The extended adaptive Quasi-Harmonic Model
C++ version

July 25, 2023

Abstract

Speech is the most universal method of communication among humans. Understanding its formation and uses has shaped several disciplines ranging from medicine to entertainment. Detection, recording, transformation of speech is essential in several fields, from sound recordings, to speech synthesizer, encompassing human-computer interfaces and theoretical modeling. This report focus on a novel method for processing human voices. The method relies on the extension of the adaptive quasi harmonic model, where both amplitudes and frequencies of the speech sample are being matched in order to reproduce the original signal with minimal distortions. The algorithm was originally implemented in MATLAB. Here, a new version of the code written in C++ is developed and tested.

Contents

List of Figures

Listings

List of Tables

List of Acronyms

SFM Source-filter model

SM Sinusoidal model

sSM Stationary sinusoidal models

aSM Adaptive sinusoidal models

QHM Quasi harmonic model

aQHM Adaptive quasi harmonic model

eaQHM Extended adaptive quasi harmonic model

SRER Signal to reconstruction error ratio

Chapter 1

Introduction

1.1 Overview and Applications

The importance of speech in human history cannot be understated. The ability of communicating ideas, establishing trades, and entertaining audiences is at the heart of human society.

Nowadays, everybody is aware of intelligent virtual assistants [1–3] such as Apple's Siri, Microsoft's Cortana, Google's Assistant and Amazon's Alexa. All of these tools are examples of well-conceived human-machine interfaces that allow to drive digital devices with the sound of one's voice.

Speech processing is also of paramount importance in medicine. Event related brain potentials are small voltages produced within the human brain that are associated with brain activity [4, 5]. These electric signals are produced in response to several stimuli, such as motor or sensory events. The analysis of such signals can be used both to understand the functioning of the human brain as well as to identify anomalies in the brain response. Among the different signals produced in this process, the waveforms labeled as N400 and P600 are directly related to speech processing and their analysis can provide essential information on the patient.

Other methods used in medicine couple voice sampling and processing with machine learning models or neural networks to identify speech patterns that are associated with several pathologies [5–8].

Of course, speech processing is essential in telecommunications [9]. Modern telecommunication technologies rely heavily on digital processing of the signal, both to safely deliver it as well as for storage. In these techniques, it is essential that the original signal, which is by nature analog, is correctly and reliably converted into a digital signal with minimal distortion. An important example of this is voice over IP [10].

Another area where signal processing is of great importance is storage and streaming. In this context, it is very important to be able to compress the signal to reduce storage (or increase bandwidth if streaming), while at the same time maintaining a high level of fidelity. Examples of this are the several formats for storing songs, among which the most famous is arguably the MP3 format [11].

A final example of speech processing is found in noise canceling technologies, that allow to enhance and refine a signal within a noisy environment [12].

1.2 Speech modeling

Based on the discussion of the previous section, it is no surprise that a lot of effort and resources have been devoted to studying the production and processing of speech.

Concerning speech recognition, several methodologies have been put forward to date. Dynamic time warping [13] was developed to compare sequences of sounds to a reference one. This method relies on warping the speed of each sequence in a nonlinear fashion to match sounds uttered by different people at different speed.

Another important class of methods was based upon the hidden Markov model [14]. These are statistical models that assume that the final result (namely, speech) is a product of hidden processes (for instance, the workings of vocal cord and larynx) which are not modeled in details, but as a sequence of probabilistic events. In practice, the hidden states of the system are fitting parameters that are varied until they match the required output. While generating an initial set of parameters may be extremely time consuming, once the model is functioning, it is relatively fast to apply.

More modern approaches involve neural networks [15–17]. These models are generally very complex and require large datasets for the evaluation of their parameters. They aim at mimicking the way the human brain learns to generate computational models that can understand human language and grammar. A very important example of this is ChatGPT [18], a neural network that needed over 100 billion parameters and is only possible thanks to the vast amount of information available on the internet.

The algorithms described so far can reproduce and even understand language with a very high degree of confidence. However, they do not provide a theory of how speech is formed and propagated.

To tackle this problem, two main theoretical frameworks have been put forward: The source-filter model and the sinusoidal model.

In the source-filter model (SFM) first proposed by Fant [19–24] the speech is a result of two components interacting with each other. The source (for instance the larynx in people) and a filter that modulates the sound (for instance, the vocal cords). This model relies on evaluating the resonances of a wave within a cavity. In practice, this model considers the actual mechanism that generates the final sound. On the other hand, the sinusoidal models (SMs) represent the signal as a time series of sinusoidal functions where both amplitude and frequency of these functions must be modeled [25–27]. While SMs may lose details about the mechanics of speech formation, their parameters are much easier to estimate.

1.3 Sinusoidal models

Sinusoidal models are easier to evaluate and vary based on the methodology employed to update and select the proper frequencies, amplitudes and phases of the sinusoidal functions involved. In all cases, these models are easier to parameterize than the equivalent SFM, making them an excellent choice for developing algorithms for speech recognition and modeling. Sinusoidal models fall themselves into two groups: Stationary and adaptive.

Stationary sinusoidal models (sSMs)

In stationary models, the signal s as a function of time t is described as:

$$s(t) = \left(\sum_{k=-K}^{K} C_k(t)\psi_k(t) \right) w(t) \tag{1.1}$$

$w(t)$ selects the time interval under examination. $w(t)$ is a function that declines rapidly at its edges and is used to define a time window used to fit the parameters of the series. In stationary models, the function

$$\psi_k(t) = 1 \cdot e^{i2\pi f_k t} \tag{1.2}$$

has fixed amplitude equal 1 and fixed frequencies f_k. So that, the fitting parameters are:

$$C_k(t) = a_k \tag{1.3}$$

It is important to notice that a_k is a complex number and is a time independent parameter within the selected time window $w(t)$.

Adaptive sinusoidal models (aSMs)

In adaptive models, on the other hand:

$$\psi_k(t) = \alpha_k(t) \cdot e^{i\phi_k(t)} \tag{1.4}$$

where the amplitude is not fixed any longer and the exponent of the complex exponential depends on the phase $\phi_k(t)$:

$$\phi_k(t) = \phi_k(t_i) + \int_{t_i}^{t_i+t} 2\pi f_k(u)du \tag{1.5}$$

where t_i is the center of the time window $w(t)$, and $\phi_k(t)$ is the instantaneous phase at time t .

1.4 Quasi harmonic model (QHM)

In order to have a functioning model, it is imperative to find a methodology that allows to quickly and reliably estimate the parameters needed. For instance, sSMs rely on a good estimate of the set of frequencies f_k. Assuming that $\hat{f}_k$ is the best estimate for the correct frequency f_k:

$$f_k = \hat{f}_k + \eta_k \tag{1.6}$$

Thus, the purpose of the fitting scheme is to minimize the error η_k. In other words, the exact solution:

$$s(t) = \left(\sum_{k=-K}^{K} a_k e^{i2\pi f_k t} \right) w(t) \tag{1.7}$$

is approximated as:

$$\hat{s}(t) = \left(\sum_{k=-K}^{K} a_k e^{i2\pi \hat{f}_k t} \right) w(t) \tag{1.8}$$

In sSM, only the frequencies $\hat{f}_k$ and their coefficients a_k are modified. At this stage, assuming that $\hat{f}_k$ is the best set of frequencies available, further improvement is only possible through the coefficients a_k.

The quasi harmonic model (QHM) replaces the coefficients a_k with $a_k + tb_k$:

$$\hat{s}_q(t) = \left(\sum_{k=-K}^{K} (a_k + tb_k) e^{i2\pi \hat{f}_k t} \right) w(t) \tag{1.9}$$

In this equation, the term tb_k acts as a further "correction" to the error introduced by the choice of the frequencies.

The QHM can be paired to both stationary and adaptive sinusoidal models. In the case of an aSM, the equation becomes:

$$\hat{s}_q^a(t) = \left(\sum_{k=-K}^{K} (a_k + tb_k) e^{i\left(\hat{\phi}_k(t+t_i) - \hat{\phi}_k(t_i)\right)} \right) w(t) \tag{1.10}$$

It should be noticed, that in sSM, the QHM was used in order to ease issues with the estimation of frequencies. In the case of aSM, the same issues exists but for the phases through Eq. 1.5.

Chapter 2

Implementation details

2.1 Extended adaptive QHM (eaQHM)

In the previous chapter, several techniques for speech modeling have been described. In this chapter, the focus will be on eaQHM. In the previous chapter, it was introduced Eq. 1.10:

$$\hat{s}_q^a(t) = \left(\sum_{k=-K}^{K} (a_k + tb_k) e^{i\left(\hat{\phi}_k(t+t_i) - \hat{\phi}_k(t_i)\right)} \right) w(t) \tag{2.1}$$

However, this formula was derived for remedying errors in the estimation of phases. Another parameter that must be correctly evaluated is the amplitude. The eaQHM aims at extending the aQHM with the assessment of amplitudes as well. For this reasons, Eq. 1.10 is modified as:

$$\hat{s}_q^{ea}(t) = \left(\sum_{k=-K}^{K} (a_k + tb_k) \hat{\alpha}_k(t) e^{i\left(\hat{\phi}_k(t+t_i) - \hat{\phi}_k(t_i)\right)} \right) w(t) \tag{2.2}$$

with the addition of the new parameter:

$$\hat{\alpha}_k(t) = \frac{A_k(t+t_i)}{A_k(t_i)} \tag{2.3}$$

where $A_k(t)$ is the instantaneous amplitude at time t.

2.2 Overview

Two versions of the code are currently available. One version is written in MATLAB and another version is written in C++ in combination with the scientific library Armadillo [28,29]. Both follow the same algorithm described later on. In both cases, the algorithm is wrapped within a function that requires several parameters to be passed as arguments. These arguments are generally produced by other functions or other programs altogether.

In brief, the code starts with an initialization stage. In this stage, the function needs several input parameters to be passed to it. One, of course, is the original signal as well as some features of this signal, such as sampling frequency f_s, number of points, duration, etc. For the initialization stage, the most important feature is the fundamental frequency f_0 for each $frame$. The code, indeed, does not fit each single time instant, but breaks the signal into time intervals of fixed length, called frames. This is necessary to reduce noise and improve the fit. Once the fit has been performed, an interpolation stage recovers details for each time instant.

The aim of the code is to extract a set of complex parameters $a_k, b_k, \hat{\alpha}_k, \hat{\phi}_k$ from the original signal s_o so that a new signal s_{rec} can be recreated using Eq. 2.2.

The fit is performed in an iterative way, where a set of parameters is first estimated and then improved upon until the difference between s_o and s_{rec} is minimized.

In the following sections, details for each stage are provided.

2.3 Initialization

The code starts with reading the fundamental frequency f_0 for each frame and defines the other frequencies $f_k = kf_0$. A Blackman analysis window $w(t)$ is used as well. At this stage, the frequencies are kept constant and the parameters a_k and b_k are evaluated through a least square fit:

$$s(t) = w(t) \sum_{k=-K}^{K} (a_k + tb_k) e^{i2\pi k f_0 t/f_s} \tag{2.4}$$

where f_s is the sampling frequency and must be kept into account for discrete sampling. The phase ϕ_k for this stage can be computed as:

$$\phi_k^{int}(t) = \angle a_k(t_i) + \int_{t_i}^{t} \frac{2\pi}{f_s} k f_0(u) du \tag{2.5}$$

Least square fitting

As the least square fitting is used several times throughout the code, it is worth describing in details how the system of equations to be solved is set up. This process is similar to the one used for the other systems of equations described later, so that it should be straightforward to extend it to those.

Let's start with a very simple fit. Let's assume that the system has two fundamental frequencies $f_1 = f_0$ and $f_2 = 2f_0$. And let's consider three points in time t_1, t_2, t_3. Thus, three equations can be written (for now, the window term $w(t)$ is ignored):

$$(a_1 + t_1 b_1)e^{i2\pi f_1 t_1} + (a_2 + t_1 b_2)e^{i2\pi f_2 t_1} = s_1 \tag{2.6}$$
$$(a_1 + t_2 b_1)e^{i2\pi f_1 t_2} + (a_2 + t_2 b_2)e^{i2\pi f_2 t_2} = s_2 \tag{2.7}$$
$$(a_1 + t_3 b_1)e^{i2\pi f_1 t_3} + (a_2 + t_3 b_2)e^{i2\pi f_2 t_3} = s_3 \tag{2.8}$$

where s_n is the actual value measured at time t_n of the original signal s_o. These equations can be rearranged:

$$a_1 e^{i2\pi f_1 t_1} + a_2 e^{i2\pi f_2 t_1} + t_1(b_1 e^{i2\pi f_1 t_1} + b_2 e^{i2\pi f_2 t_1}) = s_1 \tag{2.9}$$
$$a_1 e^{i2\pi f_1 t_2} + a_2 e^{i2\pi f_2 t_2} + t_2(b_1 e^{i2\pi f_1 t_2} + b_2 e^{i2\pi f_2 t_2}) = s_2 \tag{2.10}$$
$$a_1 e^{i2\pi f_1 t_3} + a_2 e^{i2\pi f_2 t_3} + t_3(b_1 e^{i2\pi f_1 t_3} + b_2 e^{i2\pi f_2 t_3}) = s_3 \tag{2.11}$$

This system of equations can then be converted into matrix form. Defining the vectors:

$$\mathbf{x} = \begin{bmatrix} a_1 \\ a_2 \\ b_1 \\ b_2 \end{bmatrix}; \quad \mathbf{s} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} \tag{2.12}$$

and the matrix:

$$\mathbf{E} = \begin{bmatrix} e^{i2\pi f_1 t_1} & e^{i2\pi f_2 t_1} & t_1 e^{i2\pi f_1 t_1} & t_1 e^{i2\pi f_2 t_1} \\ e^{i2\pi f_1 t_2} & e^{i2\pi f_2 t_2} & t_2 e^{i2\pi f_1 t_2} & t_2 e^{i2\pi f_2 t_2} \\ e^{i2\pi f_1 t_3} & e^{i2\pi f_2 t_3} & t_3 e^{i2\pi f_1 t_3} & t_3 e^{i2\pi f_2 t_3} \end{bmatrix} \tag{2.13}$$

The equations can then be written as:

$$\mathbf{Ex} = \mathbf{s} \tag{2.14}$$

At this point it can be seen how the least square fitting is set up. The vector $\mathbf{s}$ is constructed sampling the original signal at the time instants t_n. Once the fundamental frequency f_0 is known, the frequencies $f_k = kf_0$ can be computed. With the frequencies f_k and the time series t_n, the matrix $\mathbf{E}$ can be constructed as well. Finally, the least square problem is solved and $\mathbf{x}$ is obtained. In MATLAB the least square problem is solved using the "\" operator, while C++/Armadillo provides the function *solve*.

2.4 eaQHM

This is an iterative phase where several iterations, called adaptations, are performed. The main difference between this stage and the initialization stage is that the amplitude $\hat{\alpha}_k$ is fitted as well, so that Eq. 2.4 becomes:

$$s(t) = w(t) \sum_{k=-K}^{K} (a_k + tb_k)\hat{\alpha}_k e^{i2\pi k f_0 t/f_s} \tag{2.15}$$

where the value of $\hat{\alpha}_k(t)$ can be evaluated using Eq.2.3 and the a_k from the previous step of the iteration (or from initialization). Putting everything together, our eaQHM equation looks like:

$$s(t) = \left(\sum_{k=-K}^{K} (a_k^{(m)} + tb_k^{(m)}) \left| \frac{a_k^{(m-1)}(t+t_i)}{a_k^{(m-1)}(t_i)} \right| e^{i2\pi f_k^{(m-1)} t} \right) w(t) \tag{2.16}$$

where m is the m-th adaptation that starts at $m = 1$. $m = 0$ is the initialization step described in the previous section. As it can be seen, as the $\hat{\alpha}_k$ and f_k are carried over from the previous adaptation, each adaptation only estimates the coefficients a_k and b_k, so that the same least square algorithm as before can be used.

From this fit, is then possible to compute the phases used in Eq. 2.2 using a modified version of Eq.1.5:

$$\phi_k(t) = \phi_k(t_i) + \int_{t_i}^{t_i+t} \frac{2\pi}{f_s} \left(f_k(u) + c_k(u)\right) du \tag{2.17}$$

where the $c(u)$ term is a consequence of using periodic functions. In fact, the phase is a continuous function so that $\phi_k(t_i) = \phi_k(t_i + t) + 2M\pi$, with M being an integer number. However, as the signal $s(t)$ is broken down into a sequence of frames, there is no guarantee that the value of M is retained between frames. The term $c(u)$ takes care of keeping the phase continuous between frames.

In addition, it can be shown that once a_k and b_k are known, it is possible to evaluate the error η_k from Eq. 1.6 as:

$$\eta_k = \frac{f_s}{2\pi} \frac{\Re(a_k)\Im(b_k) - \Im(a_k)\Re(b_k)}{|a_k|^2} \tag{2.18}$$

This error is used to generate a new set of frequencies f_k, together with the process described in the next section.

Frequency creation and destruction

One of the important aspects of this algorithm is that in addition to correcting frequencies using Eq. 2.18, the frequencies f_k are created and destroyed as well. Furthermore, as it will be explained later (Eq. 2.22), to compute the recreated signal s_{rec} only a_k and f_k are needed. As a consequence, if after the fit one of the $a_k = 0$, the associated frequency f_k is dropped as well. In this way the frequency is destroyed. However, from a computational point of view, this process presents some issues. In fact, the frequencies f_k are stored in a vector of fixed length K_{max}, with K_{max} being another argument passed to the code. Storing in memory elements of a vector that are not used is consuming resources and computational power for no reasons. Thus, an algorithm is devised to better store the set of f_k.

To understand the algorithm for handling frequencies, an example may be better suited. Let's focus on a specific frequency, f_1, and focus on how it evolves over time from the initial time t_0 to the final time t_{end}. The following algorithm is employed *only* if $f_1(t_0)$ and/or $f_1(t_{end})$ are zero. Let's assume that only 11 frames are available and let's assume that the value of $f_1(t)$ changes over time as:

$$0000ABC0000 \tag{2.19}$$

Here, f_1 is zero most of the time, so in most frames does not contribute to the s_{rec}, consuming resources. To improve on that, the algorithm adds two frequencies, copying the first and last frequency as follows:

$$A000ABC000C \tag{2.20}$$

Now, f_1 covers the same range $A - C$ of values as before, but it is "spread" over all of the frames. As it will be explained in the next section, an interpolation is performed on this vector to make it continuous, so that after interpolation it reads:

$$AabcABCdefC \tag{2.21}$$

The interpolated values have been reported in lower case. As it can be seen, new frequencies have been created.

Summarizing, this algorithm, has kept the original range $A - C$, while introducing novel values for f_1. These values are then fitted again in the next adaptation and if $a_1 \neq 0$ are retained. Combining this process with the correction Eq. 2.18, the frequencies are changed to improve fitting.

2.5 Interpolation

At the end of each adaptation an interpolation step is performed before starting the next adaptation. To understand the reason why an interpolation is needed, let's go back to Eq. 2.4. In order to reduce the error in the fit, the signal is broken down into frames. Each frame covers a time interval that contains several samples of the original signal, let's say s_1 to s_n. However, the fit is performed on one frame at a time, so that each frame has, for instance, just one a_k and one b_k for the whole interval. In order to recover more details from the original signal, an interpolation is necessary. In this way, a time dependence of the parameters a_k and b_k within each interval is recovered.

In practice, at the end of each adaptation, an interpolation is performed not directly for a_k and b_k, but for f_k, $|a_k|$. The phase is interpolated via integration.

2.6 Error

As the methodology here proposed is iterative in nature, a criterion for stopping the algorithm must be provided. In this case, a good criterion is comparing how close the fitted signal is to the original signal s_o. The signal s_{rec} recreated at the end of each adaptation, after the interpolation is:

$$s_{rec}(t) = \sum_{k=-K}^{K} |a_k(t)| \cos(\phi_k^{int}(t)) \tag{2.22}$$

where only the real part of the signal is retained for comparison. The error is estimated using the SRER

$$\mathrm{SRER} = 20 \log_{10} \frac{\mathrm{std}(s_o(t))}{\mathrm{std}(s_o(t) - s_{rec}(t))} \tag{2.23}$$

where std is the standard deviation. The algorithm keeps running as long as SRER keeps increasing.

Chapter 3

Evaluation

The conversion from MATLAB to C++/Armadillo required adjusting a few features that could not be directly translated. One of the most relevant is the way the interpolation step is performed.

3.1 Interpolation scheme

MATLAB and C++/Armadillo have different functions for dealing with interpolation. For instance, the MATLAB code:

```
vq3 = interp1(x,v,xq,'spline','extrap')
```

Listing 3.1: MATLAB interp1

does not have an exact translation in C++. The reasons are twofold:

1. spline is not an available option in Armadillo.
2. extrap cannot be performed in Armadillo.

For these reasons, in C++ the code is equivalent to:

```
vq3 = interp1(x,v,xq,'linear',0.0)
```

Listing 3.2: MATLAB interp1 equivalent to C++/Armadillo

That is, the interpolation is *always* linear, and the extrapolation is *always* equal to zero.

To compare to MATLAB, two groups of tests were performed. In one group, the C++ code was compared to MATLAB with all the interpolations set to linear and extrapolation equal zero. In the second group, the comparison is between C++ and the original MATLAB code. For each variation of the code, the comparison involves the aQHM case where only the phase is adapted, and the eaQHM case, where both amplitude and phase are adapted.

MATLAB with linear interpolation

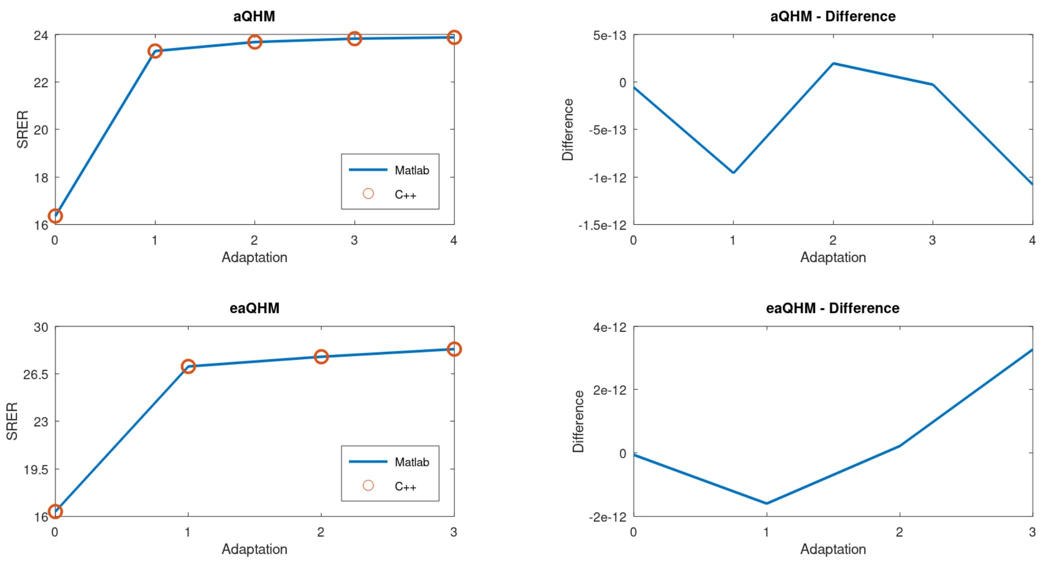

Figure 3.1: Comparison of the algorithm in C++ and MATLAB when using the same interpolation/extrapolation scheme.

In this set of tests, the MATLAB code was modified so that every interpolation is performed using the linear model instead of the spline one and the extrapolation is set equal zero. Under these assumptions the MATLAB and C++ code should perform very similarly. Figure 3.1 shows the comparison for the test data. The top of the figure shows the results for aQHM, whereas the bottom panels deal with the eaQHM case. On the left side panels, it is shown the value of SRER. As discussed earlier, the code stops as soon as the SRER starts diminishing. Adaptation zero is the initialization step. These graphs show that the first adaptation is the one that leads to a greater increase in SRER, with the following ones refining the result. Furthermore, the eaQHM converges much faster and produces a higher SRER. The right side panels show a point-by-point difference between the C++ and MATLAB code. The difference can be attribute to numerical approximations. Therefore, the two codes are practically identical.

MATLAB with spline interpolation

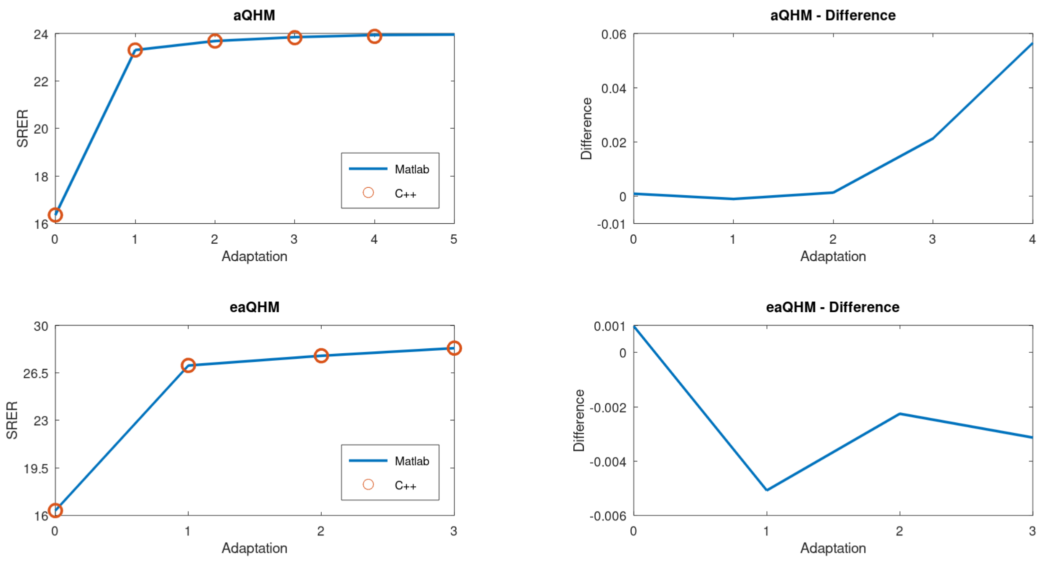

Figure 3.2: Comparison of the algorithm in C++ and MATLAB when using spline interpolation/extrapolation scheme.

This set of tests is between the new C++ code and the original MATLAB, which employs splines for interpolation and extrapolates the splines when necessary. The trends in Figure 3.2 are the same as the ones discussed in the previous section, with eaQHM still being superior to aQHM. The difference between the two codes is bigger in terms of SRER. It should also be noticed that the MATLAB code needs an extra adaptation before stopping, which provides a slightly better value for SRER. Overall, the results are very close between the two codes. In real life applications, this difference is unlikely to be relevant.

3.2 Performance

A comparison of the speed of execution of several tasks has been measured as well. The average time T and standard deviation σ have been calculated from 5 independent calculations. All calculations have been performed on an Intel Core i7-8700 CPU @ 3.20GHz with 32GB of RAM with Windows 10. The C++ code was compiled using Visual Studio 2019, whereas MATLAB version was R2021b.

File Name	Gender	MATLAB		C++		$\frac{T_C}{T_M}$
		T_M	σ_M	T_C	σ_C	
`0071_spa_KJC0017`	Male	59.762	0.333	142.66	1.450	2.4
`0072_spa_AGC0001`	Female	37.590	0.125	60.660	1.048	1.6
`0081_eus_IBE0031`	Male	63.057	0.202	140.21	1.360	2.2
`0082_eus_AGE0020`	Female	37.314	0.111	70.551	0.563	1.9
`0091_isl_m01-text1`	Male	31.262	0.141	90.344	1.957	2.9
`0092_isl_f01-text1`	Female	16.068	0.052	25.405	0.554	1.6
`0101_ind_ut-ml-m4`	Male	36.622	0.075	72.553	1.859	2.0
`0102_ind_f06-063a`	Female	31.444	0.143	43.146	1.334	1.4
`0111_tur_evenekm72`	Male	28.357	0.073	58.171	0.452	2.1
`0112_tur_yasemin51`	Female	16.794	0.097	27.455	0.527	1.6
`0121_fin_mv_0606`	Male	60.457	0.206	157.27	0.814	2.6
`0122_fin_01l_rich`	Female	16.600	0.071	28.763	1.148	1.7
`0131_ara_ut-ml-m2`	Male	64.496	0.156	153.11	0.839	2.4
`0132_ara_ut-ml-f1`	Female	51.538	0.103	102.96	2.813	2.0
`0141_chi_ut-ml-m1`	Male	30.526	0.126	61.615	1.899	2.0
`0142_chi_ut-ml-f3`	Female	16.262	0.045	40.638	1.441	2.5
`0151_kor_ut-ml-m1`	Male	34.621	0.116	78.854	0.251	2.3
`0152_kor_ut-ml-f3`	Female	31.452	0.071	29.780	1.388	0.9
`0161_rus_ut-ml-m2`	Male	64.209	0.075	96.749	0.659	1.5
`0162_rus_ut-ml-f2`	Female	26.762	0.096	46.049	1.224	1.7
`0171-nitech_jp_atr50`	Male	26.053	0.205	56.742	0.153	2.2
`0172-af049orgh`	Female	20.796	0.055	31.647	0.645	1.5
`0181-arctic_bdl1`	Male	37.792	0.121	78.060	1.252	2.1
`0182-arctic_slt1`	Female	22.518	0.103	41.781	0.811	1.8
`0191-XavierReference`	Male	34.109	0.102	70.992	0.344	2.1
`0192-Christine_01`	Female	15.203	0.130	26.624	0.384	1.7
`0201-emodb_m_39`	Male	18.889	0.046	41.145	0.118	2.2
`0202-emodb_f_107`	Female	26.155	0.072	46.431	0.986	1.8
`0211-Kostas268`	Male	93.123	0.113	244.61	1.536	2.6
`0212-Maria263`	Female	35.082	0.207	57.522	0.870	1.6
`0221-Luciano_K_It_m`	Male	29.756	0.056	87.952	0.766	3.0
`0222-Tiziana_C_It_f`	Female	19.970	0.133	35.349	0.430	1.8

Table 3.1: Benchmark of the two versions of the code in seconds.

From Table 3.1, it is clear that MATLAB is almost twice as fast as C++. To understand what happens, this analysis is further explored in the following sections.

3.3 Accuracy

File Name	Gender	MATLAB		C++		Δ	
		SRER	N	SRER	N	SRER	N
0071_spa_KJC0017	Male	34.877	3	34.883	3	−0.004	0
0072_spa_AGC0001	Female	32.218	5	32.137	4	0.081	1
0081_eus_IBE0031	Male	33.769	3	33.936	3	−0.167	0
0082_eus_AGE0020	Female	31.697	4	31.678	4	0.019	0
0091_isl_m01-text1	Male	31.747	2	32.467	3	−0.720	-1
0092_isl_f01-text1	Female	29.985	3	30.025	3	−0.040	0
0101_ind_ut-ml-m4	Male	35.128	4	35.251	4	−0.123	0
0102_ind_f06-063a	Female	26.020	5	25.853	5	0.167	0
0111_tur_evenekm72	Male	32.151	2	32.228	2	−0.077	0
0112_tur_yasemin51	Female	31.590	3	31.497	3	0.093	0
0121_fin_mv_0606	Male	34.951	2	34.997	2	−0.046	0
0122_fin_01l_rich	Female	30.708	3	30.932	3	−0.224	0
0131_ara_ut-ml-m2	Male	36.905	3	37.045	3	−0.140	0
0132_ara_ut-ml-f1	Female	31.685	6	31.418	6	0.267	0
0141_chi_ut-ml-m1	Male	26.943	3	27.066	3	−0.123	0
0142_chi_ut-ml-f3	Female	26.160	3	26.979	5	−0.819	-2
0151_kor_ut-ml-m1	Male	30.742	2	30.723	2	0.019	0
0152_kor_ut-ml-f3	Female	25.210	5	24.754	3	0.456	2
0161_rus_ut-ml-m2	Male	32.438	3	32.238	2	0.200	1
0162_rus_ut-ml-f2	Female	28.109	3	28.009	3	0.100	0
0171-nitech_jp_atr50	Male	35.091	2	34.996	2	0.095	0
0172-af049orgh	Female	33.700	6	33.432	6	0.268	0
0181-arctic_bdl1	Male	34.060	2	34.059	2	0.001	0
0182-arctic_slt1	Female	31.926	3	31.904	3	0.022	0
0191-XavierReference	Male	34.200	2	34.213	2	−0.013	0
0192-Christine_01	Female	31.650	3	31.442	3	0.208	0
0201-emodb_m_39	Male	33.283	2	33.369	2	−0.086	0
0202-emodb_f_107	Female	30.439	6	30.233	6	0.206	0
0211-Kostas268	Male	33.523	2	33.681	2	−0.158	0
0212-Maria263	Female	26.196	3	26.174	3	0.022	0
0221-Luciano_K_It_m	Male	32.561	2	32.451	3	0.110	-1
0222-Tiziana_C_It_f	Female	32.071	3	31.996	3	0.075	0

Table 3.2: SRER in dB and number of adaptations N.

As discussed in Section 3.1, a small difference between MATLAB and C++ is expected in accuracy. To further investigate the extend of such difference, several files have been tested. As shown in Table 3.2, the difference Δ is very small. In addition, except for a few cases, the number of adaptations N is the same for both cases.

It is possible to conclude that the two codes are producing comparable results, so that the difference measured in section 3.2 is related to the way the least square fitting is implemented.

3.4 Least Square Fitting

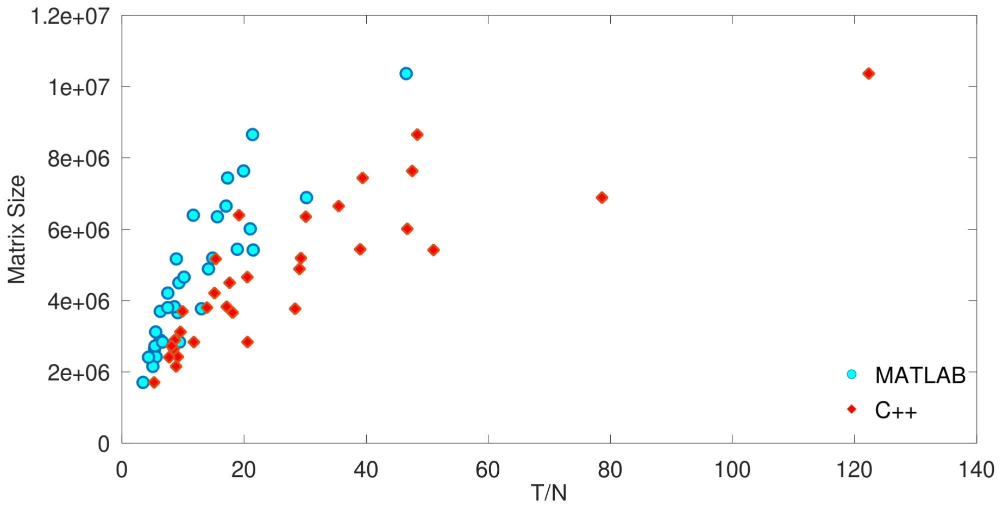

Figure 3.3: How each code scales with the size of the matrix.

As discussed in Chapter 2, the algorithm is iterative and each iteration (i.e. adaptation) computes a least square fitting for each frame, followed by an interpolation step. To measure the cost of each adaptation, the total time required to complete a calculation T is divided by the number of adaptations N. Furthermore, each least square fitting depends on the size of the matrix involved, whose size is proportional to the number K of fitting functions and the number of samples S_N. Thus, the matrix size is proportional to $K \cdot S_N$. In Figure 3.3 it is shown how the cost of each iterative step scales with the size of the matrix. It is clear that as the size of the matrix grows, MATLAB is better optmized than Armadillo.

Chapter 4

Final remarks

In this project, a novel software was developed for speech processing. The software optimizes and estimates the parameters for an extended adaptive quasi harmonic model. The original software was written in MATLAB.

4.1 Why C++?

A new version of an existing MATLAB software written in C++ has been presented. The choice of moving away from MATLAB is mostly due to flexibility:

1. Most modern programming languages can incorporate functions written in C++. Among those, MATLAB, via mex files, and Python. The advantage is that it is easier to maintain and fix bugs in a function in C++ and than use it where needed than having different versions for different programming languages.

2. C++ is an open standard implemented by several compilers. In most cases, it is possible to compile and run a code without having to deal with expensive licenses, like in the case of MATLAB. In addition, working with MATLAB very often requires buying extra packages, which adds to the cost of the project, whereas C++ has some highly optimized libraries available for free. In this project, for instance, the free library Armadillo was employed.

4.2 Further work

C++ programs are compiled to machine code, opening the door to greater optimization than interpreted languages such as MATLAB. One of the main advantages is the use of parallel computing. Even though this was not attempted in this project, the code could in principle take advantage of modern multicores and multiCPUs architectures. This may become very important, for instance, for server applications where multiple users with large files can use the software at the same time.

Another aspect that can be improved upon is the way frames are built. Instead of using a fixed duration for each frame, the duration can vary in an adaptive fashion.

As mentioned in Section 3.1, MATLAB and C++/Armadillo are currently using different interpolation schemes. This could in principle be remedied by adding a spline option for the C++ code.

In Section 2.2, it was mentioned that both the MATLAB as well as the C++ code need a set of input parameters generated by other functions or programs. At the current stage, these files are still produced using a MATLAB code that works as a pitch detector. The data is stored in a .mat file. This file can be readily used by the MATLAB code, but needs to be converted into a text file to be accessible to the C++ code. In a future effort, a pitch detector will be developed in C++ so that the data can be easily transferred to the current code.

Appendix A

Source Code

In this appendix, the source code is listed.

A.1 audio.h

This file defines the struct audio.

```
#pragma once
#ifndef AUDIO_H
#define AUDIO_H
#include <string>
#include <vector>
//#include <RcppArmadillo.h>
#include <armadillo>
using namespace arma;
//using namespace Rcpp;
using std::string;
using std::vector;

template<class T>
struct audio{
    unsigned int fs;
    T s;

    //~audio(){
    //    s.clear();
    //}

    //operator SEXP() const {
    //    return
```

```
        Rcpp::wrap(List::create(_["fs"]=fs,_["s"]=wrap(s)));
    //}
};

audio<colvec> WavRead(string);

#endif
```

A.2 AdditionalFunctions.cpp

This files includes a set of functions used throughout the program.

```cpp
#include <armadillo>
#include "Functions.h"

using namespace arma;
using namespace std;

vec cos_win(const uword N, const vec& a)
{
    vec h(N);
    for (uword i = 0; i < N; i++)
    {
        h[i] = a[0] - a[1] * cos(1.0 * datum::pi * i / (N - 1)) +
            a[2] * cos(2.0 * datum::pi * i / (N - 1)) -
            a[3] * cos(3.0 * datum::pi * i / (N - 1)) +
            a[4] * cos(4.0 * datum::pi * i / (N - 1));
    }
    return h;
}

vec hamming(const uword N)
{
    vec a = zeros<vec>(5);
    a[0] = 0.54;
    a[2] = -0.46;
    return cos_win(N, a);
}

vec blackman(const uword N)
{
    vec a = zeros<vec>(5);
    a[0] = 0.42; // 7938/18608.0
    a[2] = -0.5; // 9240/18608.0
    a[4] = 0.08; // 1430/18608.0
    return cos_win(N, a);
}

vec hanning(const uword N)
{
    vec h(N);
    for (uword i = 0; i < N; i++)
    {
        h[i] = 0.5 - 0.5 * std::cos(datum::pi * (i + 1) / (N + 1));
    }
    return h;
}

vec unwrap1(const vec& x)
{
   arma::vec P;
   double pacc = 0, pdiff = 0;
   const double thr = datum::pi;
   P.copy_size(x);
   P(0) = x(0);
   for (unsigned int r = 1; r < x.n_rows; r++)
   {
      pdiff = x(r) - x(r - 1);
      if (pdiff >= thr)
         pacc += -2 * datum::pi * int(round(fabs(pdiff) / (2 *
             datum::pi)));
      if (pdiff <= -thr)
         pacc += 2 * datum::pi * int(round(fabs(pdiff) / (2 *
             datum::pi)));
      P(r) = pacc + x(r);
   }
   return P;
}

cx_vec spectrum(const colvec& x, const vec& W)
{
    cx_vec Pxx(x.size());
    double wc = sum(W);        // Window correction factor
    Pxx = fft(x % W) / wc; // FFT calc
    return Pxx;
}

cx_mat specgram_cx(const colvec& x, const uword Nfft = 512, const
    int Fs = 1000,
 const vec W = hamming(512), const uword Noverl = 256)
{
    cx_mat Pw;

    // Def params
    uword N = x.size();
    uword D = Nfft - Noverl;
    uword m = 0;
    if (N > Nfft)
    {
        colvec xk(Nfft);
        //vec W(Nfft);

        //W = hamming(Nfft);
        uword U = static_cast<uword>(floor((N - Noverl) /
            double(D)));
        Pw.set_size(Nfft, U);
        Pw.zeros();

        // Avg loop
        for (uword k = 0; k < N - Nfft; k += D)
        {
            xk = x.rows(k, k + Nfft - 1); // Pick out chunk
            Pw.col(m++) = spectrum(xk, W);     // Calculate spectrum
        }
    }
    else
    {
        //vec W(N);
        //W = hamming(N);
        Pw.set_size(N, 1);
        Pw = spectrum(x, W);
    }
    return Pw;
}

mat specgram(const colvec & x, const uword Nfft = 512, const int
    Fs = 1000,
const vec W = hamming(512), const uword Noverl = 256)
{
    cx_mat Pw;
    mat Sg;
    Pw = specgram_cx(x, Nfft, Fs, W, Noverl);
    Sg = real(Pw % conj(Pw)); // Calculate power spectrum
    return Sg;
}


bool isPrime(int n)
```

```
{
    // Since 0 and 1 is not prime
    // return false.
    if (n == 1 || n == 0)return false;

    // Run a loop from 2 to
    // square root of n.
    for (int i = 2; i * i <= n; i++)
    {
        // If the number is divisible by i,
        // then n is not a prime number.
        if (n % i == 0)return false;
    }

    // Otherwise n is a prime number.
    return true;
}

rowvec primes(unsigned const int n)
{
    rowvec res = {};
    for (double i = 2; i <= n; i++)
        if (isPrime(i))
            res = join_rows(res, rowvec{ i });
    return res;
}
```

A.3 AQHMphase_interp.cpp

This files contains a function for interpolating the phase $\phi(t)$.

```
//#include <RcppArmadillo.h>
#include <armadillo>
#include "Functions.h"

using namespace arma;

mat AQHMphase_interp(colvec fm_hat, colvec pm_hat, uvec idx, const
    string method)
{
   //
   // Phase interpolation using either integration of instantaneous
       frequency,
   // cubic interpolation, or phase decomposition and interpolation
   //
   // (c) Yannis Pantazis, 2009 (author)
   // mail: pantazis@csd.uoc.gr
   //
   // (c) George Kafentzis, 2012 (revised and updated)
   // mail: kafentz@csd.uoc.gr
   //

   mat pm_final(fm_hat.n_elem, 1);
   colvec pm_inst, t, ft;
   rowvec a1, a2, a3, a4;
   mat a;
   const double pi2 = datum::pi * 2;

   pm_final.zeros(); //<== Added

   for (unsigned int i = 0; i < idx.n_elem - 1; ++i)
   {
      int idx_i = idx(i), idx_i_1 = idx(i + 1);
      t = myRange<colvec>(0, idx_i_1 - idx_i, 1); // Actually this
          is t'
      if (method == "integr")
      {
         // intergation of frequency
         pm_inst = filter({ 1 }, { 1 ,-1 },
         fm_hat(span(idx_i, idx_i_1))); // <== Fixed: missing comma
         // correct phase value at idx_i
         pm_inst += pm_hat(idx_i) - pm_inst(0);
         // repmat({pm_hat(idx_i) - pm_inst(0)}, pm_inst.n_elem, 1)

         // correction for the phase value at idx_i_1
         double M = round((pm_inst(pm_inst.n_elem - 1) -
             pm_hat(idx_i_1))
          / pi2);
         double er = datum::pi * (pm_inst(pm_inst.n_elem - 1) -
          pm_hat(idx_i_1) - pi2 * M) / (2 * (idx_i_1 - idx_i));
         ft = sin(datum::pi * t / (idx_i_1 - idx_i)); // Actually
             this is ft'
         pm_inst -= filter({ 1 }, { 1, -1 }, ft * er); //<== Fixed:
             missing comma

         // save the interpolated phase
         pm_final.rows(idx_i, idx_i_1) = pm_inst;
      }
      else if (method == "cubic")
      {
         unsigned int Dt = idx_i_1 - idx_i;
         unsigned int lt = t.n_elem;

         mat::fixed<2, 2> A = {                  // <== Converted int
             into double
            {3.0 / (Dt * Dt), -1.0 / Dt},
            {-2.0 / (Dt * Dt * Dt), 1.0 / (Dt * Dt)} };

         a1 = pm_hat(idx_i); // <== Fixed
         a2 = fm_hat(idx_i); // <== Fixed
         double M = round((pm_hat(idx_i) + fm_hat(idx_i) *
         Dt - pm_hat(idx_i_1) + (fm_hat(idx_i_1) - fm_hat(idx_i)) *
             Dt / 2)
          / pi2); // <== Fixed
         vec v = { {pm_hat(idx_i_1) + pi2 * M - (pm_hat(idx_i) +
         fm_hat(idx_i) * Dt)},{fm_hat(idx_i_1) - fm_hat(idx_i)} };
             // <== Added
         a = A * v; // <== Added
         a3 = { a(0,0) }; // <== Fixed
         a4 = { a(1,0) }; // <== Fixed

         pm_final.rows(idx_i, idx_i_1) = repmat(a1, lt, 1) + t * a2
             +
          square(t) * a3 + pow(t, 3) * a4;
      } // }else if(method == "other"){
      //    //Alternative phase implementation
      //    pm_inst = filter(1, {1 -1}, fm_hat(idx(i):idx(i+1)));
      //    //pm_tmp2 = pm(idx1,k) + pm_temp(idx1(j),k);
      //    pm_inst = pm_inst + repmat(pm_hat(idx(i))-pm_inst(1),
       //pm_inst.n_elem, 1);
      //    pm_inst = interp1([idx(i) idx(i+1)], pm_inst([1 end]),
       //[idx(i):idx(i+1)]', 'spline');
      //    pm_final(idx(i):idx(i+1)) = pm_inst;
      // }
    }
    return pm_final.rows(idx(0), idx(idx.n_elem - 1)); // <== Fixed
  }
```

A.4 compLSfm_akbk.cpp

This file contains the function for the least square fitting of the aQHM.

```cpp
//#include <RcppArmadillo.h>
#include <armadillo>
#include "DataStructures.h"
#include "Functions.h"

using namespace arma;

//[[Rcpp::export]]
compLSfmAkbk compLSfm_akbk(colvec s, mat fm, colvec win, unsigned
    int fs)
{
   const int len = fm.n_rows, K = fm.n_cols, N = (len - 1) / 2;
   colvec n = linspace<colvec>(-N, N, 2 * N + 1);
   mat f_an(K, len, fill::zeros);
   rowvec tmp(len); // <== Fixed: size
   colvec::fixed<1> b{ 1 };
   colvec::fixed<2> a{ 1, -1 };

   for (int k = 0; k < K; ++k)
   {
      tmp = filter(b, a, fm.col(k)).t();
      f_an.row(k) = tmp - tmp(N);
   }


   mat t = (2.0 * datum::pi / fs) * f_an.t();
   cx_mat E(cos(t), sin(t));

   E = join_horiz(E, repmat(n, 1, K) % E);
   cx_mat Ew = repmat(win, 1, 2 * K) % E;
   cx_mat R = Ew.t() * Ew;

   compLSfmAkbk str{}; // <== Moved

   if (rcond(R) <= 1e-10) { // <== Changed
      cerr << "CAUTION: Bad condition of matrix.\n";
      str.ak.zeros(K); // <== Added
      str.bk.zeros(K); // <== Added
      str.y.zeros(K); // <== Added
      str.SNR = 0.0;  // <== Added
    return str;      // <== Changed
  }

  cx_mat x = solve(R, (Ew.t() * (win % s(span(0, win.n_elem -
     1)))));

  str.ak = x(span(0, K - 1), 0);
  str.bk = x(span(K, 2 * K - 1), 0);
  str.y = E * join_vert(str.ak, str.bk);
  str.SNR = 20.0 * log10(stddev(s) / stddev(s(span(0, str.y.n_elem
     - 1)) - real(str.y)));
  return str;
}
```

A.5 compLSamfm_akbk.cpp

This file contains the function for the least square fitting of the eaQHM.

```
//#include <RcppArmadillo.h>
#include <armadillo>
#include "DataStructures.h"
#include "Functions.h"

using namespace arma;

//[[Rcpp::export]]
compLSamfmAkbk compLSamfm_akbk(colvec s, mat am, mat fm, colvec
    win, unsigned int fs)
{
   const int len = fm.n_rows, K = fm.n_cols, N = (len - 1) / 2;
   constexpr double eps = 1e-04;
   colvec n = linspace<colvec>(-N, N, 2 * N + 1);
   mat f_an(K, len, fill::zeros);
   rowvec tmp(len); // <== Fixed: size
   colvec::fixed<1> b{ 1 };
   colvec::fixed<2> a{ 1, -1 };

   for (int k = 0; k < K; ++k)
   {
      tmp = filter(b, a, fm.col(k)).t();
      f_an.row(k) = tmp - tmp(N);
   }


   mat t = (2.0 * datum::pi / fs) * f_an.t();
   cx_mat E(cos(t), sin(t));

   E = ((am + eps) / (repmat(am.row(N), 2 * N + 1, 1) + eps)) % E;

   E = join_horiz(E, repmat(n, 1, K) % E);
   cx_mat Ew = repmat(win, 1, 2 * K) % E;
   cx_mat R = Ew.t() * Ew;


   compLSamfmAkbk str{}; // <== Moved

   if (rcond(R) <= 1e-15) { // <== Changed
      cerr << "CAUTION: Bad condition of matrix.\n";
    str.ak.zeros(K); // <== Added
    str.bk.zeros(K); // <== Added
    str.SNR = 0.0;  // <== Added
    return str;     // <== Changed
  }

  cx_mat x = solve(R, (Ew.t() * (win % s(span(0, win.n_elem -
     1)))));

  str.ak = x(span(0, K - 1), 0);
  str.bk = x(span(K, 2 * K - 1), 0);
  mat realy = real(E * join_vert(str.ak, str.bk));
  str.SNR = 20.0 * log10(stddev(s) / stddev(s(span(0, realy.n_elem
     - 1)) - realy));
  return str;
}
```

A.6 DataStructures.h

This file defines several structures.

```
#pragma once
#ifndef DATASTRUCTURES_H

//#include <RcppArmadillo.h>
#include <armadillo>
#include <string>
#include "audio.h"
using namespace arma;
using namespace std;
//using namespace Rcpp;

struct compLSamfmAkbk {
    cx_mat ak;
    cx_mat bk;
    double SNR;

    compLSamfmAkbk(std::initializer_list<bool> empty) {}
    compLSamfmAkbk(cx_mat ak,cx_mat bk,double SNR) : ak(ak),
       bk(bk), SNR(SNR){}

    /*operator SEXP() const {
      return Rcpp::wrap(Rcpp::List::create( Rcpp::_["ak"]=ak,
          Rcpp::_["bk"]=bk,Rcpp::_["SNR"]=SNR));
    }*/
};

struct compLSfmAkbk {
  cx_mat ak;
  cx_mat bk;
  cx_mat y;
  double SNR;

  compLSfmAkbk(std::initializer_list<bool> empty) {}
  compLSfmAkbk(cx_mat ak,cx_mat bk, cx_mat y,double SNR) : ak(ak),
      bk(bk), y(y), SNR(SNR){}

  /*operator SEXP() const {
    return Rcpp::wrap(Rcpp::List::create(
        Rcpp::_["ak"]=ak,Rcpp::_["bk"]=bk,
        Rcpp::_["y"]=y,Rcpp::_["SNR"]=SNR));
  }*/
};

struct icompLSakbk
{
  cx_mat ak;
  cx_mat bk;
  cx_rowvec df;
  double SNR;

  icompLSakbk(std::initializer_list<bool> empty) {}
  icompLSakbk(cx_mat ak,cx_mat bk, cx_rowvec df,double SNR) :
      ak(ak), bk(bk), df(df), SNR(SNR){}

  /*operator SEXP() const {
    return Rcpp::wrap(Rcpp::List::create(
        Rcpp::_["ak"]=ak,Rcpp::_["bk"]=bk,Rcpp::_["df"]=df,
        Rcpp::_["SNR"]=SNR));
  }*/
};

struct GetLin {
  mat value;
  unsigned int previ;
  unsigned int nexti;
  double g;

  GetLin(mat value = {}, unsigned int previ = 0, unsigned int nexti
      = 0, double g = 0) :
    value(value), previ(previ), nexti(nexti), g(g){}

  /*operator SEXP() const {
    return Rcpp::wrap(Rcpp::List::create(
        Rcpp::_["value"]=value,Rcpp::_["previ"]=previ,
        Rcpp::_["nexti"]=nexti,Rcpp::_["g"]=g));
  }*/
};

struct opt_swipep
{
   float f0min = 50;
   float f0max = 1000;
   float dt = 0.001f;
   float dlog2p = 1 / 48;
   float dERBs = 0.1f;
   float woverlap = 0.5;
   float sTHR = -INFINITY;
    vector<string> fmapin = {"wav", "fs", "snd"};
    vector<string> fmapout = {"f0s", "f0_swipep"};

};

struct optionFile
{
    int fl = 1;
    int frc = 0;
    int fb = 1;
    int ea = 1;
    int AIR = 0;
    int adpt = 10;
    int lpfilt = 0;
    int NoP = 3;
    int SWIPEP = 1;
    int YIN = 0;
    int num_part = 0;

};

struct mod_s
{
    float ts = 1.0;
    float ps = 1.0;
};

struct Vstruct //: (other stuff)
{
    colvec s;// speech signal
    unsigned fs;// sampling frequency
    colvec qh;// reconstructed deterministic part
    colvec env;// noise envelope
};

struct Pstruct
{
    double ti; //: analysis time instants
    bool isS; //: is speech
    bool isV; //: is voiced
};

struct D
{
    cx_mat am_hat;
    cx_mat fm_hat;
    cx_mat pm_hat;
    cx_colvec s_hat;
    float SRER;

};

struct Dstruct
{
    double ti; //: analysis time instants
    bool isS  ; //: is speech
    bool isV  ; //: is voiced
    double a0; //: DC component
    rowvec ak; //: amplitude vector
    rowvec fk; //: frequency vector
    rowvec pk; //: phase vector

    bool fb;
    int Kmax; // dim
    int Fmax; // freq
    string filename;
    bool fl;

};

struct Sstruct
{
    int ti; //analysis time instants
    bool isS ;  //is speech
    vec ap ;   //AR coefficients
    rowvec env_ak ;  //amplitude vector(for time - envelope)
    rowvec env_fk ;  //frequency vector(for time - envelope)
    rowvec env_pk ;  //phase vector(for time - envelope)

};

struct Mstruct
```

```
{
    mat fm;
    mat fm_sc;
    mat am;
    mat am_sc;
    mat ph;
    mat ph_sc;

};

// struct AQHMVUV {
//   double ti;
//   double isS;
//   double isV;
//
//   AQHMVUV(std::initializer_list<bool> empty) {}
//   AQHMVUV(double ti=0,double isS=0, double isV=0) : ti(ti),
    isS(isS), isV(isV) {}
//
//   operator SEXP() const {
//     return Rcpp::wrap(Rcpp::List::create(
    Rcpp::_["ti"]=ti,Rcpp::_["isS"]=isS,Rcpp::_["isV"]=isV));
//   }
// };



struct eaQHManalysisStruct
{
    vector<Dstruct> D;
    vector<Sstruct> S;
    Vstruct V;
    rowvec SRER;
    mat aSNR;

};


#endif
```

A.7 eaQHManalysis.cpp

This is the main program for performing the fitting of the eaQHM

```
#include <armadillo>
#include "DataStructures.h"
#include "Functions.h"
#include "eaQHMinput.h"
using namespace arma;
using namespace std;

//function [D, S, V, SRER, aSNR] = eaQHManalysis(speechFile, step,
    gender, opt)
eaQHManalysisStruct eaQHManalysis(string speechFile, unsigned int
    step, string gender, optionFile opt = {})
{
   //% %%%%%%%%%%%% (extended) Adaptive Quasi-Harmonic Analysis of
       Speech %%%%%%%%%%%%%
   //% EAQHMANALYSIS Speech analysis using the extended adaptive
       Quasi-Harmonic
   //% Model.
   //%   [D, S, V, SRER, aSNR] = eaQHManalysis(speechFile, step,
       gender, opt)
   //%   decomposes speech into AM-FM components according to the
       eaQHM model.
   //%
   //%   Implementation based on publications:
   //%   -------------------------------------
   //%   1] Yannis Pantazis, Georgios Tzedakis, Olivier Rosec and
       Yannis Stylianou,
   //%      Analysis/Synthesis of Speech based on an Adaptive
       Quasi-Harmonic plus Noise Model,
   //%      In IEEE International Conference on Acoustics, Speech,
       and Signal Processing (ICASSP), 2010
   //%   2] George Kafentzis, Olivier Rosec, and Yannis Stylianou,
   //%      Robust Adaptive Sinusoidal Analysis and Synthesis of
       Speech,
   //%      In IEEE International Conference on Acoustics, Speech,
       and Signal Processing (ICASSP) 2014
   //%
   //% ----INPUT PARAMETERS----
   //%    speechFile: speech wav file
   //%    step: step size (in samples)
   //%    gender: gender of speaker (male/female)
   //%   opt: options (B for binary, N for numerical)
   //%   opt.fl  : Full waveform length analysis (B)
   //%   opt.frc : Frequency correction mechanism (B) (no longer
       available)
   //%   opt.fb  : Full band analysis-in-voiced-frames flag (B)
   //%   opt.ea  : Extended aQHM (B)
   //%   opt.AIR : Adaptive Iterative Refinement of f0 (B)
   //%   opt.adpt: Adaptation number (default : 6) (N)
   //%   opt.lpfilt  : Lowpass filtering @ 30 Hz (preprocess) B)
   //%   opt.NoP     : Number of analysis window size, in pitch
       periods (N) (default: 3)
   //%   opt.SWIPEP  : SWIPEP pitch estimator (B)
   //%   opt.YIN     : YIN pitch estimator (B)
   //%   opt.num_part : Number of partials (N)
   //%
   //%  ----OUTPUT PARAMENTERS----
   //%  Structure D: (deterministic)
   //%    D.ti: analysis time instants
   //%    D.isS: is speech
   //%    D.isS: is voiced
   //%    D.a0: DC component
   //%    D.ak: amplitude vector
   //%    D.fk: frequency vector
   //%    D.pk: phase vector
   //%
   //%  Structure S: (stochastic - optional)
   //%    S.ti: analysis time instants
   //%    S.isS: is speech
   //%    S.ap: AR coefficients
   //%    S.env_ak: amplitude vector (for time-envelope)
   //%    S.env_fk: frequency vector (for time-envelope)
   //%    S.env_pk: phase vector (for time-envelope)
   //%
   //%  Structure V: (other stuff)
   //%    V.s: speech signal
   //%    V.fs: sampling frequency
   //%    V.qh: reconstructed deterministic part
   //%    V.env: noise envelope
   //%
   //% (c) George Kafentzis, 2013
   //%
   //% Version 5.0
   //%
```

```
//% mail: kafentz@csd.uoc.gr
//%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%
//
//% Just to hide the log-of-zero warning - increases speed
//warning off MATLAB:log:logOfZero

//load('Parameters.mat');

// Here, we set some default values:
unsigned int const_value_adpt = 10;
unsigned int const_value_Fmax = 7800;
unsigned int const_value_fs = 16000;
string deterministic_file_name = "sa19_s.txt";
string f0_file_name = "f0s.txt";
string f0sin_file_name = "f0sin.txt";
string P_file_name = "P.txt";
gender = "female";
//int i = 794;
int Kmax = 68;
unsigned int len = 63488;
int NoP = 3;
opt = {};
int opt_pitch_f0min = 120;
opt_swipep opt_swipep = {};
int p_step = 80;

speechFile = "SA19.wav";
step = 15;


// Read Data from file
ifstream inputFile("eaQHMinput.in");
if (inputFile.is_open()) {
   eaQHMinput* dataFromFile;

try {
      dataFromFile = new eaQHMinput(inputFile);
   }
   catch (const std::invalid_argument& e) {
      cerr << "Error: " << e.what() << endl;
      cerr << "Using Default Values "<< endl;
   }

   //Assign values:
   const_value_adpt = dataFromFile->adpt;
   deterministic_file_name = dataFromFile->filename_det;
   f0_file_name = dataFromFile->filename_f0;
   f0sin_file_name = dataFromFile->filename_f0sin;
   P_file_name = dataFromFile->filename_P;
   speechFile = dataFromFile->speechFile;
   const_value_Fmax = dataFromFile->Fmax;
   const_value_fs = dataFromFile->fs;
   gender = dataFromFile->gender;
   //i = dataFromFile->i;
   Kmax = dataFromFile->Kmax;
   len = dataFromFile->len;
   NoP = dataFromFile->NoP;
   opt_pitch_f0min = dataFromFile->opt_pitch_f0min;
   p_step = dataFromFile->p_step;
   step = dataFromFile->step;

   inputFile.close(); // Finished with file
   delete dataFromFile; // We do not need it any longer
}
else {
   cout << "Unable to open input file " << endl;
   cerr << "Using Default Values " << endl;
}

//First, let's set the constants:
const unsigned int adpt = const_value_adpt;
const unsigned int Fmax = const_value_Fmax;
const unsigned int fs = const_value_fs;

//  Then let's read the datafiles and store their values:
mat deterministic_part;
mat f0s;
mat f0sin;
mat P;

f0s.load(f0_file_name);
f0sin.load(f0sin_file_name);
P.load(P_file_name);
deterministic_part.load(deterministic_file_name);
```

```
// Seeting up the V structure
mat s = deterministic_part;
Vstruct V;
V.s = s;
V.fs = fs;
mat wav = deterministic_part;

//
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%
//%
//%                                 Information to the user
//%
//%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//fprintf(1, '\t\t e a Q H M A N A L Y S I S
    \n\t\t-----------------------------\n');
cout << "\t\t e a Q H M  A N A L Y S I S
    \n\t\t-----------------------------\n";

//fprintf(1, 'Processing filename: %s\n', speechFile);
cout << "Processing filename: " << speechFile << '\n';

//fprintf(1, 'Maximum Voiced Frequency: %d Hz\n', Fmax);
cout << "Maximum Voiced Frequency : " << Fmax << " Hz\n",

   //fprintf(1, 'Analysis step size: %d samples (%f sec)\n',
      step, step/fs);
   printf("Analysis step size: %d samples( %lf sec)\n", step,
      (double)step / (double)fs);
   //cout << "Analysis step size: " << step << " samples(" <<
      (float)step / (float)fs << " sec)\n";



//if opt.ea == 1
//    fprintf(1, 'Full adaptation (eaQHM).\n');
//else
//    fprintf(1, 'Phase adaptation (aQHM).\n');
//end
```

```
if (opt.ea == 1)
   cout << "Full adaptation (eaQHM).\n";
else
   cout << "Phase adaptation (aQHM).\n";

//if opt.frc == 1
//    fprintf(1, 'Frequency matching is on.\n');
//else
//    fprintf(1, 'Frequency matching is off.\n');
//end
if (opt.frc == 1)
   cout << "Frequency matching is on.\n";
else
   cout << "Frequency matching is off.\n";

//if opt.AIR == 1
//    fprintf(1, 'Adaptive Iterative Refinement of SWIPEP(f_0)
    is on.\n');
//elseif opt.SWIPEP == 1
//    fprintf(1, 'SWIPEP pitch estimation is on.\n');
//elseif opt.YIN == 1
//    fprintf(1, 'YIN pitch estimation is on.\n');
//else
//    error('This should never happen. You have not selected a
    pitch estimator!\n');
//end
if (opt.AIR == 1)
   cout << "Adaptive Iterative Refinement of SWIPEP(f_0) is
       on.\n";
else if (opt.SWIPEP == 1)
   cout << "SWIPEP pitch estimation is on.\n";
else if (opt.YIN == 1)
   cout << "YIN pitch estimation is on.\n";
else
   throw "This should never happen. You have not selected a
       pitch estimator!\n";

//if opt.fl == 1
//    fprintf(1, 'Full waveform length analysis is
    performed.\n');
//else
//    fprintf(1, 'Only voiced parts analysis is performed. Noise
    will be added to model unvoiced speech.\n');
```

```
//end
if (opt.fl == 1)
   cout << "Full waveform length analysis is performed.\n";
else
   cout << "Only voiced parts analysis is performed. Noise will
       be added to model unvoiced speech.\n";

//fprintf('Maximum number of adaptations allowed: %d\n', adpt);
cout << "Maximum number of adaptations allowed: " << adpt <<
    '\n';

//fprintf('Gender specified: %s\n\n', gender);
cout << "Gender specified: " << gender << "\n\n";

//
//
//
//
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%
//%
//%                        A N A L Y S I S   S T A R T S   H E R E
//%
//%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//
//
//
//% ---------------------------------------------
    ----------------------------
//% ---------------------------------------------
    ----------------------------
//%                       DETERMINISTIC PART
//% ---------------------------------------------
    ----------------------------
//% ---------------------------------------------
    ----------------------------

//% Analysis time instants
//ti = 1:step:len;
uvec ti = regspace<uvec>(0, step, len - 1);
```

```
//% Number of analysis time instants
//No_ti = length(ti);
const uword No_ti = ti.n_elem;

//% Initializations
//N = zeros(No_ti,1);
mat N = zeros(No_ti, 1);

//f0_val = zeros(len, 1);
mat f0_val = zeros(len, 1);

//fm_cur = zeros(len, Kmax);
mat fm_cur = zeros(len, Kmax);

//
//aSNR = zeros(adpt, No_ti);
mat aSNR = zeros(adpt, No_ti);
//
//% Repeat adaptations
//for m = 0:adpt
mat am_cur = zeros(len, Kmax);
vector<Dstruct> D(No_ti);
//rowvec SRER;
rowvec SRER = zeros<rowvec>(adpt + 1); // <== Diff: Initialized

mat a0_fin;
mat am_fin;
mat fm_fin;
mat pm_fin;
for (int m = 0; m < adpt + 1; m++)
{
   //   fprintf(1, 'Please wait. Adapting...\n');
   cout << "Please wait. Adapting...\n";

   //   a0_hat = zeros(len, 1);
   mat a0_hat = zeros(len, 1);

   //   am_hat = zeros(len, Kmax);
   mat am_hat = zeros(len, Kmax);

   //   fm_hat = zeros(len, Kmax);
   mat fm_hat = zeros(len, Kmax);
```

```
//    pm_hat = zeros(len, Kmax);
mat pm_hat = zeros(len, Kmax);

//    % For every analysis instant...
//    for i = 1:No_ti
for (unsigned int i = 0; i < No_ti; i++)
{
  //       D(i).ti = ti(i);
  D[i].ti = ti(i);

  //       % Pitch estimates start @ sample 240: do the
     analysis!
  //       if ti(i) > 480 && ti(i) < len-480
  if (ti(i) >= 480 && ti(i) < len - 480)
  {
    //           p_i = ti(i)/p_step;
    double p_i = (double)(ti(i) + 1) / (double)p_step;

    //           pf_i = floor(p_i);
    int pf_i = floor(p_i);

    //           % If two consecutive frames are voiced,
       mark the instant as such
    //           if P(pf_i).isV && P(pf_i+1).isV
    if (P(pf_i, 2) == 1 && P(pf_i + 1, 2) == 1)
    {
      //               D(i).isS = 1;
      D[i].isS = 1;

      //               D(i).isV = 1;
      D[i].isV = 1;

      //               % analysis of frame - QHM/iQHM BEGINS
         HERE! (case: m == 0)
      //               if m==0
      //int K;
      uword K;  // <== Diff: data type
      cx_mat ak;
      cx_mat bk;
      cx_rowvec df;
      double f0;
      if (m == 0)
```

```
{
  //                  % weighting frequency
  //                  lamda = p_i-pf_i;
  double lamda = p_i - pf_i;

  //                  f0 = (1-lamda)*f0sin(pf_i, 2)
     + lamda*f0sin(pf_i+1, 2);
  f0 = (1 - lamda) * f0sin(pf_i - 1, 1) + lamda *
     f0sin(pf_i, 1);

  //                  K = min(Kmax, floor(Fmax/f0));
  K = min(Kmax, (int)floor(Fmax / f0));

  //                  fk = (-K:K)*f0;
  vec fk = regspace(-int(K), int(K)) * f0; // <==
     Diff: K explicitly converted to int


  //
  //                  % window definition
  //                  N(i) = max(120,
     round(NoP/2*fs/f0)); % half of analysis window
  int tmp = max(120, (int)round((double)NoP / 2 * fs
     / f0));
  N(i, 0) = tmp;

  //                  n = -N(i):N(i); % time window
     where everything is watched through
  vec n = regspace(-tmp, tmp);

  //                  win = blackman(2*N(i)+1); %
     analysis window
  vec win = blackman(2 * N(i, 0) + 1); // <== Diff:
     N(i) -> N(i,0)

  //
  //                  % QHM (liter > 0 => iQHM)
  //                  [ak, bk, df] =
     icompLS_akbk(s(ti(i)+n), fk, win, fs, 0);
  icompLSakbk tempStruct =
     icompLS_akbk(s.rows(conv_to<uvec>::from(ti(i) +
     n)), fk, win, fs, 0);
  ak = tempStruct.ak;
```

```
    bk = tempStruct.bk;
    df = tempStruct.df;
    double SNR = tempStruct.SNR;

    //                  % either way, put f0 as f0
       value for this time instant
    //                  f0_val(ti(i)) = f0;
    f0_val(ti(i)) = f0;
}
//
//                  % ----------------------
   ------------------------------
//              else % aQHM analysis BEGINS HERE!!!!
   (case: m ~= 0)
//                  % ----------------------
   ------------------------------
else
{
    //                  n = -N(i):N(i); % window
       samples
    //uvec n = regspace<uvec>(-N(i, 0), N(i, 0));
    ivec n = regspace<ivec>(-N(i, 0), N(i, 0)); // <==
       Diff: ivec not uvec(-N has a sign...)

            //                  win =
               hamming(2*N(i)+1); % window
    vec win = hamming(2 * N(i, 0) + 1); // <== Diff:
       N(i) -> N(i,0)
    //
    //                  % What is going on here? Let's
       see...
    //                  % This is the part were
       frequency trajectories are born or die, in a
       frame!!
    //
    //                  % At first, fm_cur == 0 for
       all values.
    //                  % Later, it gets the indices
       of frequencies that
    //                  % are not zero at the center
       of the analysis
    //                  idx = find(fm_cur(ti(i),:) ~=
       0);
```

```
uvec idx = find(fm_cur.row(ti(i)) != 0);

//                  if isempty(idx) % at first, it
    is empty (for m == 1)
if (idx.is_empty())
{
   //                        idx = 1;
   idx = { 0 };

   //                        fm_cur(ti(i), idx) =
       140; % set the first harmonic value at 100
       Hz (why?) for the center of this frame - not
       the best practice
   fm_cur(ti(i), 0) = 140;

   //                        am_cur(ti(i), idx) =
       10^-4; % set the amplitude of the first
       harmonic at a small value - not the best
       practice
   am_cur(ti(i), 0) = pow(10, -4);
   //                    end
}
//
//                  % !!fm_tmp has the nonzero
    frequency trajectories in a
//                  % frame!!
//                  fm_tmp = fm_cur(ti(i)+n, idx);
    % At first, fm_tmp goes 140 in the center and
    zero elsewhere, and has size [2N(i)+1, 1]
//                                               %
    Later, it gets the trajectory in the frame of
    corresponding frequencies that are not zero.
//                                               %
    Size [2N(i)+1, idx]
ivec tmp_tin = ti(i) + n; // <== Diff: need ivec
    to do math instead of uvec.
mat fm_tmp = fm_cur(conv_to<uvec>::from(tmp_tin),
    idx); // <== Diff: back to uvec after math.

//                  am_tmp = am_cur(ti(i)+n, idx);
    % Get the amplitude trajectory in the frame of
    corresponding frequencies that are not zero
//                                               %
```

```
    Size [2N(i)+1, idx]
mat am_tmp = am_cur(conv_to<uvec>::from(tmp_tin),
    idx); // <== Diff: back to uvec after math.
//
//
//                  K = idx(end); % K is the index
    of highest nonzero frequency
//uword K = idx(idx.n_elem - 1);
K = idx(idx.n_elem - 1); // <== Diff: Removed
    uword. This is the same K as before.

         //
         //                  % For all of these
             nonzero frequencies...
         //                  for k = 1:length(idx)
int L;
for (unsigned int k = 0; k < idx.n_elem; k++)
{
   //                      L = length(fm_tmp(:,
       k)); %... set L = length of k-th frequency
       trajectory (size of frame, actually?)
   L = fm_tmp.col(k).n_elem;

   //                      zr = find(fm_tmp(:, k)
       == 0); % ... find zero values in this
       trajectory
   uvec zr = find(fm_tmp.col(k) == 0);

   //                      nzr = find(fm_tmp(:, k)
       ~= 0); % ... find nonzero values in this
       trajectory
   uvec nzr = find(fm_tmp.col(k) != 0);

   //
   //                      % New frequency was
       born! Hurraaaay! :)
   //                      if ~isempty(zr) && zr(1)
       == 1 % In the case where there's at least
       one zero value, and that is the at index ==
       1...
   if (!zr.is_empty() && zr(0) == 0) // <== Diff:
       == 0, not == 1
   {
```

```
   //                          fm_tmp(zr(1), k) =
      fm_tmp(nzr(1), k); % ... replace this
      index value with the first nonzero one,
      for the k-frequency...
   fm_tmp(zr(0), k) = fm_tmp(nzr(0), k);

   //                          am_tmp(zr(1), k) =
      am_tmp(nzr(1), k); % same for amps
   am_tmp(zr(0), k) = am_tmp(nzr(0), k);

   //                          nzr = [zr(1);
      nzr]; % ... and a new nonzero index
      matrix is formed with zr(1) == 1, and the
      other nonzero index values
   nzr = join_cols(uvec{ zr(0) }, nzr);

   //                      end
}
//                      % A frequency has died!
   Call a priest to bury it... :P
//                      if ~isempty(zr) &&
   zr(end) == L % In the case where there's at
   least one zero value, and that is at the
   index == L...
if (!zr.is_empty() && zr(zr.n_elem - 1) == L -
   1) // <== Diff: == L-1, not == L
{
   //                          fm_tmp(zr(end), k)
      = fm_tmp(nzr(end), k); % ... replace this
      index value with the last nonzero one, as
      done before...
   fm_tmp(zr(zr.n_elem - 1), k) =
      fm_tmp(nzr(nzr.n_elem - 1), k);

   //                          am_tmp(zr(end), k)
      = am_tmp(nzr(end), k); % same for amps
   am_tmp(zr(zr.n_elem - 1), k) =
      am_tmp(nzr(nzr.n_elem - 1), k);

   //                          nzr = [nzr;
      zr(end)]; % ... and form a new nonzero
      index vector, as done before
   nzr = join_cols(nzr, uvec{ zr(zr.n_elem - 1)
                });

              //                      end
            }
            //
            //                      % Interpolate linearly
               between samples, to extend
            //                      % the frequency
               trajectory
            //                      % Same for amplitudes

            //                      fm_tmp(:, k) =
               interp1(nzr, fm_tmp(nzr, k), (1:L)',
               'linear'); % could be linear
            vec X = conv_to<vec>::from(nzr);
            vec Y = conv_to<vec>::from(fm_tmp(nzr, uvec{ k
               }));
            vec XI = conv_to<vec>::from(regspace(0, L -
               1).t());
            vec YI;
            interp1(X, Y, XI, YI, "linear");
            fm_tmp.col(k) = conv_to<colvec>::from(YI);

            //                      am_tmp(:, k) =
               interp1(nzr, am_tmp(nzr, k), (1:L)',
               'linear'); % could be linear
            X = conv_to<vec>::from(nzr);
            Y = conv_to<vec>::from(am_tmp(nzr, uvec{ k }));
            XI = conv_to<vec>::from(regspace(0, L - 1).t());
            YI;
            interp1(X, Y, XI, YI, "linear");
            am_tmp.col(k) = conv_to<colvec>::from(YI);

            //                  end
        }
        //
        //                  % A new index is formed
        //                  idx = K+1 + [-fliplr(idx), 0,
           idx];

//// DIFF ///
// This line:
        //idx = K + 1 +
            join_rows(-fliplr<uvec>(idx.as_row()).as_row(),
            uvec{ 0 }, idx.as_row());
// Becomes this block:
        ivec tmp_idx = zeros<ivec>(2 * idx.n_elem + 1);
                    // <== Diff
        tmp_idx(span(0, idx.n_elem - 1)) =
            -conv_to<ivec>::from(reverse(idx + 1));// <==
            Diff
        tmp_idx(idx.n_elem) = 0;// <== Diff
        tmp_idx(span(idx.n_elem + 1, 2 * idx.n_elem)) =
            conv_to<ivec>::from(idx + 1);// <== Diff
        tmp_idx = tmp_idx + K + 1;// <== Diff
        idx = conv_to<uvec>::from(tmp_idx); // <== Diff


                //                  % fm_tmp flips
                    positive frequencies and makes them
                //                  % negative, puts zero
                    between, and adds the positive
                //                  % ones
                //                  % amplitudes follow
                    the same rule
                //                  % The zero vector in
                    between is the zero frequency!
                //                  % So, fm_tmp has the
                    frequencies of the current frame
                //                  % (truly existing or
                    born)

                //                  fm_tmp =
                    [-flipud(fm_tmp), zeros(L,1), fm_tmp];
        //fm_tmp = join_cols(-flipud(fm_tmp), zeros(L, 1),
            fm_tmp);
        fm_tmp = join_rows(-flipud(fm_tmp), zeros(L, 1),
            fm_tmp); //<== Diff: rows not cols


                //                  am_tmp =
                    [flipud(am_tmp), zeros(L,1), am_tmp];
        //am_tmp = join_cols(flipud(am_tmp), zeros(L, 1),
            am_tmp);
        am_tmp = join_rows(flipud(am_tmp), zeros(L, 1),
   am_tmp); //<== Diff


        //
        //
        //                %
           -------------------------------------
           ----------------
        //                % !!!
           ------------------- aQHM
           -------------------- !!!
        //                %
           -------------------------------------
           ----------------
        //
        //                % For these modified
           frequencies,
        //                % hit aQHM
        //                if opt.ea == 0
cx_mat ak_tmp;
cx_mat bk_tmp;
tmp_tin = ti(i) + n; // <== Diff: need ivec to do
   math instead of uvec.

if (opt.ea == 0)
{
  //                    [ak_tmp, bk_tmp, SNR] =
     compLSfm_akbk(s(ti(i)+n), fm_tmp, win, fs);
  //                    %aSNR(m, i) = SNR;
  //compLSfmAkbk tempStruct =
     compLSfm_akbk(s(ti(i) + n), fm_tmp, win, fs);
  compLSfmAkbk tempStruct =
     compLSfm_akbk(conv_to<vec>::from(
     s(conv_to<uvec>::from(tmp_tin))), fm_tmp,
     win, fs); // <== Diff
  ak_tmp = tempStruct.ak;
  bk_tmp = tempStruct.bk;
  double SNR = tempStruct.SNR;

}
//                    else % or eaQHM
else
{
   //                       [ak_tmp, bk_tmp, SNR] =
      compLSamfm_akbk(s(ti(i)+n), am_tmp, fm_tmp,
      win, fs);
   //                       %aSNR(m, i) = SNR;
   //compLSamfmAkbk tempStruct =
      compLSamfm_akbk(s(ti(i) + n), am_tmp,
      fm_tmp, win, fs);
   compLSamfmAkbk tempStruct = compLSamfm_akbk(
      s(conv_to<uvec>::from(tmp_tin)), am_tmp,
      fm_tmp, win, fs); // <== Diff
   ak_tmp = tempStruct.ak;
   bk_tmp = tempStruct.bk;
   double SNR = tempStruct.SNR;
   //                   end
}
//
//                   % Correction term
//                   df_tmp =
    fs/(2*pi)*((real(ak_tmp).*imag(bk_tmp)-
    imag(ak_tmp) .*real(bk_tmp)) ./
    abs(ak_tmp).^2)';
rowvec df_tmp = fs / (2 * datum::pi) *
    ((real(ak_tmp) % imag(bk_tmp) - imag(ak_tmp) %
    real(bk_tmp)) / square(abs(ak_tmp))).t();

//
//                   % Place ak, bk, df values in
    vectors
//                   ak = [];
//ak.clear();
ak = zeros<cx_mat>(idx(idx.n_elem - 1) + 1, 1); //
    <== Diff

        //                   ak(idx) = ak_tmp;
        //ak(idx) = ak_tmp;
ak(idx, uvec{ 0 }) = ak_tmp; // <== Diff

        //                   bk = [];
        //bk.clear();
bk = zeros<cx_mat>(idx(idx.n_elem - 1) + 1, 1); //
    <== Diff

        //                   bk(idx) = bk_tmp;
```

```
                //bk(idx) = bk_tmp;
            bk(idx, uvec{ 0 }) = bk_tmp; // <== Diff

                //                    df = [];
                //df.clear();
            df = zeros<cx_rowvec>(idx(idx.n_elem - 1) + 1); //
                <== Diff

                //                    df(idx) = df_tmp;
                //df(idx) =
                    conv_to<cx_rowvec>::from(df_tmp);
            df(idx) = conv_to<cx_rowvec>::from(df_tmp); // <==
                Diff
                //
                //                    % aQHM process end
                    here!
                //                    %
                    ----------------------
        }
        //              end
        //
        //              % amplitude of mean value in the
            center of the window
        //              % is set as the ak(K+1) (central
            estimate)
        //              a0_hat(ti(i)) = real(ak(K+1));
//             a0_hat(ti(i)) = real(ak(K + 1));

        //
        //              % Parameters are set as > K+1 indices
            of the estimated
        //              % values (keep only positive
            frequency values)
        //              ak = ak(K+2:2*K+1);
  //           ak = ak.rows(span(K + 1, 2 * K));

        //              bk = bk(K+2:2*K+1);
    //         bk = bk.rows(span(K + 1, 2 * K));

        //              df = df(K+2:2*K+1);
      //       df = df.cols(span(K + 1, 2 * K));

    // START: Diff block //
```

```
if (m == 0) {
   a0_hat(ti(i)) = real(ak(K));
   ak = ak.rows(span(K + 1, 2 * K));
   bk = bk.rows(span(K + 1, 2 * K));
   df = df.cols(span(K + 1, 2 * K));
}
else {

   a0_hat(ti(i)) = real(ak(K + 1));
   ak = ak.rows(span(K + 2, 2 * K + 2));
   bk = bk.rows(span(K + 2, 2 * K + 2));
   df = df.cols(span(K + 2, 2 * K + 2));

}

// END: Diff block //


        //
        //                % Maximum logarithm of |ak|
        //                ak_log_max =
            20*log10(max(abs(ak)));
double ak_log_max = 20 * log10(max(abs(ak.as_row())));

int Kp;             //< == Changed
if (m == 0) Kp = K;  //< == Changed
else Kp = K + 1;     //< == Changed


//
//                % For every frequency index between 1
    and K...
//                for k = 1:K
for (int k = 0; k < Kp; k++)
{
   //                   % ... compute log of |ak| ...
   //                   ak_log = 20*log10(abs(ak(k)));
   double ak_log = 20 * log10(abs(ak(k, 0))); // <==
       Diff

   //                   % If some restrictions apply...
   //                   % (log|ak|+50 > log(max|ak|))
       AND (|df(k)| < f0) (AS IT
//                  % ORIGINALLY WAS)
//                  % !!!!!!!!!!!!!!!!!!! CHANGES
    HERE !!!!!!!!!!!!!!!!!!!
//                  if  ak_log+150 > ak_log_max &&
    abs(df(k)) < f0/(m+1)
if (ak_log + 150 > ak_log_max && abs(df(k)) < f0 /
    (m + 1))
{
  //                      %
  //                      % am estimation for
      k-frequency in the center is |ak(k)|
  //                      am_hat(ti(i), k) =
      abs(ak(k));
  am_hat(ti(i), k) = abs(ak(k, 0)); // <== Diff

  //                      % pm estimation for
      k-frequency in the center is <) ak(k)
  //                      pm_hat(ti(i), k) =
      angle(ak(k));
  pm_hat(ti(i), k) = arg(ak(k, 0));

  //
  //                      if m == 0 % In the
      QHM/iQHM context...
  if (m == 0)
  {
    //                        % ...fm estimation
        for k-frequency is k*f0 + correction term
        for k-th frequency
    //                        fm_hat(ti(i), k) =
        k*f0 + df(k);
    fm_hat(ti(i), k) = (k + 1) * f0 + real(df(k));

    //                        %pause;
  }
  //                      elseif f0 >
      opt_pitch_f0min % In the aQHM context...
  else if (f0 > opt_pitch_f0min)
  {
    //                        % ...fm estimation
        for the k-frequency is fm_curr +
        correction term for k-th frequency
    //                        fm_hat(ti(i), k) =
                fm_cur(ti(i), k) + df(k);
            fm_hat(ti(i), k) = fm_cur(ti(i), k) +
                real(df(k));
        }
        //                      else
        else
        {
            //                          fm_hat(ti(i), k) =
                fm_cur(ti(i), k);
            fm_hat(ti(i), k) = fm_cur(ti(i), k);

            //                      end
        }
        //                  end
    }
    //              end % for loop ends here
  }
}
//          else % if two consecutive frames are not
    both voiced, do the necessary marking...
else
{
  //              D(i).isS = 1;
  D[i].isS = 1;
  //              D(i).isV = 0;
  D[i].isV = 0;

  //          end % First IF ends here
}
//

}
//      else % We are outside interval [M, Len-M], where
    M is half the pitch analysis window,
else
{
  //          % so we mark the frame as non-speech.
  //          D(i).isS = 0;
  D[i].isS = 0;

  //          D(i).isV = 0;
  D[i].isV = 0;
```

```
   }

   //       end
   //    end % Analysis using time instants ends here
}
//
//    % ----------------------------------------------
    -----------------------
//    %                   PARAMETER INTERPOLATION
//    % ----------------------------------------------
    -----------------------
//    fm_cur = zeros(len, Kmax);
fm_cur = zeros(len, Kmax);

//    am_cur = zeros(len, Kmax); % eaQHM
am_cur = zeros(len, Kmax);

//
//    % Interpolating and extrapolating amplitude of mean
    value for !all! samples
//    a0_hat = interp1(ti, a0_hat(ti), (1:len)', 'spline',
    'extrap');
vec X = conv_to<vec>::from(ti);
vec Y = conv_to<vec>::from(a0_hat.rows(ti));
vec XI = conv_to<vec>::from(regspace(0, len - 1).t());
vec YI;
//interp1(X, Y, XI, YI);
interp1(X, Y, XI, YI, "linear", 0.0); // <== Diff
a0_hat = conv_to<colvec>::from(YI);

//
//    % For all the sinusoids up to the maximum allowed...
//    for k = 1:Kmax
for (unsigned int k = 0; k < Kmax; k++)
{
   //       % ... find the nonzero values of the am
      estimations...
   //       nzv = find(am_hat(:,k) ~= 0); % non zero values
   uvec nzv = find(am_hat.col(k) != 0).as_col();

   //
   //       % ... differentiate the indices, creating dznv =
      t_{i+1}-t_i...
```

```
//       dnzv = diff([1; nzv; len]);
uvec dnzv = diff(join_cols(uvec{ 1 }, nzv, uvec{ len }));

//%        figure(m+1);plot(nzv, fm_hat(nzv,k), 'b.');hold
    on;
//
//       % ... and keep those indices corresponding to
    those who have time difference < 10 msec...
//       dnzv_idx = dnzv <= step; % 10 ms
uvec dnzv_idx = dnzv <= step;

//
//       % start of time instant and end of time
    instants?????????
//       st_ti = find(diff(dnzv_idx) == 1);
uvec st_ti = find(diff(dnzv_idx) == 1);

//       en_ti = find(diff(dnzv_idx) == -1);
uvec en_ti = find(diff(dnzv_idx) == -1);

//
//       % For every time instant...
//       for i = 1:length(st_ti)
for (int i = 0; i < st_ti.n_elem; i++)
{
   //           idx1 = nzv(st_ti(i) : en_ti(i));
   uvec idx1 = nzv(span(st_ti(i), en_ti(i)));

   //           idx2 = nzv(st_ti(i)) : nzv(en_ti(i));
   //uvec idx2 = nzv(span(st_ti(i)), nzv(en_ti(i)));
   uvec idx2 = regspace<uvec>(nzv(st_ti(i)),
       nzv(en_ti(i))); // <== Diff: it's a range

         //
         //           % linear amplitude interpolation
         //           am_hat(idx2,k) = interp1(idx1,
             am_hat(idx1,k), idx2', 'linear');
   vec X = conv_to<vec>::from(idx1);
   vec Y = conv_to<vec>::from(am_hat(idx1, uvec{ k }));
   vec XI = conv_to<vec>::from(idx2.t());
   vec YI;
   interp1(X, Y, XI, YI, "linear");
   am_hat(idx2, uvec{ k }) = conv_to<colvec>::from(YI);
```

```
//
//          % spline frequency interpolation
//          fm_hat(idx2,k) = interp1(idx1,
    fm_hat(idx1,k), idx2', 'spline');
X = conv_to<vec>::from(idx1);
Y = conv_to<vec>::from(fm_hat(idx1, uvec{ k }));
XI = conv_to<vec>::from(idx2.t());
YI;
interp1(X, Y, XI, YI);
fm_hat(idx2, uvec{ k }) = conv_to<colvec>::from(YI);

//          %fm_old(idx2,k) = interp1(idx1,
    fm_old(idx1,k), idx2', 'spline');
//
//          % phase interpolation using the integral
    method
//          pm_hat(idx2,k) =
    AQHMphase_interp(2*pi/fs*fm_hat(:,k), pm_hat(:,k),
    idx1, 'integr'); % 'integr' or 'cubic' or 'other'
pm_hat(idx2, uvec{ k }) = AQHMphase_interp(2 * datum::pi
    / fs * fm_hat.col(k), pm_hat.col(k), idx1, "integr");

//
//          % frequency tracks generated by unwrapped
    phase (after phase
//          % interpolation and correction)
//          fm_cur(idx2,k) = [fm_hat(idx2(1),k);
    fs/(2*pi)*diff(unwrap(pm_hat(idx2,k)))];
//fm_cur(idx2, uvec{ k }) = join_cols(colvec{
    fm_hat(idx2(1), k) }, fs / (2 * datum::pi) *
    diff(unwrap1(pm_hat(idx2, uvec{ k }))));
vec fm_hat_col = fm_hat.col(k); // <== Diff
vec pm_hat_col = pm_hat.col(k); // <== Diff
fm_cur(idx2, uvec{ k }) = join_cols(colvec{
    fm_hat_col(idx2(0)) }, fs / (2 * datum::pi) *
    diff(unwrap1(pm_hat_col(idx2)))); // <== Diff


     //          %fm_cur(idx2,k) = fm_cur(fm_cur(idx2,
         k) < fs/2);
     //          %fm_cur(idx2,k) = fm_hat(idx2, k);
     //       end
```

```
   }
   //    end
}

//    %figure(m+1); plot(fm_hat);
//
//    % Keeping amplitudes for next adaptation (eaQHM only)
//    am_cur = am_hat;
am_cur = am_hat;

//
//    % Reconstruct the signal
//    s_hat = a0_hat + 2*sum(am_hat.*cos(pm_hat), 2);
//mat s_hat = a0_hat + 2 * sum(am_hat % cos(pm_hat), 2);
mat s_hat = a0_hat + 2 * sum(am_hat % cos(pm_hat), 1); //<==
    Diff: Last parameters = 1

   //
   //    % Find the SRER
   //    SRER(m+1) = 20*log10(std(deterministic_part) /
       std(deterministic_part-s_hat));
SRER(m) = 20 * log10(stddev(deterministic_part.as_row()) /
    stddev(deterministic_part.as_row() - s_hat.as_row()));

//
//    if m~=0 % In the aQHM context...
if (m != 0)
{
   //       if SRER(m+1) <= SRER(m) % ...stop if the new SRER
       is leq than previous
   if (SRER(m) <= SRER(m - 1))
   {
      //           break;
      break;
   }
   //       else % or else, update the final parameters
   else
   {
      //           a0_fin = a0_hat;
      //           am_fin = am_hat;
      //           fm_fin = fm_hat;
      //           pm_fin = pm_hat;
      //           % Store deterministic part
```

```
            //          V.qh = s_hat;
            a0_fin = a0_hat;
            am_fin = am_hat;
            fm_fin = fm_hat;
            pm_fin = pm_hat;
            V.qh = s_hat.as_col();
            //      end
        }
    }
    //   else % In the QHM/iQHM context, just update the
       parameters
    else
    {
        //      a0_fin = a0_hat;
        //      am_fin = am_hat;
        //      fm_fin = fm_hat;
        //      pm_fin = pm_hat;

        a0_fin = a0_hat;
        am_fin = am_hat;
        fm_fin = fm_hat;
        pm_fin = pm_hat;
        V.qh = s_hat.as_col();

        //   end
    }
    //
    //   % Show the final SRER
    //   str = sprintf('SRER: %f in Adaptation No: %d\n',
       SRER(m+1), m);
    //   disp(str);
    printf("SRER: %lf in Adaptation No : %d \n\n", SRER(m), m);
    //cout << "SRER: " << SRER(m) << " in Adaptation No : " << m
       << "\n"; // <== Diff: Double quote around \n
    //
    //   %figure(m+1); plot(deterministic_part); hold
       on;plot(s_hat,'r'); plot(deterministic_part-s_hat,'g');
       hold off; %pause;
    //end
}
//
//
//% Save the parameters of deterministic part (aQHM) in data
```

```
    structure D.
//for i = 1:No_ti % For all time instants...
for (int i = 0; i < No_ti; i++)
{
   //   if D(i).isV % if we are in a voiced frame, start
       storing...
   if (D[i].isV)
   {
      //       ti = D(i).ti;
      ti = (uword)D[i].ti;

      //       D(i).a0 = a0_fin(ti);
      D[i].a0 = a0_fin(D[i].ti, 0);

      //       idx = find(am_fin(ti, :)~=0);
      uvec idx = find(am_fin.rows(ti) != 0);

      //       D(i).ak(idx) = am_fin(ti, idx);
      D[i].ak = zeros<rowvec>(am_fin.n_cols); // <== Diff
      D[i].ak(idx) = am_fin(ti, idx);

      //       D(i).fk(idx) = fm_fin(ti, idx);
      D[i].fk = zeros<rowvec>(am_fin.n_cols); // <== Diff
      D[i].fk(idx) = fm_fin(ti, idx);

      //       D(i).pk(idx) = pm_fin(ti, idx);
      D[i].pk = zeros<rowvec>(am_fin.n_cols); // <== Diff
      D[i].pk(idx) = pm_fin(ti, idx);
      //   end
   }
   //end
}
//
//% Storing data into struct
//D(end).fb = opt.fb;
//D(end).Kmax = Kmax;
//D(end).Fmax = Fmax;
//D(end).filename = speechFile;
//D(end).fl = opt.fl;
D[D.size() - 1].fb = opt.fb;
D[D.size() - 1].Kmax = Kmax;
D[D.size() - 1].Fmax = Fmax;
D[D.size() - 1].filename = speechFile;
```

```
  D[D.size() - 1].fl = opt.fl;

  //
  //fprintf(1, 'Signal adapted to %.4f dB SRER\n', max(SRER));
  printf("Signal adapted to %lf dB SRER\n", max(SRER));
  //cout << "Signal adapted to " << max(SRER) << " dB SRER\n";
  //
  //end
  eaQHManalysisStruct output = {};
  output.aSNR = aSNR;
  output.D = D;
  output.SRER = SRER;
  output.V = V;
  return output;
}



optionFile eaQHManalysis()
{
  optionFile opt;
  return opt;
}
```

A.8 eaQHMinput.cpp

This file defines a class for user input.

```
#include "eaQHMinput.h"

std::string eaQHMinput::getPath(std::string value, const char
    delim_path) {
   size_t pos_data, pos_path;
   std::string tmp_value;

      // Find first delimiter
      pos_path = value.find(delim_path);
      if (pos_path == std::string::npos) throw
          std::invalid_argument("Wrong format for path");
      tmp_value = value.substr(pos_path + 1, std::string::npos);

      //Find final delimiter
      pos_path = tmp_value.find(delim_path);
      if (pos_path == std::string::npos) throw
          std::invalid_argument("Wrong format for path");
      tmp_value = tmp_value.substr(0, pos_path);

      //Set the value
      return tmp_value;

}

eaQHMinput::eaQHMinput() {}

eaQHMinput::eaQHMinput(std::ifstream& filename) {
   std::string line;


   while (std::getline(filename, line))
   {
      const char delim_comment = '#'; // delimiter for comments
      const char delim_value = '='; // delimiter for data
      const char delim_path = '"'; // delimiter for data

      std::string tag, value;
      size_t pos_comment,pos_value, pos_data,pos_path;

      // Skip empty lines
```

```
if (line.empty()) continue;

// Drop comments
pos_comment = line.find(delim_comment);
if (pos_comment != std::string::npos) {
   line = line.substr(0, pos_comment);
}

// Skip if only a comment is present (or line is actually
    empty)
if (line.empty()) continue;

//Get the data
pos_value = line.find(delim_value);
if (pos_value != std::string::npos) {
   tag = line.substr(0, pos_value);
   value = line.substr(pos_value+1, std::string::npos);
}
else { throw std::invalid_argument("Missing value for: " +
    line); }

// Search for value can that be
pos_data = tag.find("deterministicPath");
if (pos_data != std::string::npos) {

   try {
      filename_det = eaQHMinput::getPath(value, delim_path);
   }
   catch (...) {
      throw std::invalid_argument("Something went wrong with
          the deterministic path");
   }

   continue;
}

pos_data = tag.find("fundamentalFrequencyPath");
if (pos_data != std::string::npos) {

   try {
      filename_f0 = eaQHMinput::getPath(value, delim_path);
   }
   catch (...) {
        throw std::invalid_argument("Something went wrong with
            the path for the fundamental frequencies");
    }

    continue;
}

pos_data = tag.find("f0sinPath");
if (pos_data != std::string::npos) {

    try {
        filename_f0sin = eaQHMinput::getPath(value, delim_path);
    }
    catch (...) {
        throw std::invalid_argument("Something went wrong with
            the path for f0sin");
    }

    continue;
}

pos_data = tag.find("speakerGender");
if (pos_data != std::string::npos) {

    try {
        gender = eaQHMinput::getPath(value, delim_path);
    }
    catch (...) {
        throw std::invalid_argument("Something went wrong with
            gender of the speaker");
    }

    continue;
}

pos_data = tag.find("speechFilePath");
if (pos_data != std::string::npos) {

    try {
        speechFile = eaQHMinput::getPath(value, delim_path);
    }
    catch (...) {
        throw std::invalid_argument("Something went wrong with
            the path for the speech file");
      }

      continue;
   }


   pos_data = tag.find("pitchFilePath");
   if (pos_data != std::string::npos) {

      try {
         filename_P = eaQHMinput::getPath(value, delim_path);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the path for the file with pitch info");
      }

      continue;
   }


   pos_data = tag.find("adaptations");
   if (pos_data != std::string::npos) {
      try {
         adpt = std::stoul(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            adaptations");
      }
      continue;
   }

   pos_data = tag.find("maximumFreq");
   if (pos_data != std::string::npos) {

      try {
         Fmax = std::stoul(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the maximum frequency Fmax");
      }

      continue;
   }

   pos_data = tag.find("samplingFreq");
   if (pos_data != std::string::npos) {

      try {
         fs = std::stoul(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the sampling frequency fs");
      }

      continue;
   }

   pos_data = tag.find("samplingLength");
   if (pos_data != std::string::npos) {

      try {
         len = std::stoul(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the sampling length");
      }

      continue;
   }


   pos_data = tag.find("stepNumber");
   if (pos_data != std::string::npos) {

      try {
         step = std::stoul(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the number of steps");
      }

      continue;
   }

   pos_data = tag.find("valueI");
   if (pos_data != std::string::npos) {

      try {
         i = std::stoi(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the variable i");
      }

      continue;
   }

   pos_data = tag.find("maximumK");
   if (pos_data != std::string::npos) {

      try {
         Kmax = std::stoi(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            Kmax");
      }

      continue;
   }

   pos_data = tag.find("numberPitchPoints");
   if (pos_data != std::string::npos) {

      try {
         NoP = std::stoi(value, nullptr, 0);
      }
      catch (...) {
         throw std::invalid_argument("Something went wrong with
            the number of pitch points");
      }
```

```
        continue;
    }


    pos_data = tag.find("pitchMinimumF0");
    if (pos_data != std::string::npos) {

        try {
            opt_pitch_f0min = std::stoi(value, nullptr, 0);
        }
        catch (...) {
            throw std::invalid_argument("Something went wrong with
                the minimum f0 for pitch");
        }

        continue;
    }


    pos_data = tag.find("pitchSteps");
    if (pos_data != std::string::npos) {

        try {
            p_step = std::stoi(value, nullptr, 0);
        }
        catch (...) {
            throw std::invalid_argument("Something went wrong with
                the number of steps for pitch");
        }

        continue;
    }

    // If you reach so far, you have not found a good keyword

    throw(std::invalid_argument("Wrong keyword in file: " + tag));

  }
}


void eaQHMinput::printAll() {
  std::cout << "deterministic path = " << "\""<< filename_det
      <<"\"" << std::endl;
  std::cout << "f0 path = " << "\"" << filename_f0 << "\"" <<
      std::endl;
  std::cout << "f0sin path = " << "\"" << filename_f0sin << "\""
      << std::endl;
  std::cout << "gender = " << "\"" << gender << "\"" << std::endl;
  std::cout << "speechfile = " << "\"" << speechFile << "\"" <<
      std::endl;
  std::cout << "P path = " << "\"" << filename_P << "\"" <<
      std::endl;

  std::cout << "adpt = " << adpt << std::endl;
  std::cout << "Fmax = " << Fmax << std::endl;
  std::cout << "fs = " << fs << std::endl;
  std::cout << "len = " << len << std::endl;
  std::cout << "steps = " << step << std::endl;

  std::cout << "i = " << i << std::endl;
  std::cout << "Kmax = " << Kmax << std::endl;
  std::cout << "NoP = " << NoP << std::endl;
  std::cout << "pitch f0 = " << opt_pitch_f0min << std::endl;
  std::cout << "step p = " << p_step << std::endl;

}

void eaQHMinput::printSample() {

  std::cout << "deterministicPath = " << "\"" << filename_det <<
      "\"" << std::endl;
  std::cout << "fundamentalFrequencyPath = " << "\"" <<
      filename_f0 << "\"" << std::endl;
  std::cout << "f0sinPath = " << "\"" << filename_f0sin << "\"" <<
      std::endl;
  std::cout << "speakerGender = " << "\"" << gender << "\"" <<
      std::endl;
  std::cout << "speechFilePath = " << "\"" << speechFile << "\""
      << std::endl;
  std::cout << "pitchFilePath = " << "\"" << filename_P << "\"" <<
      std::endl;

  std::cout << "adaptations = " << adpt << std::endl;
  std::cout << "maximumFreq = " << Fmax << std::endl;
  std::cout << "samplingFreq = " << fs << std::endl;
  std::cout << "samplingLength = " << len << std::endl;
  std::cout << "stepNumber = " << step << std::endl;

  std::cout << "valueI = " << i << std::endl;
  std::cout << "maximumK = " << Kmax << std::endl;
  std::cout << "numberPitchPoints = " << NoP << std::endl;
  std::cout << "pitchMinimumF0 = " << opt_pitch_f0min << std::endl;
  std::cout << "pitchSteps = " << p_step << std::endl;

}
```

A.9 eaQHMinput.h

This is a header file for *eaQHMinput.cpp*.

```
#include <string>
#include <iostream>
#include <fstream>

#ifndef eaQHMinput_H
#define eaQHMinput_H

/**
 *  Store information gathered from the input file
*/
class eaQHMinput
{

  std::string getPath(std::string value, const char delim_path);

  public:

    std::string filename_det = "sa19_s.txt"; //!< File name with
       the deterministic part
    std::string filename_f0 = "f0s.txt"; //!< File name with the
       fundamental frequency f0
    std::string filename_f0sin = "f0sin.txt"; //!< File name with
       f0sin
    std::string gender = "female"; //!< Gender of speaker
    std::string speechFile = "SA19.wav"; //!< File with the audio
    std::string filename_P = "P.txt"; //!< File with pitch
       information

    unsigned int adpt = 10;//!< number of adaptations
    unsigned int Fmax = 7800;//!< Maximum voice frequency
    unsigned int fs = 16000; //!< Sampling frequency
    unsigned int len = 63488; //!< number of time instants
    unsigned step = 15; //!< number of samples

    int i = 794; //!< unknown
    int Kmax = 68; //!< maximum K
    int NoP = 3; //!< Number of analysis window size, in pitch
       periods
    int opt_pitch_f0min = 120; //!< Pitch minimum f0
    int p_step = 80; //!< unknown
```

```
        /**
         * Reads the data from the input file
         *
         * @param filename The name of the file with the input data
         *
        */
        eaQHMinput(std::ifstream& filename);

        /**
         * Create the variable with default values
         *
         *
        */
        eaQHMinput();

        /**
         * Print all of the values stored within the objects
         *
         *
        */
        void printAll();

        /**
         * Print a sample input file with default values
         *
         *
        */
        void printSample();

};

#endif
```

A.10 filter.cpp

This file defines a function for filtering out noise.

```
//#include <RcppArmadillo.h>
#include <armadillo>
#include "Functions.h"

using namespace arma;

//[[Rcpp::export]]
colvec filter(colvec b, colvec a, colvec x)
{
   colvec a1 = getRealArrayScalarDiv(a, a[0]);
   colvec b1 = getRealArrayScalarDiv(b, a[0]);
   unsigned int sx = x.n_elem;
   colvec filter(sx);
   filter[0] = b1[0] * x[0];
   for (unsigned int i = 1; i < sx; ++i)
   {
      filter[i] = 0.0;
      for (unsigned int j = 0; j <= i; j++)
      {
        unsigned int k = i - j;
         if (j > 0)
         {
            if ((k < b1.n_elem) && (j < x.n_elem))
            {
               filter[i] += b1[k] * x[j];
            }
            if ((k < filter.n_elem) && (j < a1.n_elem))
            {
               filter[i] -= a1[j] * filter[k];
            }
         }
         else
         {
            if ((k < b1.n_elem) && (j < x.n_elem))
            {
               filter[i] += (b1[k] * x[j]);
            }
         }
      }
   }
   return filter;
}
```

A.11 Functions.cpp

This files includes a set of functions used throughout the program.

```cpp
//#include <RcppArmadillo.h>
#include <armadillo>
using namespace arma;

colvec getRealArrayScalarDiv(colvec dDividend, double dDivisor)
{
   colvec dQuotient(dDividend.n_elem);

   for (unsigned int i = 0; i < dDividend.n_elem; ++i)
   {
      if (dDivisor != 0.0)
      {
         dQuotient[i] = dDividend[i] / dDivisor;
      }
      else
      {
         if (dDividend[i] > 0.0)
         {
            dQuotient[i] = datum::inf;
         }
         if (dDividend[i] == 0.0)
         {
            dQuotient[i] = datum::nan;
         }
         if (dDividend[i] < 0.0)
         {
            dQuotient[i] = -datum::inf;
         }
      }
   }
   return dQuotient;
}
```

A.12 Functions.h

This is a header file for *Functions.cpp*.

```
#pragma once
#ifndef FUNCTIONS_H
//#include <RcppArmadillo.h>
#include <armadillo>
#include "audio.h"
#include "DataStructures.h"
using namespace arma;
using namespace std;

vec cos_win(const uword N, const vec& a);
vec hamming(const uword N);
vec blackman(const uword N);
vec hanning(const uword N);
vec unwrap1(const vec& x);
cx_mat specgram_cx(const colvec& x, const uword Nfft, const int
    Fs, const vec W, const uword Noverl);
cx_vec spectrum(const colvec& x, const vec& W);
mat specgram(const colvec& x, const uword Nfft, const int Fs,
    const vec W, const uword Noverl);
rowvec primes(unsigned const int n);
colvec filter(colvec b, colvec a, colvec x);
colvec filtfilt(colvec &b, colvec &a,colvec &x);
audio<colvec> WavRead(string Fn);
colvec getRealArrayScalarDiv(colvec dDividend, double dDivisor);
mat loadpitch(string FileName, string type = "float");
rowvec medFilt(colvec x, unsigned int p);
mat swipep(colvec, int, rowvec, float, float, float, float, float);
eaQHManalysisStruct eaQHManalysis(string, unsigned int, string,
    optionFile);
optionFile eaQHManalysis();
pair<mat, Mstruct> eaQHMsynthesis(vector<Dstruct>,
    vector<Sstruct>, unsigned int, mod_s);
icompLSakbk icompLS_akbk(colvec s, colvec fk, colvec win, const
    int fs, unsigned int iter);
compLSfmAkbk compLSfm_akbk(colvec s, mat fm, colvec win, unsigned
    int fs);
compLSamfmAkbk compLSamfm_akbk(colvec s, mat am, mat fm, colvec
    win, unsigned int fs);
pair<cx_colvec, mat> AMFMrec(mat am_hat, mat fm_hat, mat pm_hat);
tuple<mat, mat, mat> freq_match(mat am, mat fm, mat pm, const
    unsigned int step, vec ti, const unsigned int Fs);
mat AQHMphase_interp(colvec fm_hat, colvec pm_hat, uvec idx, const
    string method);
vector<D> AMFMdec(colvec s, unsigned int fs, rowvec fk, const
    unsigned int step, int M, string interp_mthd, int
    extended_flag);
void eaQHMwrapper();

template<class T>
T myRange(int start, int end, int step){
  T res(end-start+1);
  for (unsigned int i = 0; i < res.n_elem; ++i, start+=step)
    res[i]=start;
  return res;
}

#endif
```

A.13 icompLS_akbk.cpp

This file contains the function for the least square fitting for the initialization step.

```
//#include <RcppArmadillo.h>
#include <armadillo>
#include "DataStructures.h"
#include "Functions.h"

using namespace arma;

//[[Rcpp::export]]
icompLSakbk icompLS_akbk(colvec s, colvec fk, colvec win, const
    int fs, unsigned int iter)
{
   const int K = fk.n_elem, N = (s.n_elem - 1) / 2;
   colvec n = linspace<colvec>(-N, N, 2 * N + 1); // This is
       actually n'

   mat t = n * (2.0 * datum::pi / fs) * fk.t(); // fk.t() instead
       of fk because MATLAB uses a rowvec instead of colvec
   cx_mat E(cos(t), sin(t));
   E = join_horiz(E, repmat(n, 1, K) % E);
   cx_mat Ew = repmat(win, 1, 2 * K) % E;
   cx_mat R = Ew.t() * Ew;

   icompLSakbk str{}; // <== Moved

   if (rcond(R) <= 1e-10) { // <== Changed
      cerr << "CAUTION: Bad condition of matrix.\n";
      str.ak.zeros(K); // <== Added
      str.bk.zeros(K); // <== Added
      str.df.zeros(K); // <== Added
      str.SNR = 0.0;   // <== Added
      return str;      // <== Changed
   }

   cx_mat x = solve(R, (Ew.t() * (win % s))); // <== Fixed

   str.ak = x(span(0, K - 1), 0);
   str.bk = x(span(K, 2 * K - 1), 0);
   str.df.zeros(K); // <== Added
```

```
cx_colvec cfk;
uvec idx;
colvec tmp_df(K, fill::zeros); // <== This is composed of real
    number, whereas str.df is complex

for (unsigned int i = 0; i < iter; ++i) {
   tmp_df = tmp_df + fs / (2 * datum::pi) * ((imag(str.bk) %
      real(str.ak) - imag(str.ak) % real(str.bk)) /
      square(abs(str.ak)));

   vec tmp_cfk(fk.n_elem + 2);
   tmp_cfk(0) = -0.5 * fs;
   tmp_cfk(span(1, tmp_cfk.n_elem - 2)) = fk;
   tmp_cfk(tmp_cfk.n_elem - 1) = 0.5 * fs;
   vec cfk = 0.5 * diff(tmp_cfk);

   idx = find((tmp_df < -cfk(span(0, K - 1))) || (tmp_df >
      cfk(span(1, cfk.n_elem - 1)))); // <== if tmp_df is
      complex, I cannot compare values
   tmp_df(idx).zeros();

   t = n * (2.0 * datum::pi / fs) * (fk.t() + tmp_df.t()); //
      <== Fixed
   E = cx_mat(cos(t), sin(t));
   E = join_horiz(E, repmat(n, 1, K) % E);
   Ew = repmat(win, 1, 2 * K) % E;
   R = Ew.t() * Ew;

   if (rcond(R) <= 1e-10) { // <== Changed
      cerr << "CAUTION: Bad condition of matrix.\n";
      str.ak.zeros(K); // <== Added
      str.bk.zeros(K); // <== Added
      str.df.zeros(K); // <== Added
      str.SNR = 0.0;  // <== Added
      return str;     // <== Changed
   }

   cx_mat x = solve(R, (Ew.t() * (win % s))); // <== Added

   str.ak = x(span(0, K - 1), 0);
   str.bk = x(span(K, 2 * K - 1), 0);
   str.df = cx_vec(tmp_df, zeros(tmp_df.n_elem)).t(); // <==
        str.df is complex, but df only has real part. Imaginary
        part = 0
   }

   mat realy = real(E * join_vert(str.ak, str.bk));
   str.SNR = 20.0 * log10(stddev(s) / stddev(s(span(0, realy.n_elem
       - 1)) - realy));
   return str;
}
```

Bibliography

[1] M. Hoy, "Alexa, siri, cortana, and more: An introduction to voice assistants," *Medical Reference Services Quarterly*, vol. 37, pp. 81–88, Jan-Mar 2018.

[2] M. L. Rohlfing, D. P. Buckley, J. Piraquive, C. E. Stepp, and L. F. Tracy, "Hey siri: How effective are common voice recognition systems at recognizing dysphonic voices?," *Laryngoscope*, vol. 131, pp. 1599–1607, Jul 2021.

[3] A. Palanica and Y. Fossat, "Medication name comprehension of intelligent virtual assistants: A comparison of amazon alexa, google assistant, and apple siri between 2019 and 2021," *Frontiers in Digital Health*, vol. 3, 2021.

[4] S. Sur and V. Sinha, "Event-related potential: An overview," *Industrial Psychiatry Journal*, vol. 18, pp. 70–73, Jan-Jun 2009.

[5] R. Islam, E. Abdel-Raheem, and M. Tarique, "Voice pathology detection using convolutional neural networks with electroglottographic (egg) and speech signals," *Computer Methods and Programs in Biomedicine Update*, vol. 2, p. 100074, 2022.

[6] M. A. Mohammed, K. H. Abdulkareem, S. A. Mostafa, M. Khanapi Abd Ghani, M. S. Maashi, B. Garcia-Zapirain, I. Oleagordia, H. Alhakami, and F. T. AL-Dhief, "Voice pathology detection and classification using convolutional neural network model," *Applied Sciences*, vol. 10, no. 11, 2020.

[7] N. Q. Abdulmajeed, B. Al-Khateeb, and M. A. Mohammed, "A review on voice pathology: Taxonomy, diagnosis, medical procedures and detection techniques, open challenges, limitations, and recommendations for future directions," *Journal of Intelligent Systems*, vol. 31, no. 1, pp. 855–875, 2022.

[8] S. A. Syed, M. Rashid, S. Hussain, A. Imtiaz, H. Abid, and H. Zahid, "Inter classifier comparison to detect voice pathologies," *Mathematical Biosciences and Engineering*, vol. 18, no. 3, pp. 2258–2273, 2021.

[9] J. G. Wilpon, "Voice-processing technologies—their application in telecommunications," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 92, no. 22, pp. 9991–9998, 1995.

[10] R. M. Gray, "A history of realtime digital speech on packet networks: Part ii of linear predictive coding and the internet protocol," *Found. Trends Signal Process.*, vol. 3, p. 203–303, apr 2010.

[11] D. R. Finlayson, "A More Loss-Tolerant RTP Payload Format for MP3 Audio." RFC 5219, Feb. 2008.

[12] Y. Song, "Active noise cancellation and its applications," *Journal of Physics: Conference Series*, vol. 2386, p. 012042, dec 2022.

[13] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[14] L. Rabiner and B. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, pp. 4–16, Jan 1986.

[15] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, "Speech recognition using deep neural networks: A systematic review," *IEEE access*, vol. 7, pp. 19143–19165, 2019.

[16] R. P. Lippmann, "Review of neural networks for speech recognition," *Neural computation*, vol. 1, no. 1, pp. 1–38, 1989.

[17] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8599–8603, IEEE, 2013.

[18] OpenAI, "Gpt-4 technical report," 2023.

[19] G. Fant, *Acoustic Theory of Speech Production with Calculations based on X-Ray Studies of Russian Articulations.* Berlin, Boston: De Gruyter Mouton, 1971.

[20] G. Fant, J. Liljencrants, and Q. Lin, "A four-parameter model of glottal flow," *STL-QPSR*, vol. 4, 01 1985.

[21] M. Plumpe, T. Quatieri, and D. Reynolds, "Modeling of the glottal flow derivative waveform with application to speaker identification," *IEEE Transactions on Speech and Audio Processing*, vol. 7, pp. 569–586, Sep. 1999.

[22] J. Makhoul, "Linear prediction: A tutorial review," *Proceedings of the IEEE*, vol. 63, pp. 561–580, April 1975.

[23] P. Alku, "Glottal wave analysis with pitch synchronous iterative adaptive inverse filtering," *Speech Communication*, vol. 11, no. 2, pp. 109–118, 1992. Eurospeech '91.

[24] A. El-Jaroudi and J. Makhoul, "Discrete all-pole modeling," *IEEE Transactions on Signal Processing*, vol. 39, pp. 411–423, Feb 1991.

[25] R. McAulay and T. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, pp. 744–754, August 1986.

[26] P. Stoica, R. Moses, B. Friedlander, and T. Soderstrom, "Maximum likelihood estimation of the parameters of multiple sinusoids from noisy measurements," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 378–392, March 1989.

[27] E. George and M. Smith, "Speech analysis/synthesis and modification using an analysis-by-synthesis/overlap-add sinusoidal model," *IEEE Transactions on Speech and Audio Processing*, vol. 5, pp. 389–406, Sep. 1997.

[28] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[29] C. Sanderson and R. Curtin, "A user-friendly hybrid sparse matrix class in c++," in *Mathematical Software – ICMS 2018* (J. H. Davenport, M. Kauers, G. Labahn, and J. Urban, eds.), (Cham), pp. 422–430, Springer International Publishing, 2018.