamilton's algorithms performed similarly good and their mean RRI is the same, $\approx$ 737 ms, which is a valid healthy RRI. Actually, in terms of consistency, these two showed the lowest variances in the computed RRI. So, we will acknowledge Elgendi's and Hamilton's as the best overall performers for this particular signal.

It should also be noted that besides Christov's algorithm performing poorly in terms of accuracy and RRI variance, it also presents the highest time complexity by far – $O(n \cdot p^2 \cdot log(p))$, where n is the number of samples and p is the number of peaks detected.

Moving on to the PPG signal, we tried to find the P-peaks using also `scipy.signal.find_peaks` and to find their onsets using `biosppy.bvp.bvp`, which uses the original Zong's method [6], which works on the same approach as the aforementioned SSF algorithm, but for PPG signals. In both cases, the PPG signal was filtered with a bandpass Butterworth filter from 1 Hz to 8 Hz. We ran `find_peaks` with the following parameters:

- prominence = 35
- distance = 64 (to allow a minimum distance between peaks; $\approx$ 0.5 second.)

Figure 16 shows the P-peaks detected by `scipy.signal.find_peaks` for the four defined PPG envelopes, yielding a mean P-P-interval (PPI) of 765 ms, with a standard deviation of 99 ms (Figure 17). As for the `biosppy.bvp.bvp` algorithm, it yields a similar mean PPI of 774 ms, with a standard deviation of 170 ms. Figure 18 shows the onsets of the P-peaks detected by the algorithm and Figure 19 plots the *instantaneous* PPI. This goes in hand with the previous mean RRI found in the ECG signal.

The mean squared error (MSE) of the `find_peaks`'s founded PPI and the Hamilton's founded RRI is 0.0346 [s^2] (0.186 [s]). With the Elgendi's founded RRI, the MSE is 0.0268 [s^2] (0.164 [s]). This allows us to conclude that actually the Elgendi's founded RRI in the ECG signal is less distant to the PPI founded in the PPG signal, but only for a little difference. The instantaneous heart rate derived from the PPG is plotted in Figure 20 with a computed mean of 81 bpm.

Table 1: Accuracy of the different R-peak detection algorithms, based solely on data from the four envelopes. The RRI mean and standard deviation given the detected peaks of each algorithm are also shown.

Algorithm	R-peak TP	R-peak FP	Accuracy	RRI Mean	RRI Std
find_peaks	89	5	0.946	0.689	3.684
chirstov	89	64	0.582	0.346	1.238
engzee	77	21	0.740	0.661	0.243
hamilton	83	11	0.883	0.738	0.163
gamboa	53	95	0.358	0.488	0.226
ssf	80	26	0.755	0.647	0.369
elgendi	86	5	0.945	0.7365	0.129

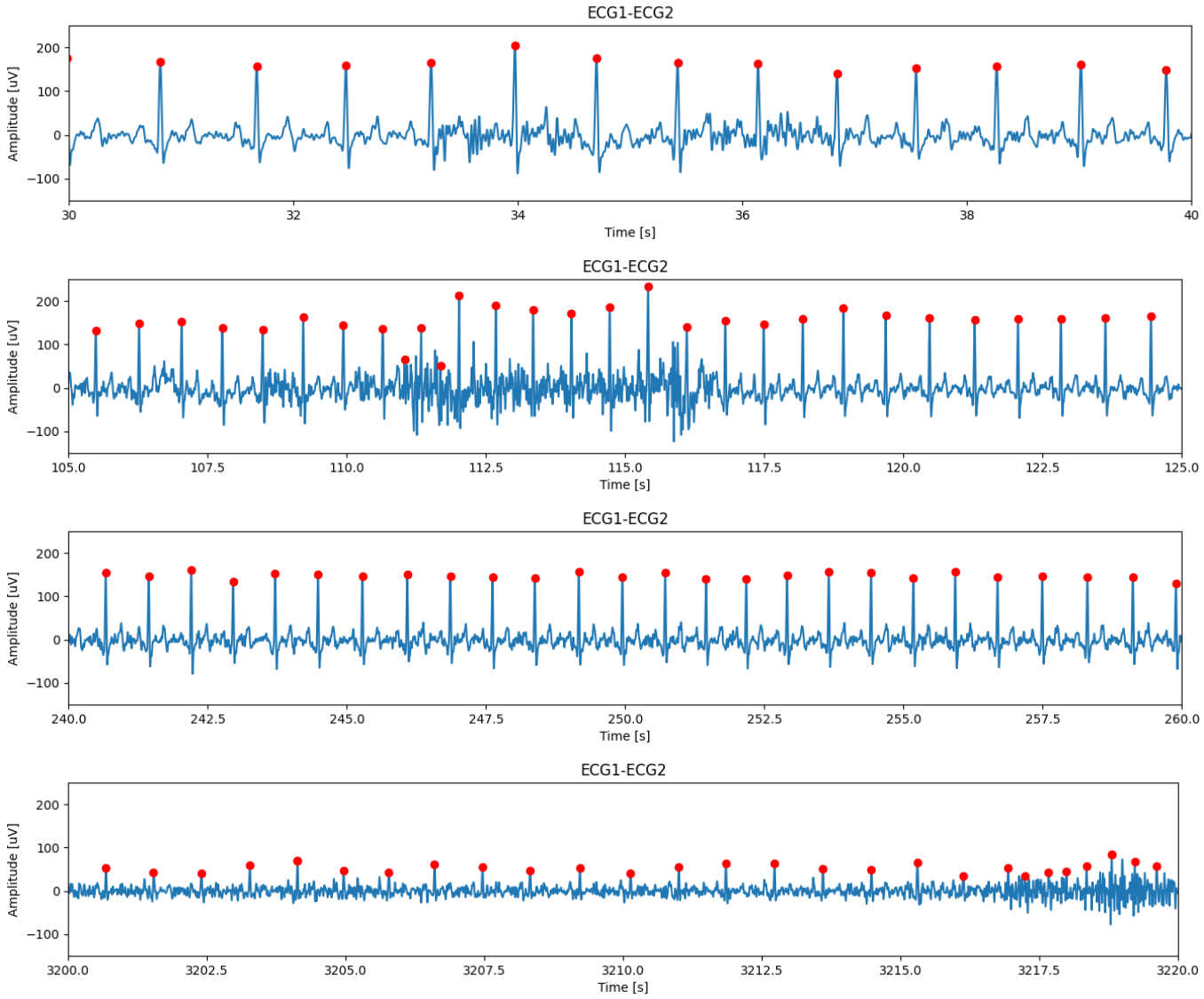

Figure 2: Results of the `scipy.signal.find_peaks` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

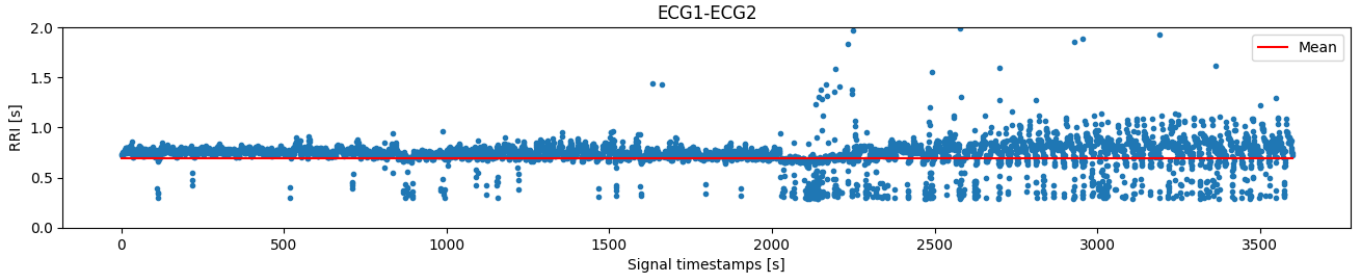

Figure 3: RRI of the filtered ECG signal (blue) based on the R-peaks found by `scipy.signal.find_peaks` algorithm. The mean is highlighted by the red line.

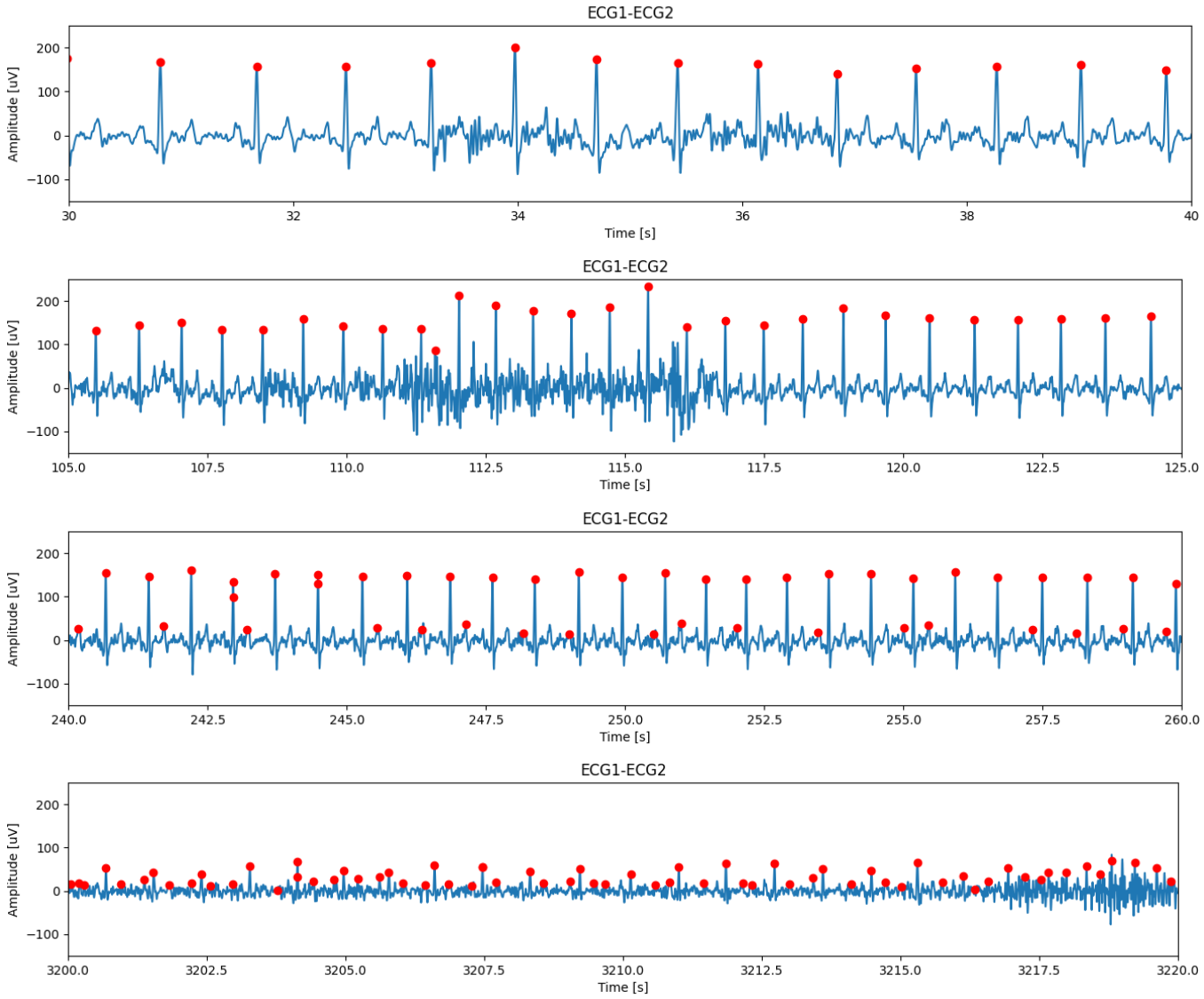

Figure 4: Results of the `biosppy.ecg.christov_segmenter` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

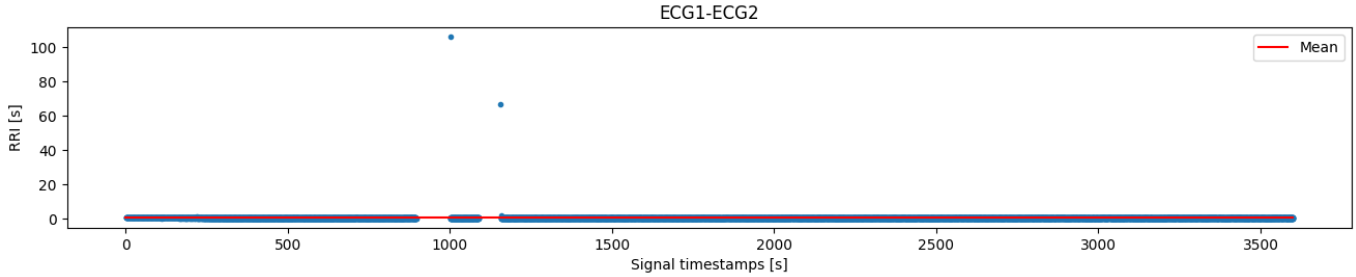

Figure 5: RRI of the filtered ECG signal (blue) based on the R-peaks found by `biosppy.ecg.christov_segmenter` algorithm. The mean is highlighted by the red line.

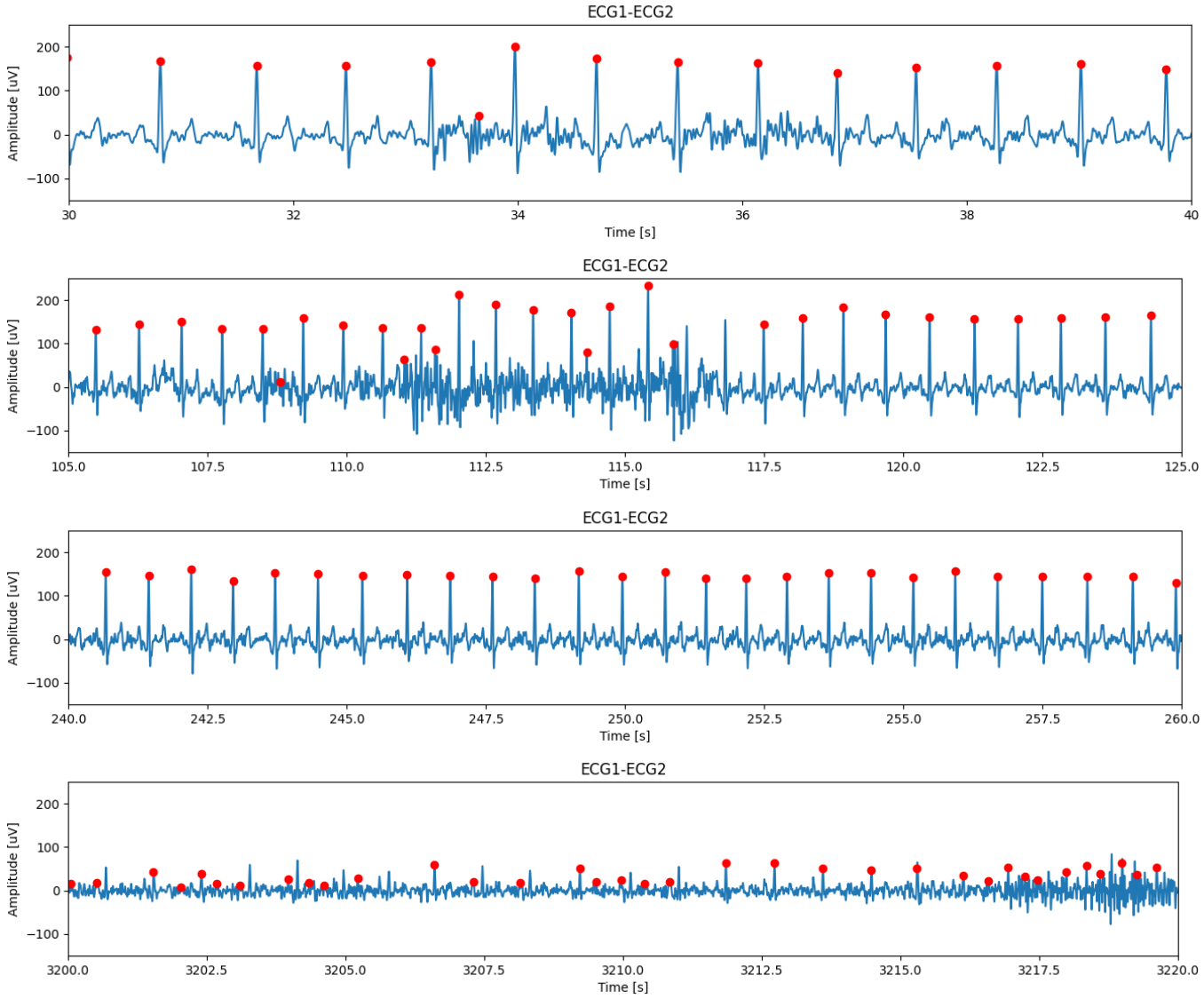

Figure 6: Results of the `biosppy.ecg.engzee_segmenter` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

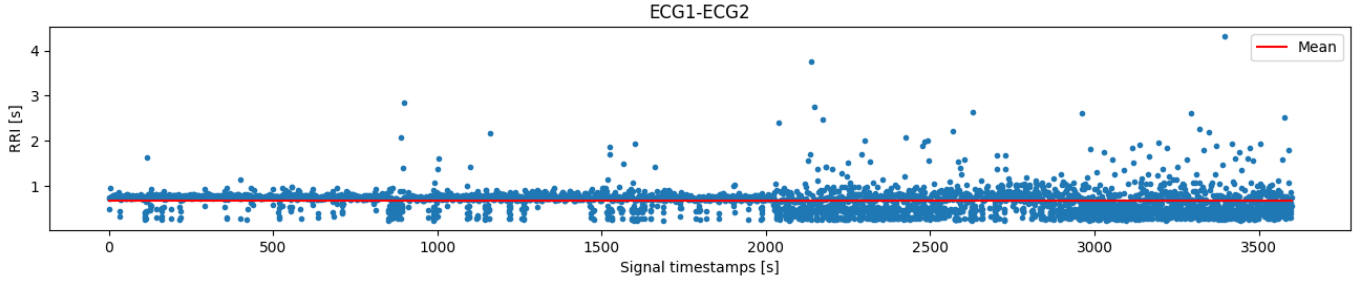

Figure 7: RRI of the filtered ECG signal (blue) based on the R-peaks found by `biosppy.ecg.engzee_segmenter` algorithm. The mean is highlighted by the red line.

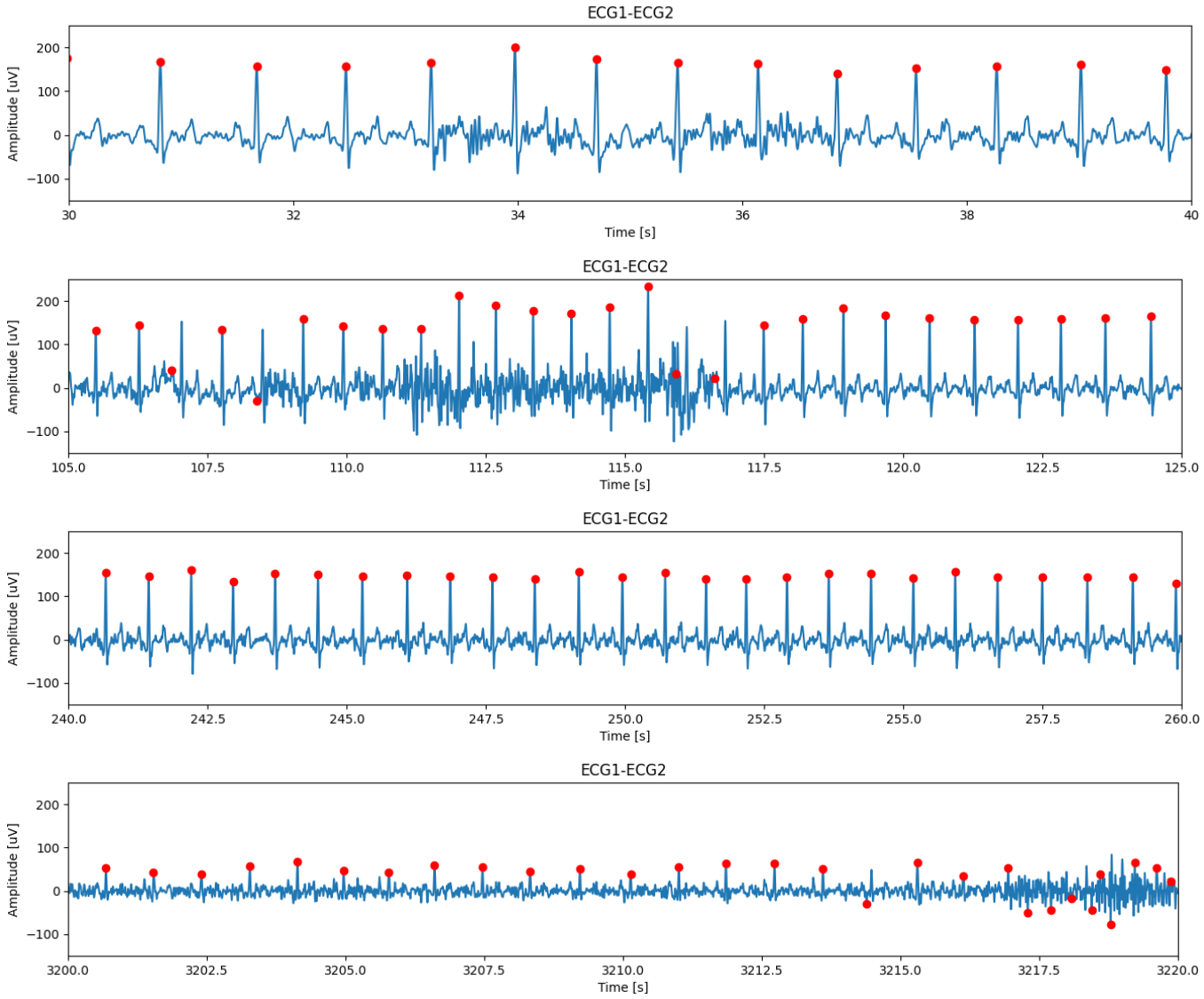

Figure 8: Results of the `biosppy.ecg.hamilton_segmenter` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

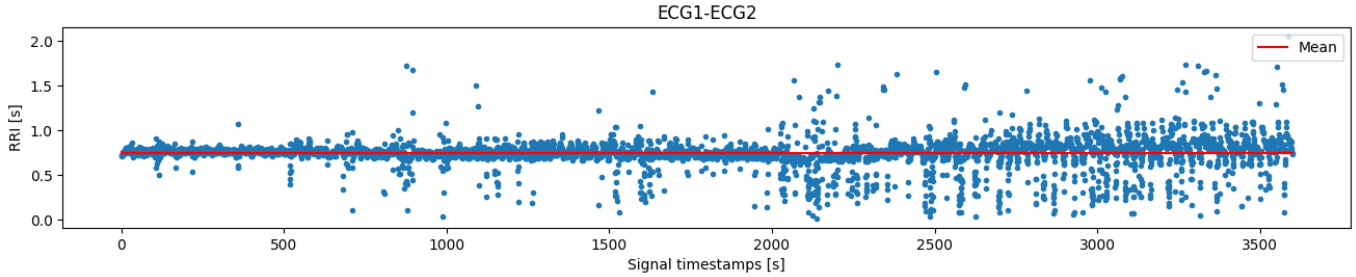

Figure 9: RRI of the filtered ECG signal (blue) based on the R-peaks found by `biosppy.ecg.hamilton_segmenter` algorithm. The mean is highlighted by the red line.

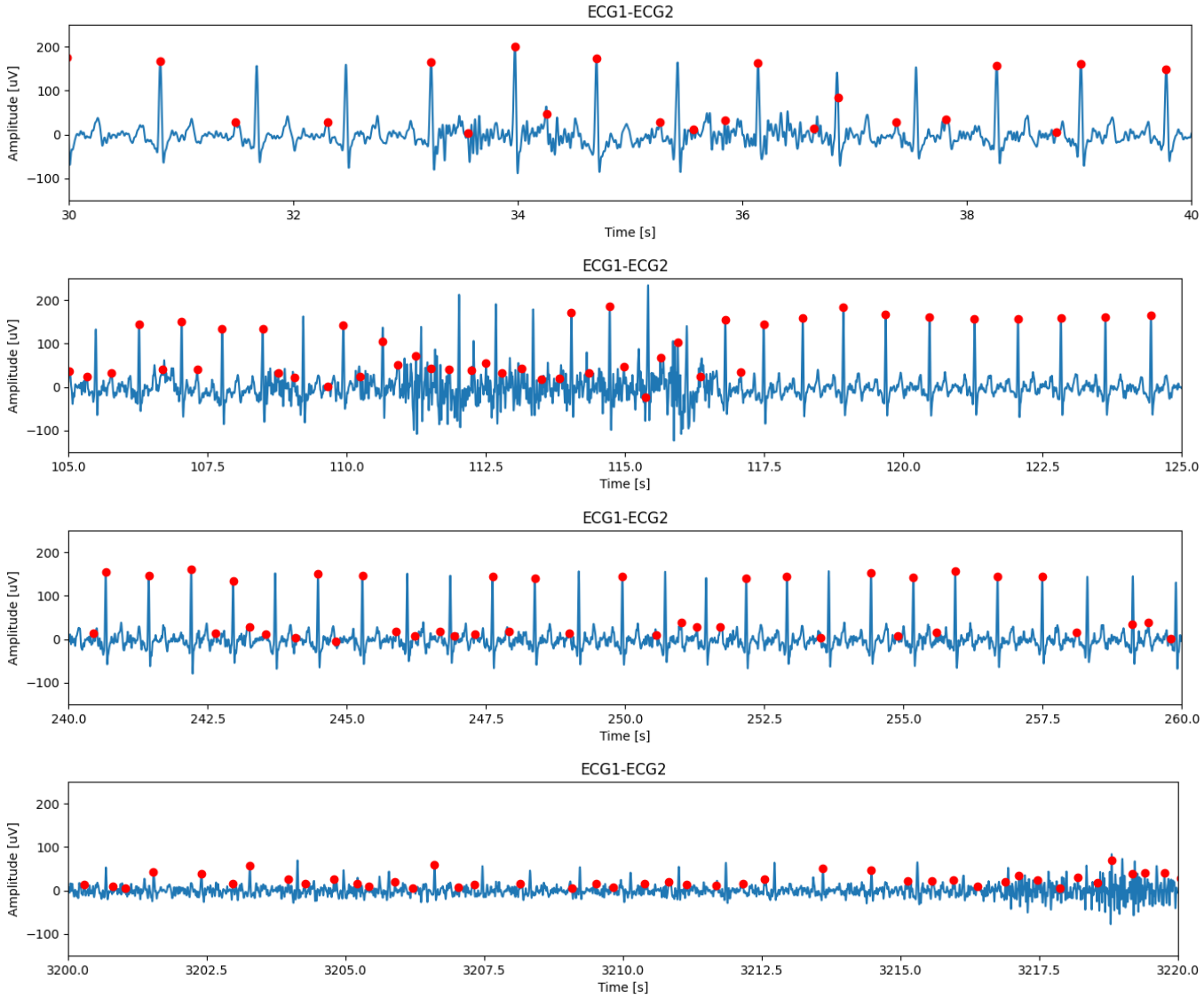

Figure 10: Results of the `biosppy.ecg.gamboa_segmenter` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

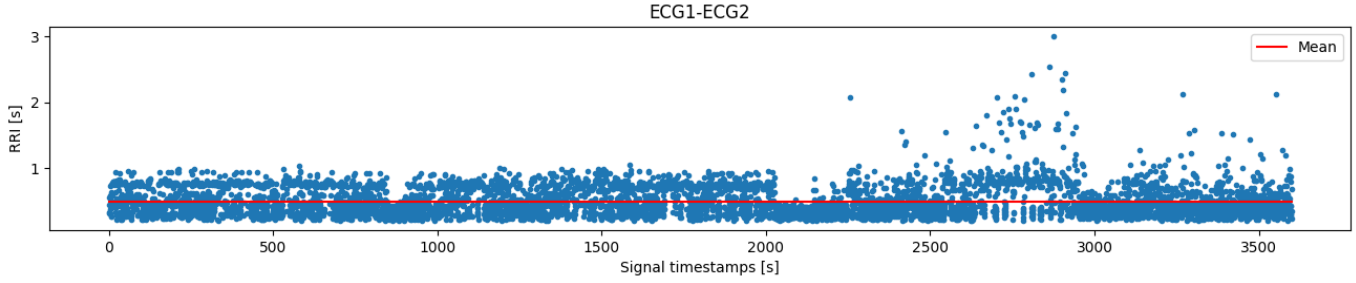

Figure 11: RRI of the filtered ECG signal (blue) based on the R-peaks found by `biosppy.ecg.gamboa_segmenter` algorithm. The mean is highlighted by the red line.

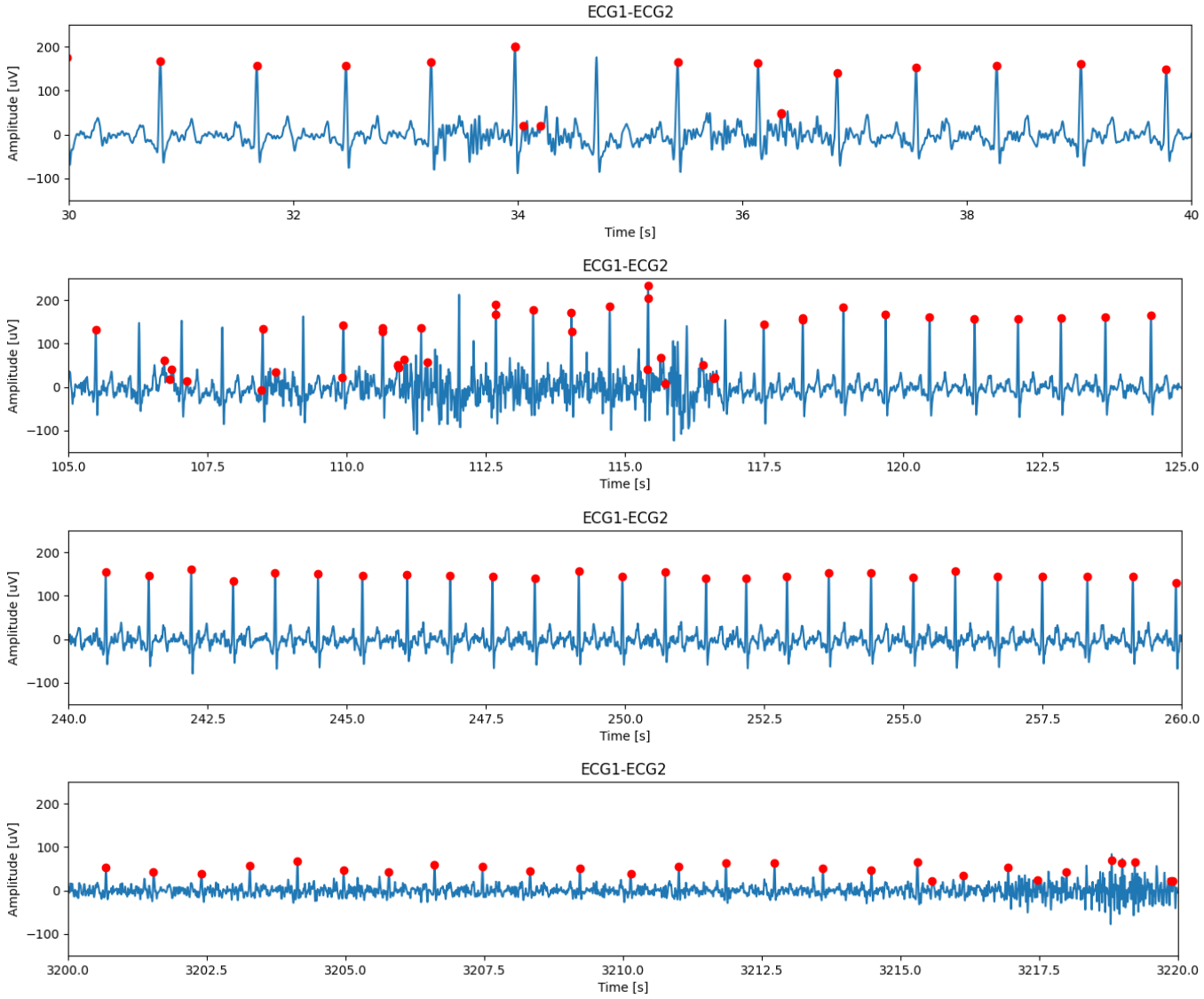

Figure 12: Results of the `biosppy.ecg.ssf_segmenter` algorithm on finding the R-peaks (red) of the signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

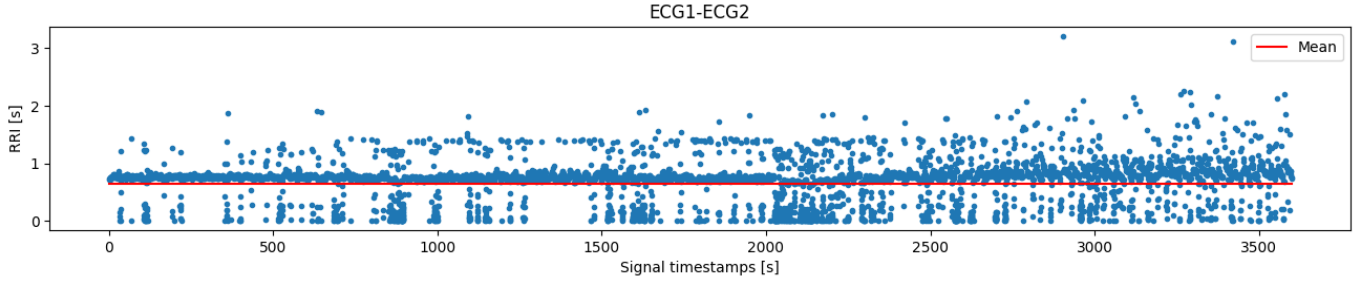

Figure 13: RRI of the filtered ECG signal (blue) based on the R-peaks found by `biosppy.ecg.ssf_segmenter` algorithm. The mean is highlighted by the red line.

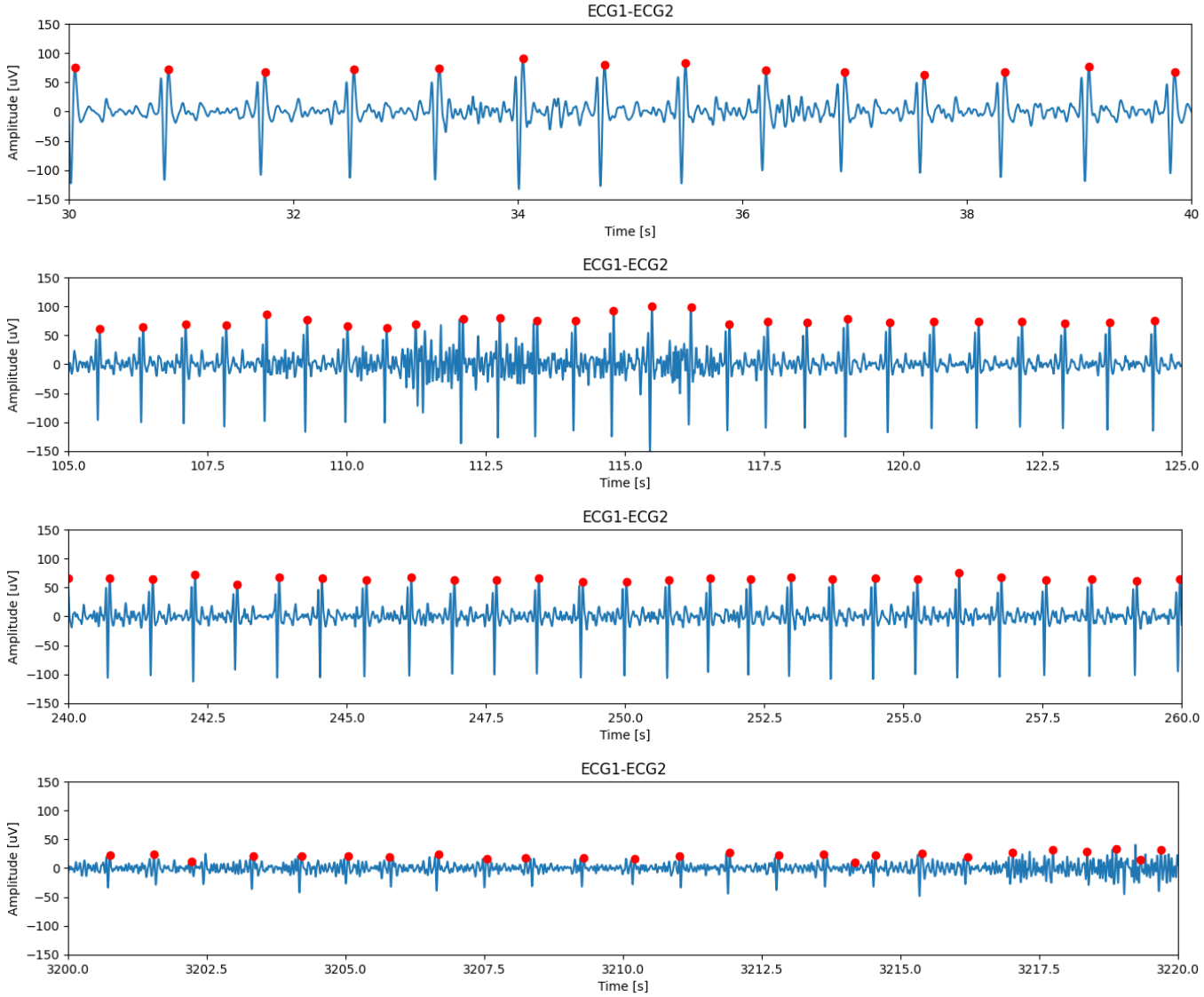

Figure 14: Results of the `py-ecg-detectors.ecgdetectors.elgendi` algorithm on finding the R-peaks (red) of the filtered ECG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

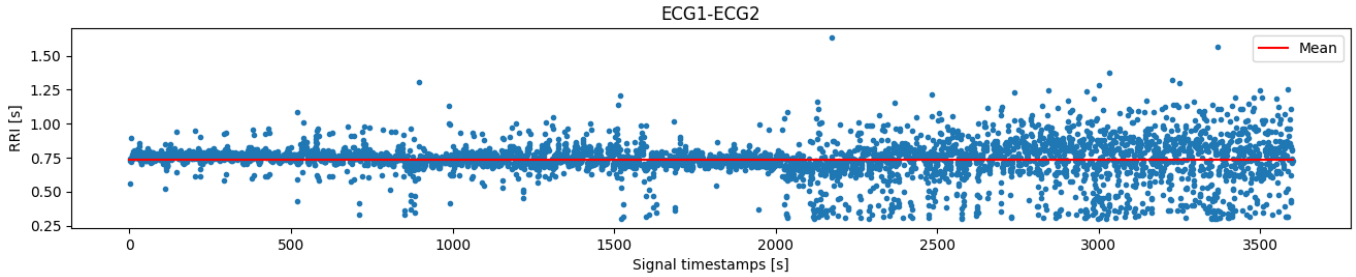

Figure 15: RRI of the filtered ECG signal (blue) based on the R-peaks found by `py-ecg-detectors.ecgdetectors.elgendi` algorithm. The mean is highlighted by the red line.

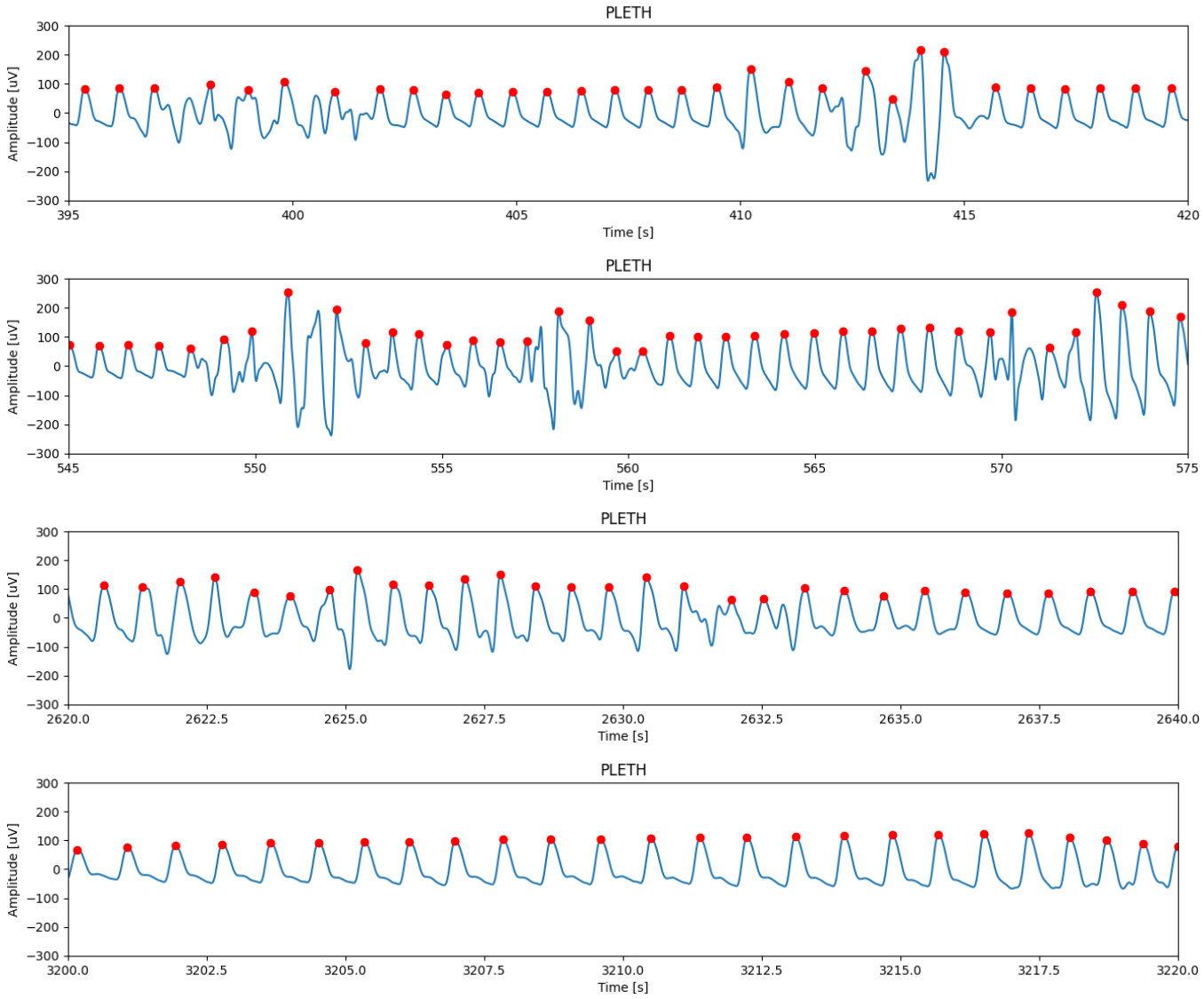

Figure 16: Results of the `scipy.signal.find_peaks` algorithm on finding the P-peaks (red) of the filtered PPG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

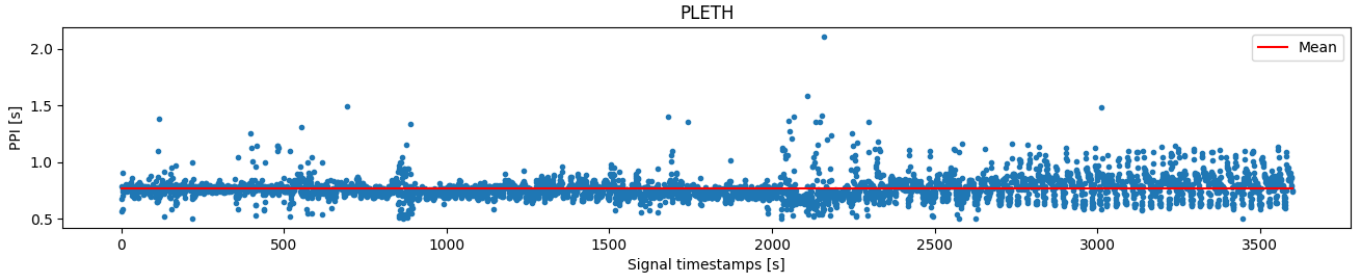

Figure 17: PPI of the filtered PPG signal (blue) based on the P-peaks found by `scipy.signal.find_peaks` algorithm. The mean is highlighted by the red line.

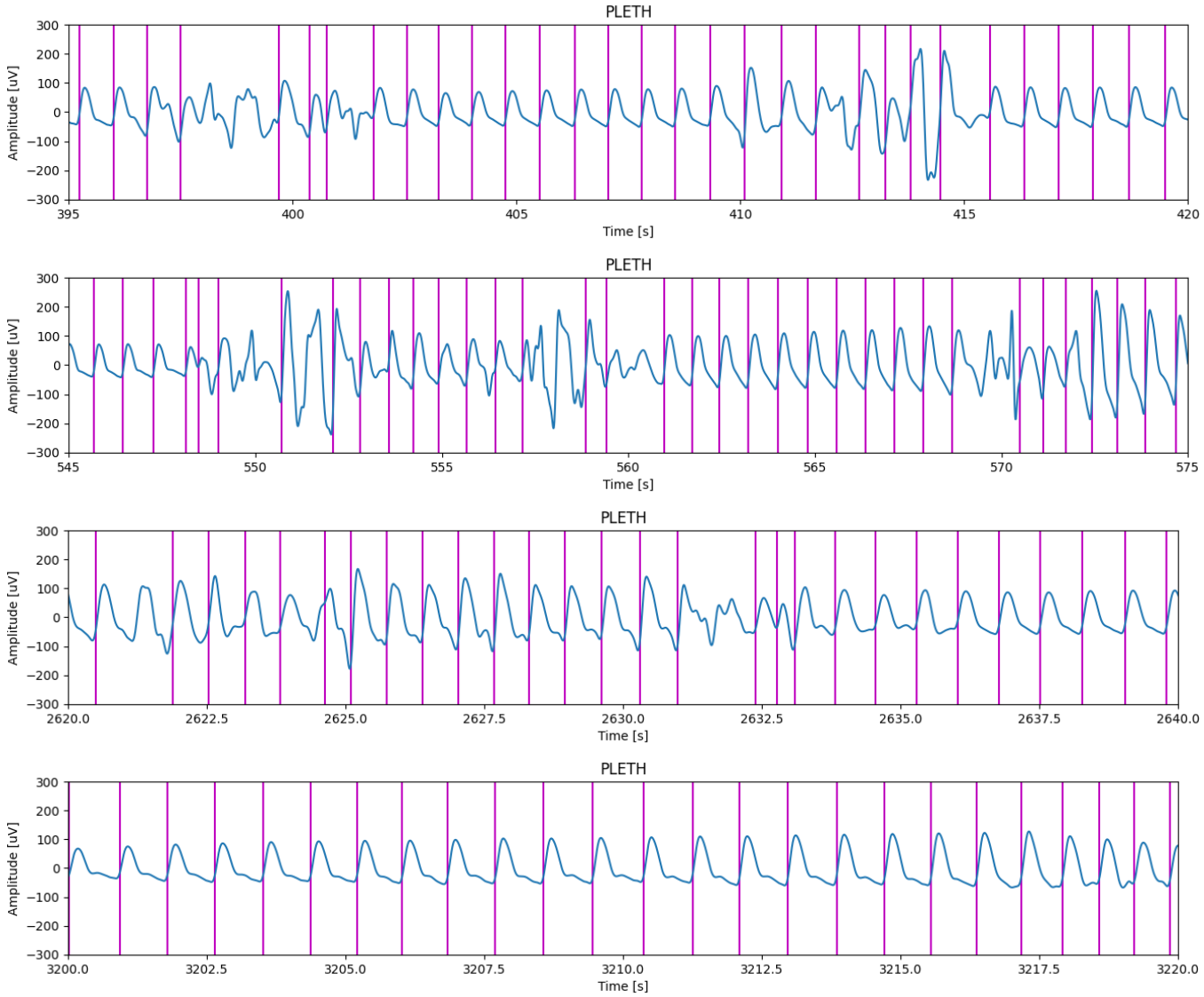

Figure 18: Results of the `biosppy.bvp.bvp` algorithm on finding the onsets (pink) of the filtered PPG signal (blue). Envelopes 1, 2, 3 and 4 are shown vertically stacked by this order.

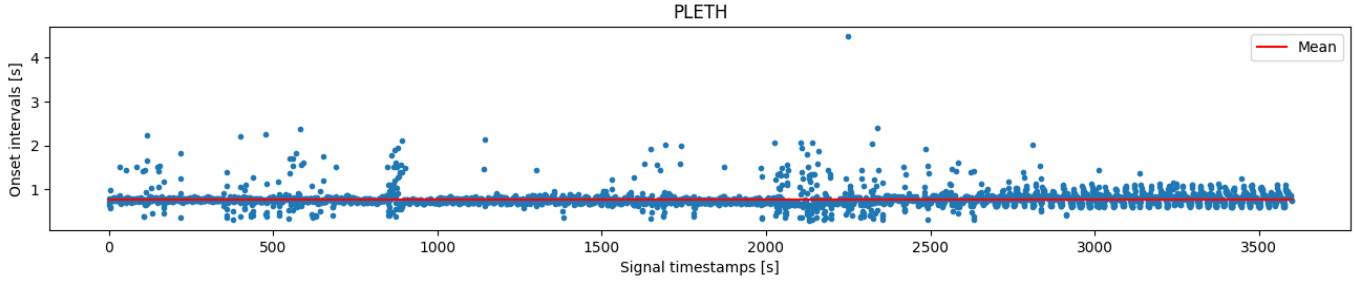

Figure 19: PPI of the filtered PPG signal (blue) based on the onsets found by `biosppy.bvp.bvp` algorithm. The mean is highlighted by the red line.

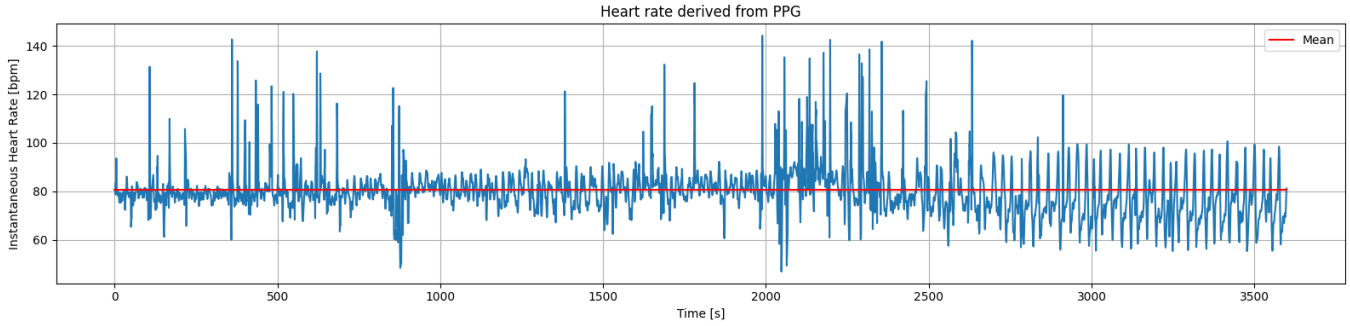

Figure 20: Instantaneous heart rate in beats-per-minute (bpm) derived from the PPG signal by the `biosppy.bvp.bvp` algorithm. The mean heart rate of the acquisition is highlighted by the red line.

2 Up-Down-Re-sampling, Decimation and Interpolation

For Exercise 2 (`ex2/ex2.py`), a 3 second sinusoidal signal was created with frequency 2 kHz and sampling frequency of $SF = 22.4$ kHz, using procedure `ex2_1_generate_sinusoid`. A snippet of it is shown in Figure 21. This signal was written into a wav file named `2k.wav`.

Procedure `ex2_2_downsample` allows us to downsample a signal and plot it against the original, using `scipy.signal.resample`, which uses the Fourier method to downsample the signal if the number of samples given is a fraction of the original number of samples. This fraction is obtained dividing the original number of samples by the desired downsampling factor. We run the procedure on the original signal for factors 2, 4, 8 and 16. The results are shown in Figure 22.

Procedure `ex2_3_decimate` allows us to decimate a signal, that is, downsample it after applying an anti-aliasing filter. An order 8 Chebyshev type I filter was used, lowpassing frequencies below $0.8/M \cdot SF$ Hz, where M is the downsampling factor. We run the procedure on the original signal for factors 2, 4, 8 and 16. The results are shown in Figure 23. We can see in that using factors 8 and 16 for downsampling/decimating do not yield relevant results.

Procedure `ex2_4_absolute_spectrum` plots the absolute frequency spectrum of a signal given in the time domain, by taking its FFT. It used function `spect` which is a simple translation of the one provided in `spect.m`. The spectrum for the original signal can be found in the bottom panel of Figure 21; for the downsampled signals in Figure 24; and for the decimated signals in Figure 25.

In order to not experience aliasing, the following condition must be satisfied:

$$B < \frac{0.5}{T} \cdot \frac{1}{M}$$

where B is the signal bandwidth, T is the signal period. One can ask what is the theoretical maximum downsampling factor without aliasing, and that is only true for M values less than:

$$M \leq \frac{1}{2 \cdot B \cdot T} = \frac{SF}{2 \cdot B}$$

where $SF = 1/T$ is the signal sampling frequency. Since the original signal frequency is 2000 Hz, let $B = 2000$, and then we can state:

$$M \leq \frac{22400}{2 \cdot 2000} = 5.6$$

which means the theoretical maximum integer M is 5, before we start experiencing aliasing.

Procedure `ex2_6_interpolate` allows us to interpolate a signal, that is, upsample it and apply an anti-aliasing filter after. It uses `scipy.signal.upfirdn` which processes the signal in three steps:

1. Upsamples by an integer factor M (with zero insertion).
2. Applies a lowpass FIR filter with kaiser window and cutoff frequency of $0.5/M \cdot SF$ Hz.
3. Decimation by an integer factor P.

Overall, it provides interpolation by a rational factor, M/P. Running the procedure on the original signal, the order of the filter chosen was 205, to allow for a 400 Hz transition width and 60 dB attenuation in the stop band. The results with a factor 2/11 (cutoff = 5.6 kHz) and with a factor 4/23 (cutoff = 2.8 kHz) are depicted in Figure 26. The results of the same procedure to `data/looneyTunes.wav`, with a filter order of 74 and a factor 3/8 (cutoff = 1.3 kHz) are shown in Figure 27.

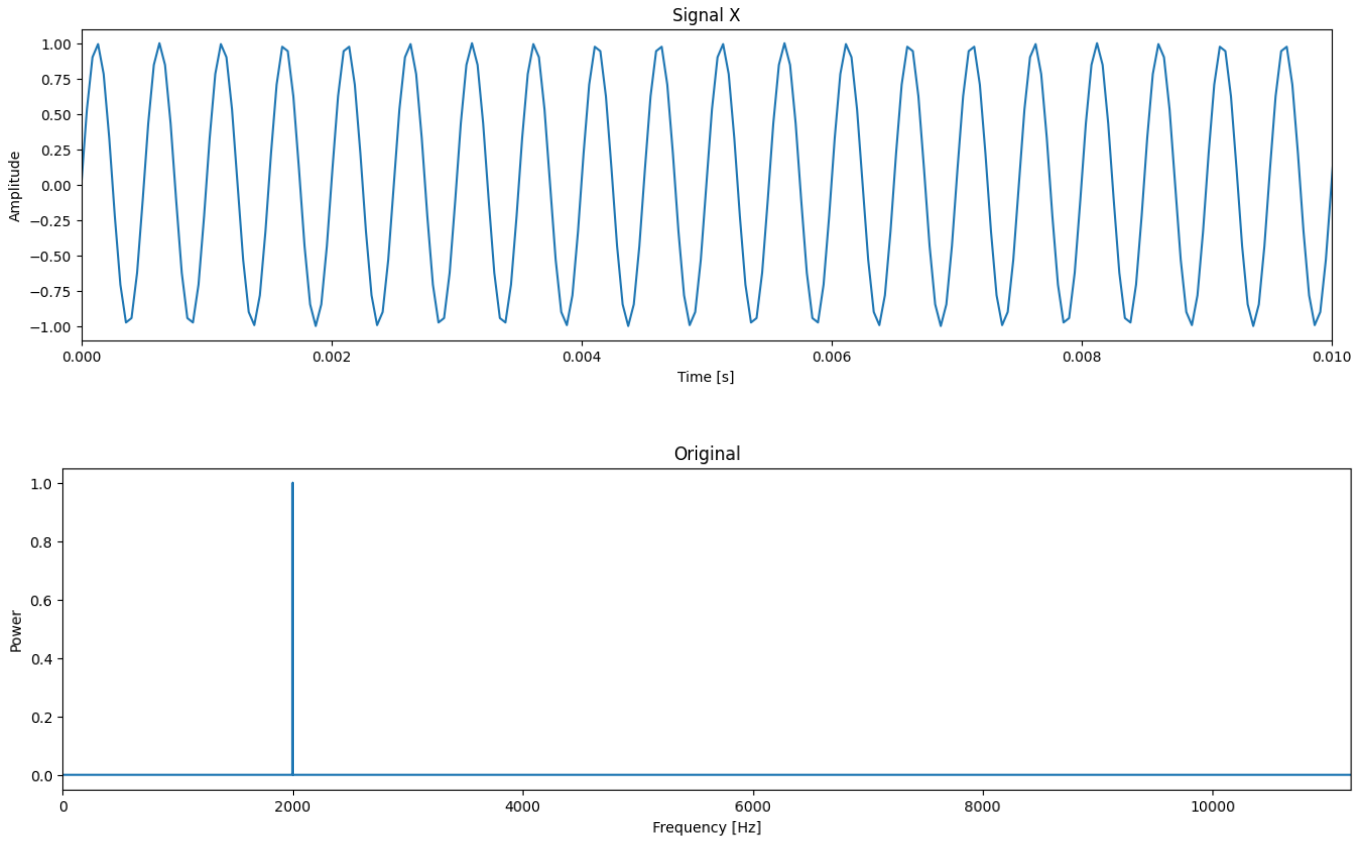

Figure 21: Generated original sinusoidal signal. Sampled at 22.4 kHz. Top panel: Time domain in envelope [0, 0.01] s. Bottom panel: Absolute frequency spectrum.

Procedure `ex2_8_resample` converts the signal in `data/tone.wav` to new desired sampling frequency. It uses the `scipy.signal.resample` function, which uses the Fourier method to downsample or upsample the signal. The original sampling frequency is 8 kHz. The results of upsampling it to 7.7 kHz and downsampling it to 4.1 kHz are shown in Figure 28.

References

[1] I. I. Christov. Real time electrocardiogram QRS detection using combined adaptive threshold. *BioMedical Engineering OnLine*, 3(1):28, Dec. 2004. ISSN 1475-925X. doi: 10.1186/1475-925X-3-28. URL `https://biomedical-engineering-online.biomedcentral.com/articles/10.1186/1475-925X-3-28`.

[2] M. Elgendi. Fast QRS Detection with an Optimized Knowledge-Based Method: Evaluation on 11 Standard ECG Databases. *PLoS ONE*, 8(9):e73557, Sept. 2013. ISSN 1932-6203. doi: 10.1371/journal.pone.0073557. URL `https://dx.plos.org/10.1371/journal.pone.0073557`.

[3] H. Gamboa. Multi-Modal Behavioural Biometrics Based on HCI and Electrophysiology (PhD Thesis). Technical report, IST, 2008.

[4] P. Hamilton. Open source ECG analysis. In *Computers in Cardiology*, pages 101–104, Memphis, TN, USA, 2002. IEEE. ISBN 978-0-7803-7735-6. doi: 10.1109/CIC.2002.1166717. URL `http://ieeexplore.ieee.org/document/1166717/`.

[5] W. A. H. Engelse and C. Zeelenberg. A single scan algorithm for QRS-detection and feature extraction. *1 in Cardiology*, 6:37–42, 1979.

[6] W. Zong, T. Heldt, G. Moody, and R. Mark. An open-source algorithm to detect onset of arterial blood pressure pulses. In *Computers in Cardiology, 2003*, pages 259–262, Thessaloniki Chalkidiki, Greece, 2003. IEEE. ISBN 978-0-7803-8170-4. doi: 10.1109/CIC.2003.1291140. URL `http://ieeexplore.ieee.org/document/1291140/`.

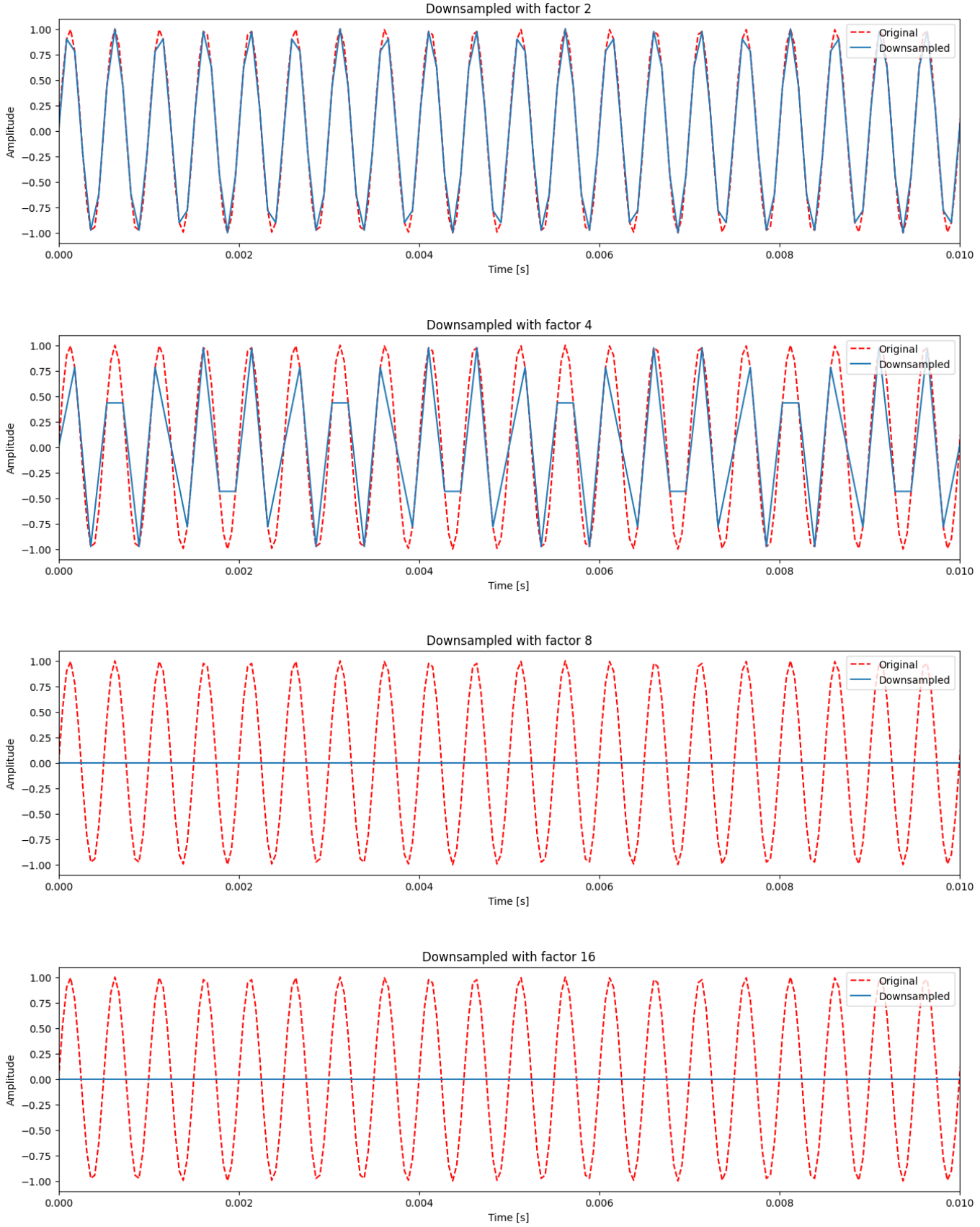

Figure 22: Downsampled signals (blue) and the original signal (red) in the interval [0, 0.01] s. From top to bottom, downsampling factors of 2, 4, 8 and 16.

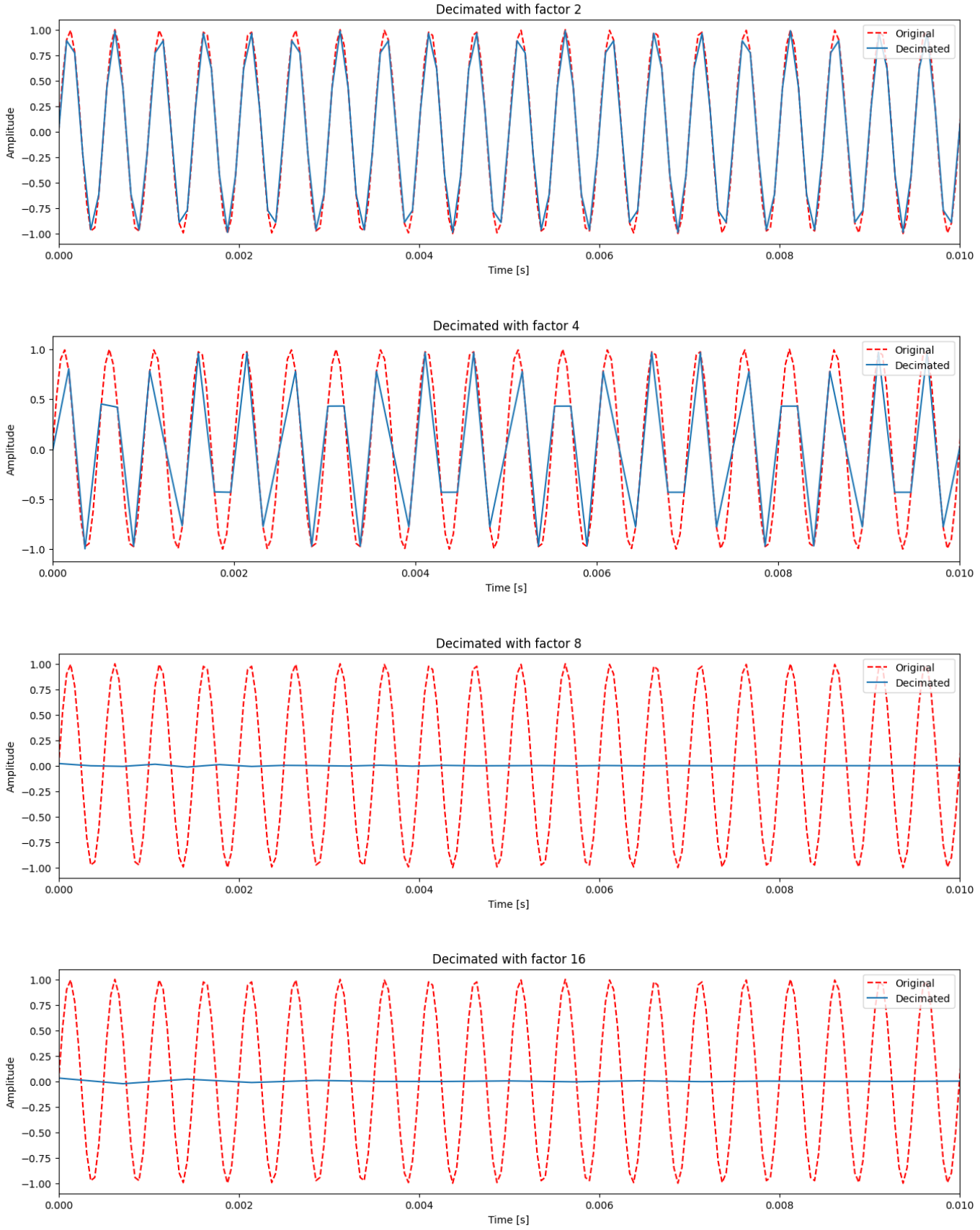

Figure 23: Decimated signals (blue) and the original signal (red) in the interval [0, 0.01] s. From top to bottom, decimation factors of 2, 4, 8 and 16.

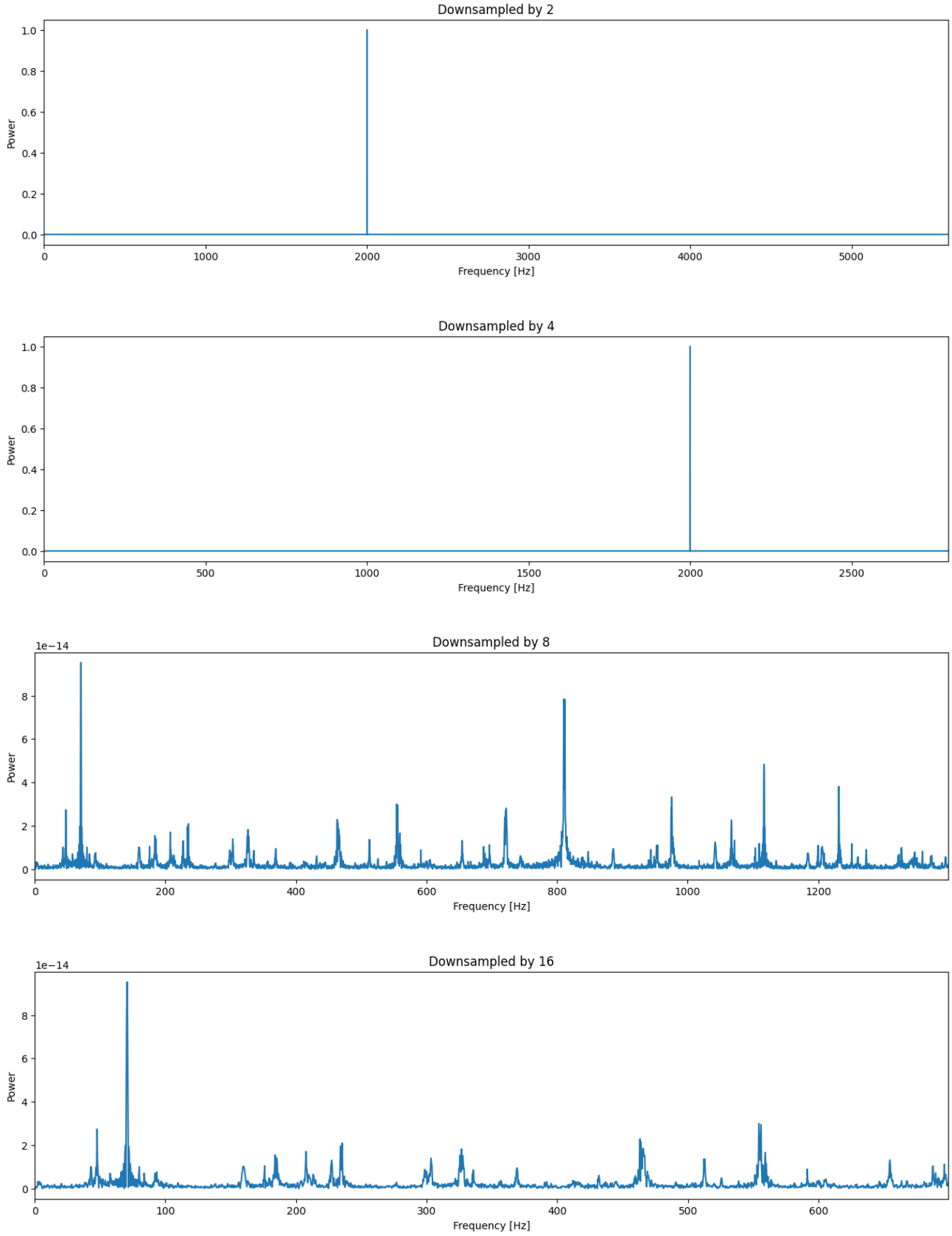

Figure 24: Absolute frequency spectra of the downsampled signals. From top to bottom, downsampling factors of 2, 4, 8 and 16.

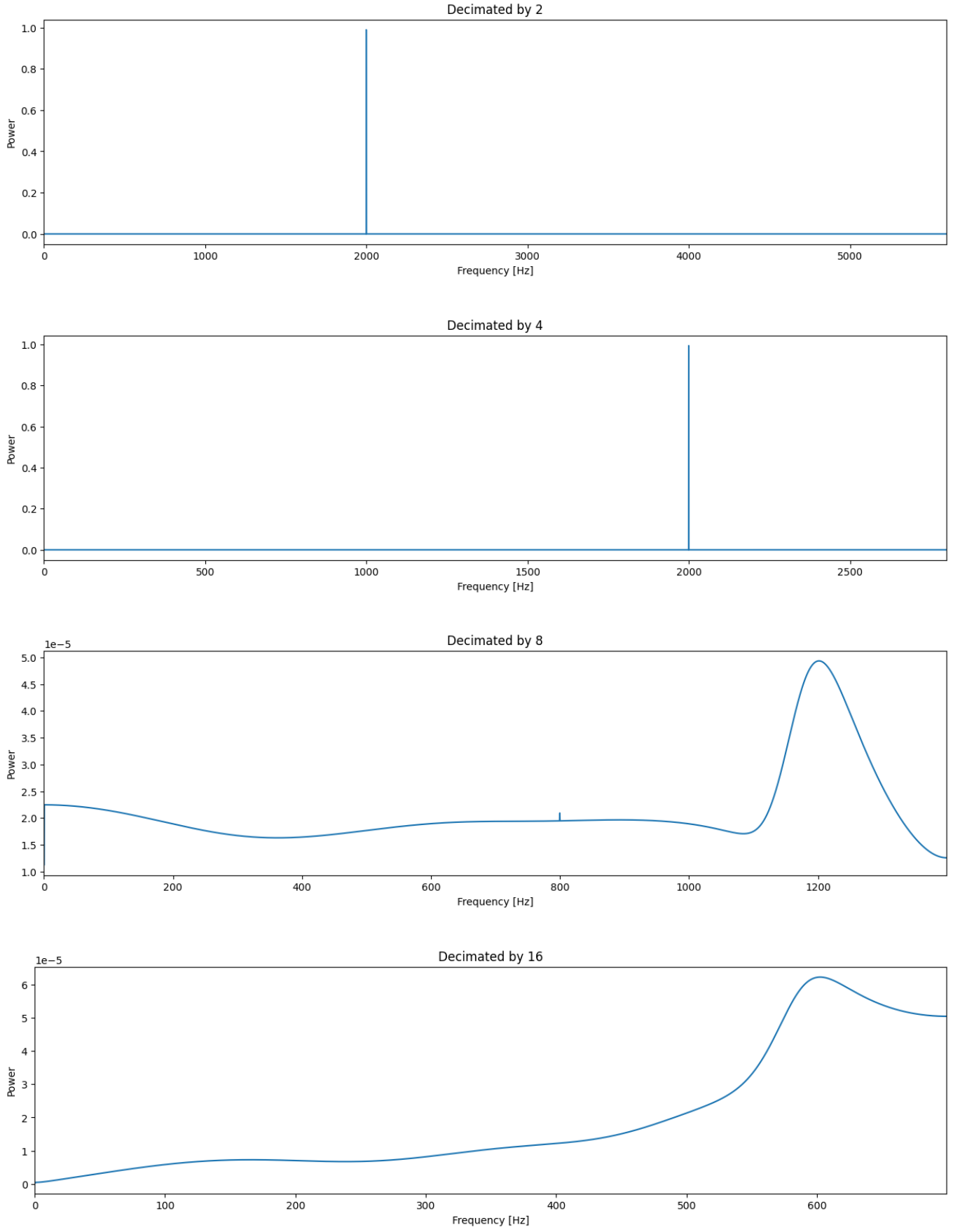

Figure 25: Absolute frequency spectra of the decimated signals. From top to bottom, decimating factors of 2, 4, 8 and 16.

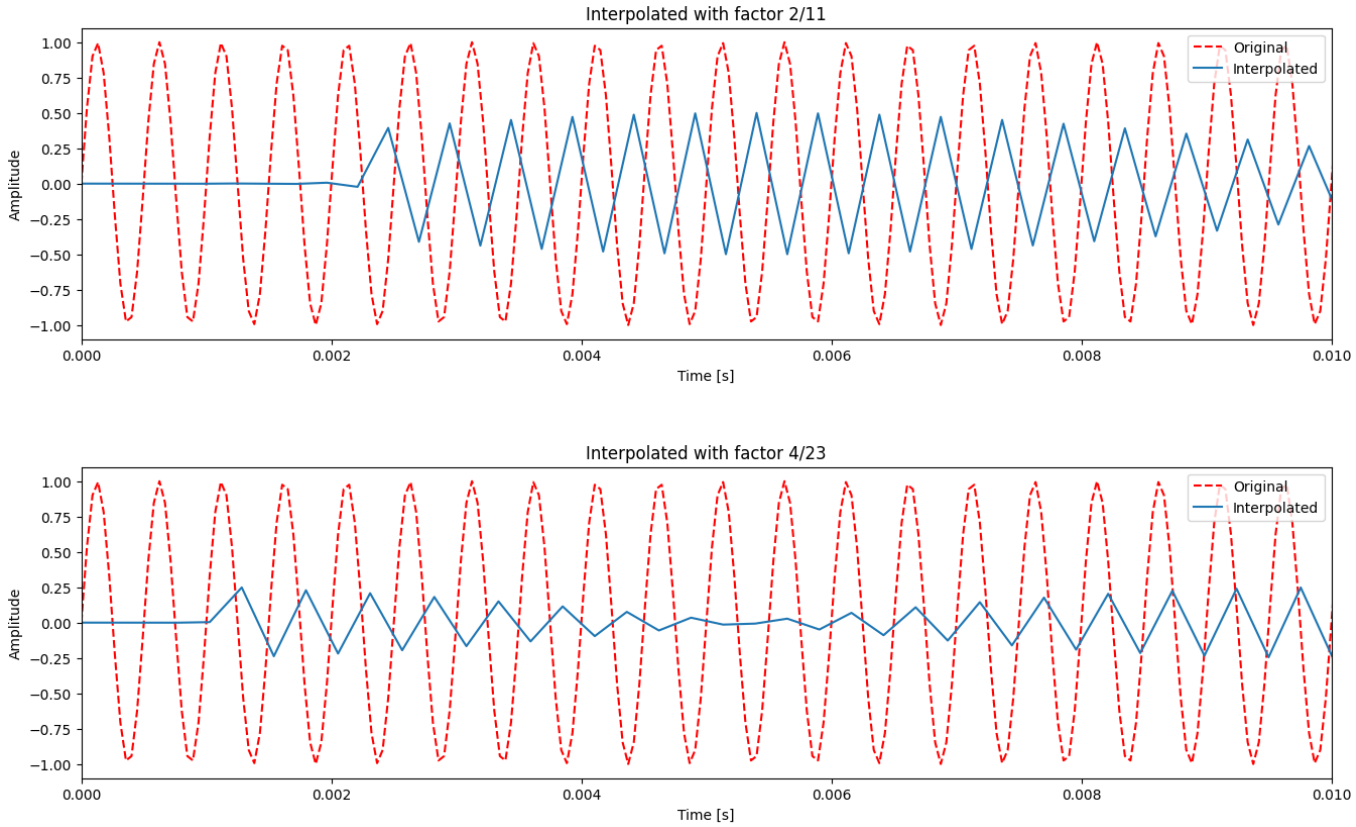

Figure 26: Interpolated signals (blue) and the original signal (red) in the interval [0, 0.01] s. From top to bottom, interpolation/decimation factors of 2/11, 4/23.

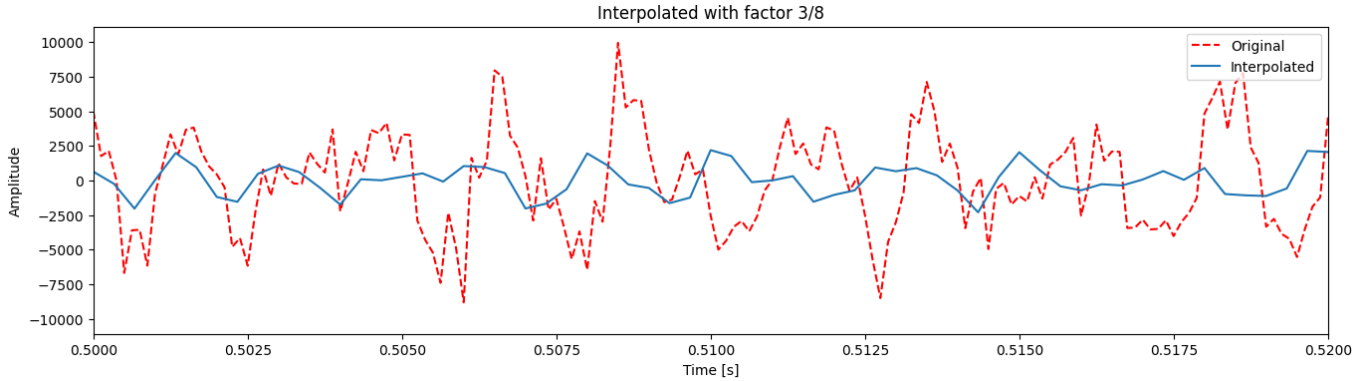

Figure 27: Interpolated signal (blue) and the original signal (red) in `data/looneyTunes.wav` plotted in the interval [0.50, 0.52] s.

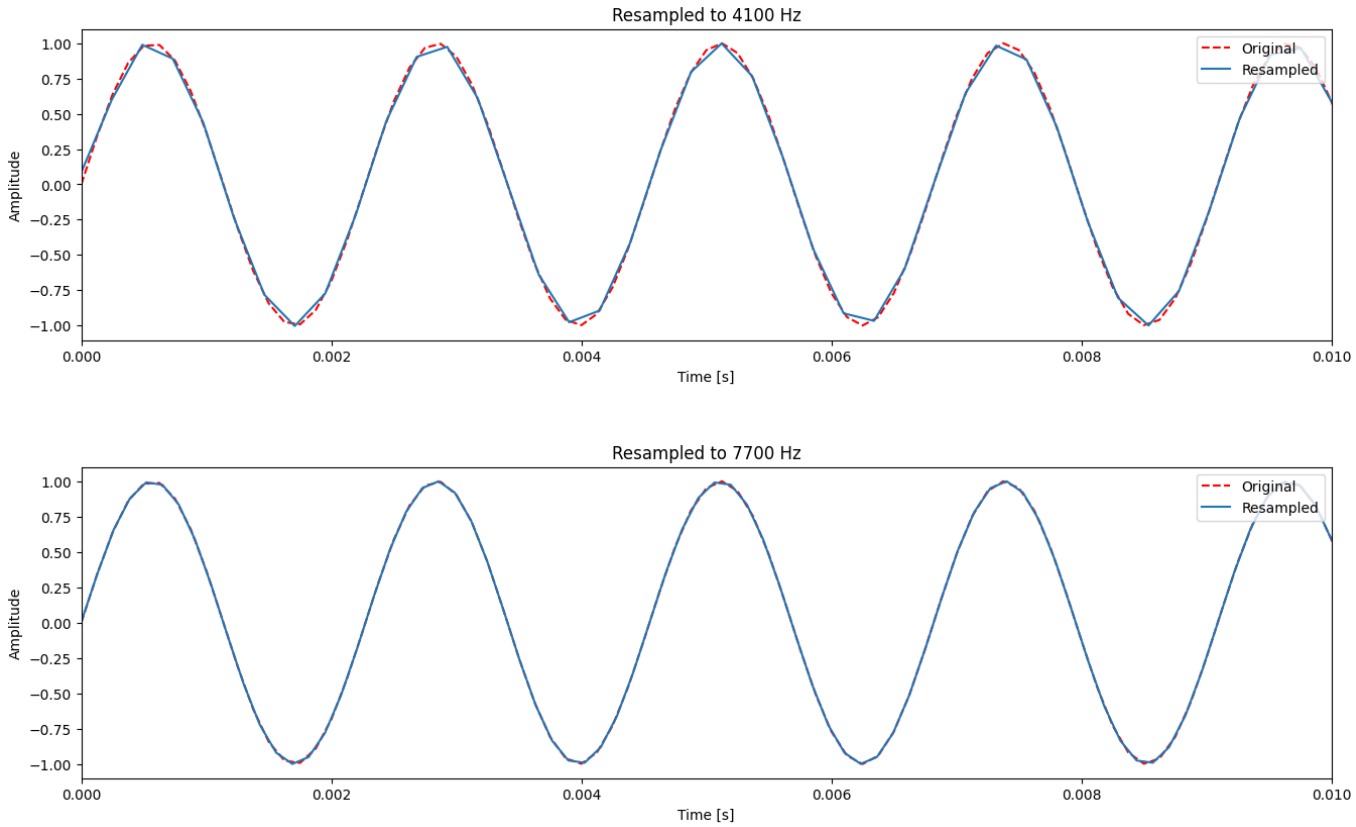

Figure 28: Resampled signals (blue) of the signal `data/tone.wav` (red) in the interval [0, 0.01] s. Top panel: downsampling to 4100 Hz. Bottom panel: upsampling to 7700 Hz.