Personal and Ubiquitous Computing
https://doi.org/10.1007/s00779-024-01806-8

ORIGINAL ARTICLE

Implementation and optimization of Burg's method for real-time packet loss concealment in networked music performance applications

Matteo Sacchetto[1] · Cristina Rottondi[1] · Andrea Bianco[1]

Received: 24 March 2023 / Accepted: 3 May 2024

Abstract
In networked music performance (NMP) applications, which entail real-time audio streaming over the Internet, strict latency requirements are needed to ensure a realistic interaction between geographically dispersed musicians. Thus, NMP applications typically leverage uncompressed audio and unreliable transport protocols to avoid unnecessary processing and re-transmission delays. Given that no guarantee on packet delivery is offered, NMP applications must deal with late/lost audio packets to mitigate the impact of the resulting audio artifacts on the quality of the playback audio stream. This paper explores an audio packet loss concealment (PLC) technique based on autoregressive (AR) models. In particular, it investigates the algorithmic implementation of Burg's method and the parameters configuration that offers the best trade-off between prediction error and computational time requirements. The purpose is to find the most suitable solution capable of running on a Raspberry Pi 4B within the real-time audio boundaries imposed by NMP applications. Additionally, we analyze the computational time required to fit the model and predict future samples by considering six implementations and various compilation flags. Results confirm that AR models can predict future audio samples more accurately than traditional PLC approaches, which consist of filling audio gaps with silence or repeating the last received audio segment. Furthermore, results demonstrate the effectiveness of the proposed solution in meeting the strict latency requirements when deployed on a Raspberry Pi 4B.

Keywords Networked music performance · Packet loss concealment · Autoregressive models · Embedded processors

1 Introduction

Networked music performances (NMPs) entail real-time musical interactions among geographically dispersed musicians by means of low-latency streaming of audio data over the Internet. NMP is a growing research field which gained a lot of interest in recent years, fostered by the COVID-19 pandemic. During lockdown periods, traditional videoconferencing tools started to be adopted also for music-related activities, such as remote music teaching and rehearsing, thanks to their ease of use, especially when compared to the complexity of setting up and using NMP solutions available at that time. However, since videoconferencing tools are designed to ensure satisfactory conditions for traditional voice/video calls, their limitations when used in a musical context immediately emerged. Indeed, due to the excessive mouth-to-ear latency and the use of audio processing techniques specifically tailored for voice and speech [1], they yielded unacceptable quality of experience during real-time interactions between remote musicians. Therefore, the implementation of NMP applications has recently gathered increasing attention in both artistic and academic communities.

To ensure a realistic interaction between performers and an adequate musical interplay, the mouth-to-ear audio transmission latency should be kept below a few tens of milliseconds [2]. To reduce latency contributions due to audio pre-/post-processing, a common solution among NMP applications is the use of uncompressed pulse code modulation (PCM) audio, which avoids encoding and decoding delays introduced by state-of-the-art audio codecs. Furthermore, PCM audio streams are transmitted over the network leveraging

✉ Matteo Sacchetto
matteo.sacchetto@polito.it

Cristina Rottondi
cristina.rottondi@polito.it

Andrea Bianco
andrea.bianco@polito.it

1 Department of Electronics and Telecommunications, Politecnico di Torino, Corso Duca degli Abruzzi 24, Turin 10129, Italy

the user datagram protocol (UDP) at transport layer, which avoids re-transmission and reordering delays introduced by the transmission control protocol (TCP) at the price of sacrificing data-transfer reliability. Thus, no guarantees on the actual delivery of a packet or on its delivery time are provided. Note that, since NMP has strict latency requirements, packets which are received too late to be reproduced by the audio playback process are logically equivalent to lost packets, so they are discarded at the receiver. Lost packets cause gaps in the playout buffer, which in turn generate audio artifacts/glitches in the reproduced sound. Since packet loss cannot be avoided, and since retransmissions are not possible due to delay requirements, alternative packet loss concealment (PLC) mechanisms are needed to fill the audio gaps in the playout buffer with reconstructed audio segments, with the aim of mitigating the perception of audio artifacts.

In this paper, we propose a PLC method designed to run on embedded devices, that leverages autoregressive (AR) models based on Burg's method [3]. AR models are a special case of multilinear regressive models in which a forecast variable is obtained as a linear combination of its past values. The idea is to fit an AR model on the past samples of a given audio signal to predict missing samples in the playout buffer before playback. This process is repeated every time the soon-to-be-reproduced audio section does not contain any data, due to one or multiple missing audio packets.

A preliminary version of this study appears in [4], where we investigated whether using AR models could improve audio quality over traditional PLC solutions used in NMP, such as silence substitution, i.e., filling the audio gaps with silence, and pattern replication, i.e., filling the audio by replicating the audio samples contained in the last correctly received packet [5]. In [4] we performed quantitative measurements as well as subjective tests to evaluate whether AR models were able to improve the overall quality w.r.t. traditional PLC solutions used in NMP. In this paper instead, we focus on the study of the algorithmic implementation of AR models and the configuration that provides the best trade-off in terms of prediction error and computational time requirements while running the predictive model on a Raspberry Pi 4B within the NMP real-time audio boundaries, i.e., in the order of few milliseconds. Furthermore, we perform an analysis of the computational time required to fit the model and predict future samples, by considering six different algorithm implementations and different compilation flags.

Results confirm that AR models, even with low orders and a relatively short history of past audio samples, can predict future audio samples more accurately than traditional PLC solutions. We also show that running AR models in real-time on an embedded device such as the Raspberry Pi 4B is feasible with a wide enough range of configurations, which gives the system designer freedom of choice in either providing higher audio quality with higher computational cost or favoring scalability by reducing computational costs at the expense of slightly lower audio quality.

The remainder of this paper is organized as follows: after briefly reviewing the related literature in Sect. 2, some background notions on AR models and specifically on Burg's method are provided in Sect. 3. The considered algorithms are detailed in Sect. 4, with experimental setup, numerical results, and considerations on potential limitations of the proposed method presented in Sect. 5. Finally, conclusive remarks appear in the last section.

2 Related work

The adoption of AR models has been widely explored and proven to be successful for audio PLC of speech signals (see, e.g., [6]), but not yet applied in NMP scenarios. However, in music-related contexts, AR models have been applied for a variety of tasks. For example, they have been leveraged in combination with N-gram models to predict beat-synchronous Mel-frequency cepstral coefficients and chroma features in musical audio streams [7], and for online prediction of the tempo curve [8], where the parameters of the AR model are determined by a deep convolutional neural network fed with the history of the tempo curve and the music score of the piece being performed. An AR self-attention-based model for music score infilling is described in [9], to generate a polyphonic music sequence that fills in the gap between given past and future excerpts. In [10], the authors discuss how to apply Burg's method to estimate the short-term coefficients of the linear-predictive coding analysis of the OPUS codec for speech and audio. Burg's method implementation used in the OPUS codec is described in [11].

Many NMP solutions are available today (e.g., JackTrip,[1] Sonobus[2] and ELK[3]), most running on embedded devices such as Raspberry Pi Single Board Computer, due to its easiness of portability and deployment. Therefore, in this study, we focus on the implementation of AR models for PLC that can be executed also on a Raspberry Pi processor.

The majority of the studies on PLC in NMP applications focus on recovery methods for the transmission of audio signals in MIDI format. The authors of [12] proposed a PLC method leveraging a recovery journal section within RTP

[1] https://github.com/jacktrip/jacktrip

[2] https://sonobus.net/

[3] https://www.elk.audio/

packets, which encodes the most recent variations of the system status, so that a receiver can recover from a wrong status induced by the non-reproduction of MIDI events and/or commands contained in previously lost packets. Other recovery approaches rely on auxiliary reliably connected channels to transmit critical MIDI events [13] or on acknowledgment mechanisms to permit the re-transmission of lost music data [14].

Differently, in [15], a deep convolutional neural network is adopted to conceal the missing audio fragments in PCM format. The proposed ML-based framework is shown to outperform the predictive capabilities of AR models, but would require GPU-accelerated hardware to ensure acceptable execution delays, which is unlikely to be deployable in embedded devices like the Raspberry Pi 4B.

Finally, it is worth mentioning that there is a rich literature on ML-based generative models for raw audio, though not necessarily focused on NMP applications. The interested reader may refer to [16] for a comprehensive survey of the most relevant studies on such topic.

3 Background

3.1 Autoregressive models

The basic assumption behind time series regression models is the existence of a linear relationship between the time series of the variable of interest y_t and another time series x_t. The y_t variable is usually called a forecast variable, regressand, dependent, or explained variable. The x_t variable is usually called a predictor variable, regressor, independent, or explanatory variable. A linear regression model can have multiple regressor variables $x_{i,t}$. In such cases, it is called a multiple linear regression model. The general form of a multiple linear regression model with k regressor variables is expressed as follows:

$$y_t = \beta_0 + \sum_{i=1}^{k} \beta_i x_{i,t} + \epsilon_t = \beta_0 + \beta_1 x_{1,t} + \ldots + \beta_k x_{k,t} + \epsilon_t \quad (1)$$

where ϵ_t is the error term. The coefficients $\beta_0 \ldots \beta_k$ define the linear relationship between each regressor $x_{i,t}$ and the forecast variable y_t. These coefficients are estimated by fitting the model over the observed data. AR models are a special case of multiple linear regression models, where the forecast variable y_t can be expressed as a linear combination of its past values. The past values of the forecast variable are also called *lags* or *lagged values*. The number of lags, usually denoted as ρ, defines the order of the AR model, i.e., the number of past values of the time series that need to be used to predict the next sample. An AR model of order ρ, denoted as AR(ρ), is defined as follows:

$$\begin{aligned} y_t &= c + \sum_{i=1}^{\rho} a_{\rho,i} y_{t-i} + \epsilon_t \\ &= c + a_{\rho,1} y_{t-1} \quad + \ldots + a_{\rho,\rho} y_{t-\rho} + \epsilon_t \end{aligned} \quad (2)$$

where ϵ_t is typically modelled as white noise and c is a constant. The order of the model can be either chosen as a fixed value, estimated while fitting the model using a given criterion (such as the Akaike information criterion (AIC) [17]), or selected for example by extracting the number of most relevant terms from the partial autocorrelation function of time series used to fit the model.

Data stationarity is a key issue: an AR model is stable if and only if the time series is stationary; otherwise, for non-stationary time series, the model could diverge or have a biased trend. Since for audio traces none of the two scenarios is desirable, we should check if the considered time series is stationary before using AR models for audio PLC. If the time series is not stationary, we can not use AR models, and we must rely either on simple, traditional methods or on more complex alternative strategies, which are not covered in this paper. The stationarity test on the time series is not covered in this paper, as it is thoroughly discussed in [4], where details on the implementation of stationarity tests are provided.

3.2 Burg's method

The main idea behind Burg's method for maximum spectral analysis [3] is that we can estimate a set of autoregressive parameters (k_1, ..., k_ρ), also referred to as *reflection coefficients*, that minimize the sum of the squared forward and backward prediction errors over a given section of the time series. These coefficients are proven to satisfy the inequality $|k_i| \leq 1$, which guarantees the stability of the model. In [3], Burg also demonstrated that there is a one-to-one correspondence between these reflection coefficients and the coefficients of the autocorrelation function, which allows us to specify the second-order statistics of the time series. So, by directly estimating the reflection coefficients through Burg's method, we obtain a representation of the second-order statistic of the time series. Then, by using Levinson's recursion [18], we can compute the coefficients of the AR model.

We can define the i-th order forward and backward propagation errors at time n (respectively indicated as $f_{i,n}$ and

$b_{i,n}$) for a time series x_n of length N as follows:

$$f_{i,n} = a_{i,0}x_n + a_{i,1}x_{n-1} + ... + a_{i,i-1}x_{n-i+1} + a_{i,i}x_{n-i} = \sum_{j=0}^{i} a_{i,j}x_{n-j}$$
$$b_{i,n} = a_{i,0}x_{n-i} + a_{i,1}x_{n-i+1} + ... + a_{i,i-1}x_{n-1} + a_{i,i}x_n = \sum_{j=0}^{i} a_{i,j}x_{n-i+j} \quad (3)$$

with $a_{i,0} = 1, i \leq n < N$. The coefficients $a_{i,j}$ are used for the calculation of both forward and backward errors, but in reversed order. Then, we can define the sum of the forward and backward error energies as follows:

$$E_i = \frac{1}{2}\sum_{n=i}^{N-1}\{f_{i,n}^2 + b_{i,n}^2\} \quad (4)$$

This sum of errors is then minimized with respect to the AR coefficients, under the constraint that the coefficients satisfy Levinson's recursion [18]:

$$a_{i,j} = a_{i-1,j} + a_{i,i}a_{i-1,i-j} \quad 1 \leq j < i \quad (5)$$

The minimizing value of (4) is obtained by setting the derivative of E_i w.r.t. $a_{i,i}$ equal to zero, obtaining Burg's relation for the i-th reflection coefficient k_i:

$$k_i = a_{i,i} = \frac{num_i}{den_i} = \frac{-2\sum_{n=i}^{N-1} f_{i-1,n}b_{i-1,n-1}}{\sum_{n=i}^{N-1}\{f_{i-1,n}^2 + b_{i-1,n-1}^2\}} \quad (6)$$

By computing the reflection coefficients k_i as shown in (6), we can derive the AR coefficients using Levinson's recursion [18] as follows:

$$[a_{i,0}, a_{i,1}, ..., a_{i,i}] = [a_{i-1,0}, a_{i-1,1}, ..., a_{i-1,i-1}, 0] + k_i[0, a_{i-1,i-1}, a_{i-1,i-2}, ..., a_{i-1,0}] \quad (7)$$

with $a_{i,0} = 1$. Let ρ be the model order, we compute the ρ reflection coefficients (k_1,..., k_ρ), and for each reflection coefficient k_i, $1 \leq i \leq \rho$, we iterate Levinson's recursion, ultimately obtaining the AR coefficients ($a_{\rho,1}$, ..., $a_{\rho,\rho}$). The AR model coefficients are then used to predict the next element of the time series through the following:

$$x_t = \sum_{i=1}^{\rho} -a_{\rho,i}x_{t-i} \quad (8)$$

It is worth mentioning that the original Burg method [3] is defined in terms of complex numbers. Since we are applying it to audio signals, the imaginary part is not present. Thus, the above formalization has been adapted to real-valued inputs.

4 Algorithmic framework

4.1 Notation

We introduce some definitions and symbols that will be used in the description of the algorithm that implements Burg's method:

- *Training set* ($\mathcal{D}$): The section of the time series used for fitting the AR model. It contains the past data samples, based on which the model predicts the missing ones. The training set size is defined by $n = |D|$, expressed in number of samples.
- *Test set* ($\mathcal{D}_{test}$): This section of the time series contains the actual future values, which will be used as ground truth to compute the prediction accuracy metrics. The size of the test set is denoted by $n_{test} = |D_{test}|$.
- γ: The index of the first value (sample) in the time series that we want to predict. Starting from this index, we define the training set ($\mathcal{D}$) as the samples in the interval $[\gamma - n, \gamma)$, and the test set ($\mathcal{D}_{test}$) as the samples in the interval $[\gamma, \gamma + n_{test})$.

4.2 AR model implementations

We discuss the implementation of the fit and predict functions of the AR models considered in this study. In Sect. 4.2.1, a general pseudocode of an AR model based on the traditional Burg's method [3] for parameters estimation and prediction of future samples is presented. Section 4.2.2 presents the pseudocode of the optimized version of traditional Burg's method based on the order recursion formula for the denominator of (6), proposed by L. J. Faber in [19]. In Sect. 4.2.3, a novel implementation is introduced, based on the combination of both the traditional Burg's method and the recursive denominator method. Lastly, in Sect. 4.2.4, we introduce a variant of these algorithms based on error-free floating-point transformations.

4.2.1 Traditional Burg's method

The implementation of Burg's method for estimating the AR model parameters introduced in Sect. 3.2 is illustrated in Alg. 1. The algorithm starts by initializing the forward and backward prediction errors with the samples in the training set $\mathcal{D}$ (lines 1–2). Then, it initializes the array containing the AR model parameters (a) and sets its first element to 1 (lines 3–4). This is needed to compute the other model parameters through Levinson's recursion. Then, the main loop (lines 5–31) repeats until the desired order (ρ) is reached. In each iteration (i) of the loop, the reflection coefficient for the current iteration (k_i) is computed. k_i is then used to update both forward and backward prediction errors (lines 11–18), as well as the AR model parameters (lines 20–27). The coefficient computed at the current iteration ($k_i = a_{i,i}$) is then added to the array of the AR model coefficients (line 28).

Once the AR model coefficients ($[a_{\rho,0}, ..., a_{\rho,\rho}]$) are computed, they are used to predict the future values of the time series by means of Algorithm 2. Since the sole purpose of coefficient $a_{\rho,0} = a[0] = 1$ is to serve as starting point for Levinson's recursion, it is removed from the array of coefficients (line 1). The prediction loop (lines 5–12) is a simple application of the AR model definition, shown in (8), where the past samples are extracted either from samples in the training set or from previously predicted samples.

4.2.2 Denominator optimization

Though Burg's method is relatively easy to implement, it requires computing three dot products for every iteration of the main loop of the fit phase. In [19], L. J. Faber demonstrated that it is possible to rewrite the denominator of Burg's method as an order recursive formula, thus drastically reducing the complexity of the algorithm. So, according to [19], the denominator of (6) can be rewritten as follows:

$$\begin{aligned} den_i &= \sum_{n=i}^{N-1} f_{i-1,n}^2 + b_{i-1,n-1}^2 \\ &= \left[\sum_{n=i-1}^{N-1} f_{i-1,n}^2 + b_{i-1,n-1}^2\right] - f_{i-1,i-1}^2 - b_{i-1,N-1}^2 \\ &= (1-k_{i-1}^2)den_{i-1} - f_{i-1,i-1}^2 - b_{i-1,N-1}^2 \end{aligned} \quad (9)$$

This reduces the number of dot products to one product in the initialization step (line 5 of Algorithm 3) to compute the initial denominator and one product for each iteration (line 7 of Algorithm 1), to compute the numerator of k_i. The initial denominator is computed as follows:

$$den_0 = \sum_{n=0}^{N-1} f_{0,n}^2 + b_{0,n}^2 = \sum_{n=0}^{N-1} D_n^2 + D_n^2 = 2\sum_{n=0}^{N-1} D_n^2 \quad (10)$$

So, the resulting algorithm is similar to Algorithm 1 with the changes shown in Algorithm 3. Note that lines 7–33 of Algorithm 3 are identical to lines 5–12 of Alg. 1, with the only difference that line 8 of Algorithm 1 is replaced with line 10 of Algorithm 3.

This optimization only affects the fit phase. The prediction phase is identical to the one of traditional Burg's algorithm (Algorithm 2).

Algorithm 1 AR model - fit - Traditional Burg's method.

```
1: $f \leftarrow training\_set[0:n]$
2: $b \leftarrow training\_set[0:n]$
3: $a \leftarrow []$
4: $a[0] \leftarrow 1$
5: $i \leftarrow 1$
6: while $i \le \rho$ do
7:     $num \leftarrow -2f[i:n] \cdot b[0:n-i]$
8:     $den \leftarrow f[i:n] \cdot f[i:n] + b[0:n-i] \cdot b[0:n-i]$
9:     $k_i \leftarrow \frac{num}{den}$          ▷ Calculate the reflection coefficient $k_i$
10:
11:    $j = i$
12:    while $j < n$ do                          ▷ Update $b$ and $f$
13:        $tmp_b \leftarrow b[j-i]$
14:        $tmp_f \leftarrow f[j]$
15:        $b[j-i] \leftarrow tmp_b + k_i tmp_f$
16:        $f[j] \leftarrow tmp_f + k_i tmp_b$
17:        $j \leftarrow j+1$
18:    end while
19:
20:    $j = 1$
21:    while $j \le \lfloor \frac{i}{2} \rfloor$ do          ▷ Levinson's recursion
22:        $tmp_a \leftarrow a[j]$
23:        $tmp_{an} \leftarrow a[i-j]$
24:        $a[j] \leftarrow tmp_a + k_i tmp_{an}$
25:        $a[i-j] \leftarrow tmp_{an} + k_i tmp_a$
26:        $j \leftarrow j+1$
27:    end while
28:    $a[i] \leftarrow k_i$
29:
30:    $i \leftarrow i+1$
31: end while
32: return $a$
```

Algorithm 2 AR model - prediction - Traditional Burg's method.

```
1: $a \leftarrow a[1:\rho+1]$                      ▷ Remove a[0] = 1
2: $history \leftarrow train\_set[0:n]$
3: $pred \leftarrow []$
4: $n_h \leftarrow n$
5: $i \leftarrow 0$
6: while $i \le n_{test}$ do             ▷ Predict the $n_{test}$ samples
7:     $s \leftarrow history[n_h - \rho : n_h]$
8:     $pred[i] = s \cdot -a$
9:     $history[n_h + i] = pred[i]$
10:    $n_h \leftarrow n_h + 1$
11:    $i \leftarrow i+1$
12: end while
13: return $pred$
```

4.2.3 Hybrid denominator

A drawback of Burg's method is that, due to the recursive nature of the algorithm, an error on the computation of the reflection coefficients (k_i) during the first iterations propagates progressively larger errors in the next reflection coefficients. So, since the traditional implementation (Algorithm 1)

Algorithm 3 AR model - fit - denominator optimization.

```
1: $f \leftarrow training\_set[0:n]$
2: $b \leftarrow training\_set[0:n]$
3: $a \leftarrow []$
4: $a[0] \leftarrow 1$
5: $den \leftarrow 2(training\_set[0:n] \cdot training\_set[0:n])$    ▷ $f = b = training\_set$
6: $k_0 \leftarrow 0$
7: $i \leftarrow 1$
8: while $i \leq \rho$ do
9:     ...
10:    $den \leftarrow (1 - k_{i-1}^2)den - |f[i-1]|^2 - |b[n-i]|^2$
11:    ...
33: end while
34: return $a$
```

using the dot product for the denominator led to smaller numerical errors on the value of k_i, we propose a hybrid approach where for the first m iterations, with $1 \leq m \leq \rho$, we compute the denominator with the traditional algorithm (Algorithm 1), and then during the next $\rho - m$ iterations, we use the denominator optimized algorithm (Algorithm 3). In our proposed version of the algorithm, by tuning the value of m, it is possible to achieve a trade-off between accuracy and computational complexity. After extensive testing, we found that with $m = \lfloor\sqrt{\rho}\rfloor$, the errors on the coefficients computed with the hybrid approach were similar to the ones computed with the traditional method. However, a minimum number of 8 iterations was found to be necessary to ensure consistent results also for lower model orders. So, in the final implementation, we set $m = \max(\lfloor\sqrt{\rho}\rfloor, 8)$.

The resulting algorithm is shown in Algorithm 4. Lines 6–36 of Algorithm 4 are identical to lines 5–31 of Algorithm 1, with the only difference that line 8 of Algorithm 1 is replaced with lines 9–13 of Algorithm 4. Also, in this case, the prediction phase is identical to the one of traditional Burg's algorithm (Algorithm 2).

Algorithm 4 AR model - fit - Hybrid denominator.

```
1: $f \leftarrow training\_set[0:n]$
2: $b \leftarrow training\_set[0:n]$
3: $a \leftarrow []$
4: $a[0] \leftarrow 1$
5: $m \leftarrow \max(\lfloor\sqrt{\rho}\rfloor, 8)$
6: $i \leftarrow 1$
7: while $i \leq \rho$ do
8:     ...
9:     if $i \leq m$ then
10:        $den \leftarrow f[i:n] \cdot f[i:n] + b[0:n-i] \cdot b[0:n-i]$
11:    else
12:        $den \leftarrow (1 - k_{i-1}^2)den - |f[i-1]|^2 - |b[n-i]|^2$
13:    end if
14:    ...
36: end while
37: return $a$
```

4.2.4 Error compensation

In the implementation of Burg's method in a computer program, floating-point arithmetic is used. Due to the representation of a real number on a limited number of bits, all operations introduce a negligible approximation error. However, when combined in a recursive way, they can result in a cumulative error which may become no longer negligible. Thus, to reduce the magnitude of the error, we propose to implement all of the above-described variants of Burg's algorithm using error-free transformations in floating-point arithmetic.

Error-free floating-point transformation is a concept that makes it possible to compute accurate results within a floating-point arithmetic. Ref. [20] describes a number of error-free transformations for the sum and multiplication of floating-point elements, whereas Ref. [21] recalls well-known error-free transformations for real and complex numbers. Among those, we opted for the usage of TwoSum, TwoProductFMA, and the Dot2, which allows performing sum, product, and dot product with doubled precision. Thus, we implemented all the variants of Burg's method replacing sums, products, and dot products with the transformations mentioned above. The operations performed by the three chosen error-free transformations are reported in Algorithms 5–7 for completeness. The interested reader is referred to the aforementioned references for a thorough discussion.

Algorithm 5 TwoSum.

```
1: function TwoSum(a, b)
2:     $x \leftarrow fl(a+b)$
3:     $z \leftarrow fl(x-a)$
4:     $y \leftarrow fl((a-(x-z))+(b-z))$
5:     return x, y
6: end function
```

Algorithm 6 TwoProductFMA.

```
1: function TwoProductFMA(a, b)
2:     $x \leftarrow fl(ab)$
3:     $y \leftarrow FMA(a, b, -x)$
4:     return x, y
5: end function
```

5 Numerical assessment

In this section, after a brief description of the experimental setup and dataset, we present the metrics and the methods used for training and testing the AR models. Numerical

Algorithm 7 Dot2.

```
1: function Dot2(v1, v2, N)
2:     $[p, s] \leftarrow TwoProductFMA(v1[0], v2[0])$
3:     for $i \leftarrow 1 : N - 1$ do
4:         $[h, r] \leftarrow TwoProductFMA(v1[i], v2[i])$
5:         $[p, q] \leftarrow TwoSum(p, h)$
6:         $s \leftarrow fl(s + (q + r))$
7:     end for
8:     return $fl(r + s)$
9: end function
```

results are then described, whereas the final subsection discusses the main limitations of AR models for PLC.

5.1 Experimental setup

As mentioned in Sect. 1, the goal of this study is to implement a PLC solution for NMP based on AR models executable in real-time on embedded devices, specifically on a Raspberry Pi 4B. In our setup, the Raspberry Pi 4B is running the Raspberry Pi OS lite version 64-bit edition and is configured to turbo the four cores ARM cortex-a72 CPU to 2.1GHz, by setting the following parameters in the /boot/config.txt.

```
over_voltage=6
arm_freq=2100
```

On the Raspberry Pi 4B, we run the three implementations of AR models based on Burg's method described in Sect. 4.2. For each implementation, we consider multiple parameter configurations. The considered AR models are benchmarked against the two most commonly used PLC solutions in the NMP field: silence substitution and pattern replication. Furthermore, we measure the computational time that each model requires for both fitting and predicting the missing packets. Tests are run considering several audio files, grouped into six different categories based on the musical instrument(s) involved. Finally, an analysis of the performance of the various models on longer prediction periods is presented, to show the limitation of the model and how the error propagates in the different implementations.

5.2 Dataset

The dataset we used is a collection of 29 audio files. Among those, 23 audio files contain individual instruments or human voices, while 6 audio files contain full songs. We grouped the audio files in six categories, based on the audio type they contain. The first four categories are Violin, Drums, Piano, and Guitar, each containing 4–5 audio files. These four categories contain only audio files associated with a single instrument. The fifth category, named Generic, contains audio files relative to single audio sources which were randomly chosen and did not fit in any of the other five categories. In this category, audio files contain either a spoken/singing voice or a horn. The last category is named Songs and encompasses six audio files with full songs, containing a mixture of different instrumental sources. This division in categories allows us to have a more general representation on the behavior of the PLC techniques when applied in different scenarios.

Since in NMP the most common scenario involves sending one or multiple mono audio channels, each one related to a different instrument (e.g., guitar and voice), to reduce the required bit-rate and the amount of equipment needed, we decided to perform a pre-processing step to convert all audio files from stereo to mono. This allows us to evaluate the performance of AR models when used to recover individual mono audio channels and makes the proposed approach generalizable to any audio channel configuration (mono, stereo, multichannel), since we can run an AR model individually for each audio channel. Alternative approaches based on vector autoregressive (VAR) models [22, 23] could be considered to jointly work on multiple channels. The inclusion of such approaches in our implementation is left for future work. Moreover, to standardize the audio file format and simplify the loading of audio files in the specific programming language we used (i.e., C++), we converted these files to WAV files with a sample rate of 44,100 Hz, commonly used in NMP applications. The pre-processing was performed using `FFmpeg`.[4]

5.3 Performance metrics

We evaluate the performance using two different metrics: mean absolute error (MAE) and root mean square error (RMSE). In the context of time series analysis, MAE is a model evaluation metric which measures the mean absolute distance between predicted values and target values of a time series y. Given a section of the time series, which we define as *test set* ($\mathcal{D}_{test}$), the MAE is defined as the sum of per-sample absolute errors divided by the number of samples in the test set. Defined as y_t the true target value of the time series at time t, y'_t the predicted value at the same time instant, and n_{test} the size of the test set, we can define the MAE as follows:

$$MAE(y', y) = \frac{\sum_{t=1}^{n_{test}} |y'_t - y_t|}{n_{test}} = \frac{\sum_{t=1}^{n_{test}} |e_t|}{n_{test}} \quad (11)$$

[4] https://ffmpeg.org/

The RMSE instead is defined as the square root of the mean square error (MSE), which measures the quadratic distance between predicted values and target values of the time series. The MSE is defined as the sum of per-sample quadratic errors divided by the number of values in the test set. The RMSE is simply the square root of that ratio. So, given the same notation previously introduced, we can define the RMSE as follows:

$$RMSE(y', y) = \sqrt{MSE(y', y)} = \sqrt{\frac{\sum_{t=1}^{n_{test}} (y'_t - y_t)^2}{n_{test}}} \tag{12}$$
$$= \sqrt{\frac{\sum_{t=1}^{n_{test}} (e_t)^2}{n_{test}}}$$

5.4 Training and testing the AR models

A typical UDP packet size for NMP accommodates 128 audio samples per audio channel. Smaller sizes may result in excessive overhead. Instead, larger sizes would increase the mouth-to-ear latency due to the increase in the time needed to initially fill the playout buffer before starting the audio playback. Since our PLC is designed specifically for NMP scenarios, we set $n_{test} = 128$. Notably, this value allows for potential future generalization of the proposed PLC solution to NMP web applications. Those applications take advantage of the Web Audio API,[5] in particular the AudioWorklet component,[6] for audio processing, which operates on buffers of 128 samples. The interested reader is referred to [1] for technical details regarding NMP with web technologies (WebRTC, WebAudioAPI). Regarding n, i.e., the size of the train set ($\mathcal{D}$), since we are working with packets of 128 audio samples per audio channel, we set it as an integer multiple of that size, i.e., $n = m \cdot 128, m \in N, m > 0$. The actual values we used for n are 512, 1024, 2048, 4096, and 8192. Finally, since we aimed at predicting 128 samples, for the model order ρ, we explored as values all powers of 2 in the range [1, 128], i.e., $\rho = 2^e, e \in N, 0 \leq e \leq 7$.

Firstly, we want to evaluate the performance of every algorithm in terms of the MAE and RMSE metrics, for each of the six dataset categories. For every audio file, after loading all the samples, we extract 100 distinct random positions γ, multiple integers of n_{test}, $\gamma = m \cdot n_{test}, m \in N, m > 0$. This number was selected to have the same number of predictions per audio file. Extracting random positions multiples of n_{test} allows us to simulate a packetized audio stream. For each γ, we fit the AR model on the n previous samples, i.e., on set $\mathcal{D}$, and then we use the fitted AR model to predict the next n_{test} samples. Next, we compute the two metrics over the predicted samples and the actual samples, i.e., $\mathcal{D}_{test}$. We also compute MAE and RMSE for the two benchmark PLC methods. Algorithm 8 provides the pseudocode that implements the above-described procedure. This algorithm needs to be run for every AR model implementation.

Secondly, we want to evaluate the computational requirements of the different algorithms presented in Sect. 4.2. To do so, we need to measure the time required to run the fit and predict methods for each algorithm, for every value of ρ and n. Algorithm 8 already includes instructions for the measurement of the computational time of the fit and predict phases.

Finally, we want to investigate how the different models perform on longer predictions. This is helpful to understand how error propagates in the three considered implementations. This test is carried out by considering a sine wave with a fixed frequency of 2 kHz and an amplitude covering the whole audio range. For each model and every value of ρ and n, we train the model on $\mathcal{D}$ and then predict the next 20 packets, i.e., $n_{test} = 20 \cdot 128$. Every run of this algorithm uses the same value of γ, set to the max value of n, i.e., 8192. We then compute the MAE to compare the predicted samples to the actual samples, i.e., $\mathcal{D}_{test}$.

5.5 Implementation details

The implementation of all the algorithms described in this paper is done in C++, using the C++17 standard. As already noted in [19], a double-precision floating point was found to be necessary. Thus, we run all the algorithms with double-precision floating-point (`double`) data type and quadruple-precision floating-point (`long double`) data type. Note that the error-compensation techniques explained in Sect. 4.2.4 are applied only when using the double-precision floating-point data type. For the quadruple-precision floating-point data type, since already working with twice the working precision of the double-precision floating-point data type, additional error compensation is not necessary.

The C++ code was compiled directly on the Raspberry Pi 4B using g++ 10.2.1 on aarch64, the ARM 64 bit architecture. Compilation was performed using the gcc compilation flags `-O3` and `-mtune=cortex-a72`. The `-O3` flag enables the highest compiler optimization level (3), which allows the compiler to perform operations that reduce code execution time. The `-mtune=cortex-a72` hints the compiler the

[5] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

[6] https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletProcessor

Algorithm 8 Training and testing.

```
$n_s \leftarrow [512, 1024, 2048, 4096, 8192]$
$\rho_s \leftarrow [1, 2, 4, 8, 16, 32, 64, 128]$
$n_{test} \leftarrow 128$

for $category$ in $categories$ do
    for $file$ in $audio\_files$ do
        $samples \leftarrow load\_samples(file)$
        $i_s \leftarrow generate\_100\_random\_indexes(samples)$

        for $i$ in $i_s$ do
            $silence \leftarrow zeros(n_{test})$
            $previous \leftarrow samples[i - n_{test} : i]$
            $\mathcal{D}_{test} \leftarrow samples[i : i + n_{test}]$

            $store(MAE(\mathcal{D}_{test}, silence))$    ▷ Store the benchmark results
            $store(RMSE(\mathcal{D}_{test}, silence))$

            $store(MAE(\mathcal{D}_{test}, previous))$
            $store(RMSE(\mathcal{D}_{test}, previous))$
        end for

        for $n$ in $n_s$ do
            for $\rho$ in $\rho_s$ do
                for $i$ in $i_s$ do
                    $\mathcal{D} \leftarrow samples[i - n, i]$
                    $\mathcal{D}_{test} \leftarrow samples[i, i + n_{test}]$

                    $AR\_model \leftarrow AR(n)$

                    $start \leftarrow start\_timer()$
                    $a \leftarrow AR\_model.fit(\mathcal{D}, \rho)$
                    $store(stop\_timer() - start)$    ▷ Store fit time

                    $start \leftarrow start\_timer()$
                    $pred \leftarrow AR\_model.predict(\mathcal{D}, a, n_{test})$
                    $store(stop\_timer() - start)$    ▷ Store predict time

                    $store(MAE(\mathcal{D}_{test}, pred))$    ▷ Store the AR results
                    $store(RMSE(\mathcal{D}_{test}, pred))$
                end for
            end for
        end for
    end for
end for
```

specific processor architecture for which the code has to be built.

During the testing, we experimented with another compiler flag: `-ffast-math`. This flag is useful in programs using floating-point mathematical operations, to allow the compiler to perform additional optimizations, by skipping some checks and making assumptions on the code. We report an extract of some of the optimizations it enables, taken from the documentation[7]: "This mode enables optimizations that allow arbitrary reassociations and transformations with no accuracy guarantees. It also does not try to preserve the sign of zeros," thus breaking strict IEEE compliance on floating-point representation. It follows that its use may not be appropriate for all applications. One thing to note about `-ffast-math` is that, since it allows for "arbitrary reassociations and transformations with no accuracy guarantees," it can not be applied to the error-compensated versions of the algorithms, since the error-free floating-point transformations shown in Sect. 4.2.4 are based on a specific execution order of the provided instructions.

For details on the optimization flags, please refer to section "Optimize Options" of the online documentation.[8] Instead, for details on compilation flags specific to the ARM processor architecture, please refer to section "ARM Options"of the online documentation.[9]

The experimental datasets and all the code developed and used for the purpose of this study are publicly available on GitHub.[10]

5.6 Numerical results

5.6.1 Evaluation of the prediction error

Figures 1 and 2 show, for each audio category, the MAE and RMSE trend of the AR models compared to that of the two benchmarks, for multiple values of ρ. The plots shown in Figs. 1 and 2 are obtained with the traditional Burg's method (Sect. 4.2.1), but identical results were observed for the other two algorithms (Sects. 4.2.2–4.2.4). Thus, in this subsection, we will adopt the general term "AR models" instead of referring to each individual algorithm.

Since the two benchmarks do not have any dependency on n and ρ, their MAE/RMSE values are displayed as horizontal lines, with shaded bands to indicate confidence intervals. The gray line represents the error obtained using the silence substitution technique, whereas the yellow line represents the error obtained with the pattern replication technique. The other lines, instead, display the error measured with AR models for different values of n. The error of the pattern replication technique is generally higher than the one of silence substitution, coherently with results reported in [4], due to phase offsets created by replicating the previous n_{test} samples. Furthermore, the variation of n only mildly affects the prediction error, with lower values of n resulting in a slightly lower error. This has two main effects. First, the possibility of using lower values of n limits the number of past

[7] https://gcc.gnu.org/wiki/FloatingPointMath

[8] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[9] https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html

[10] https://github.com/matteosacchetto/burg-implementation-experiments

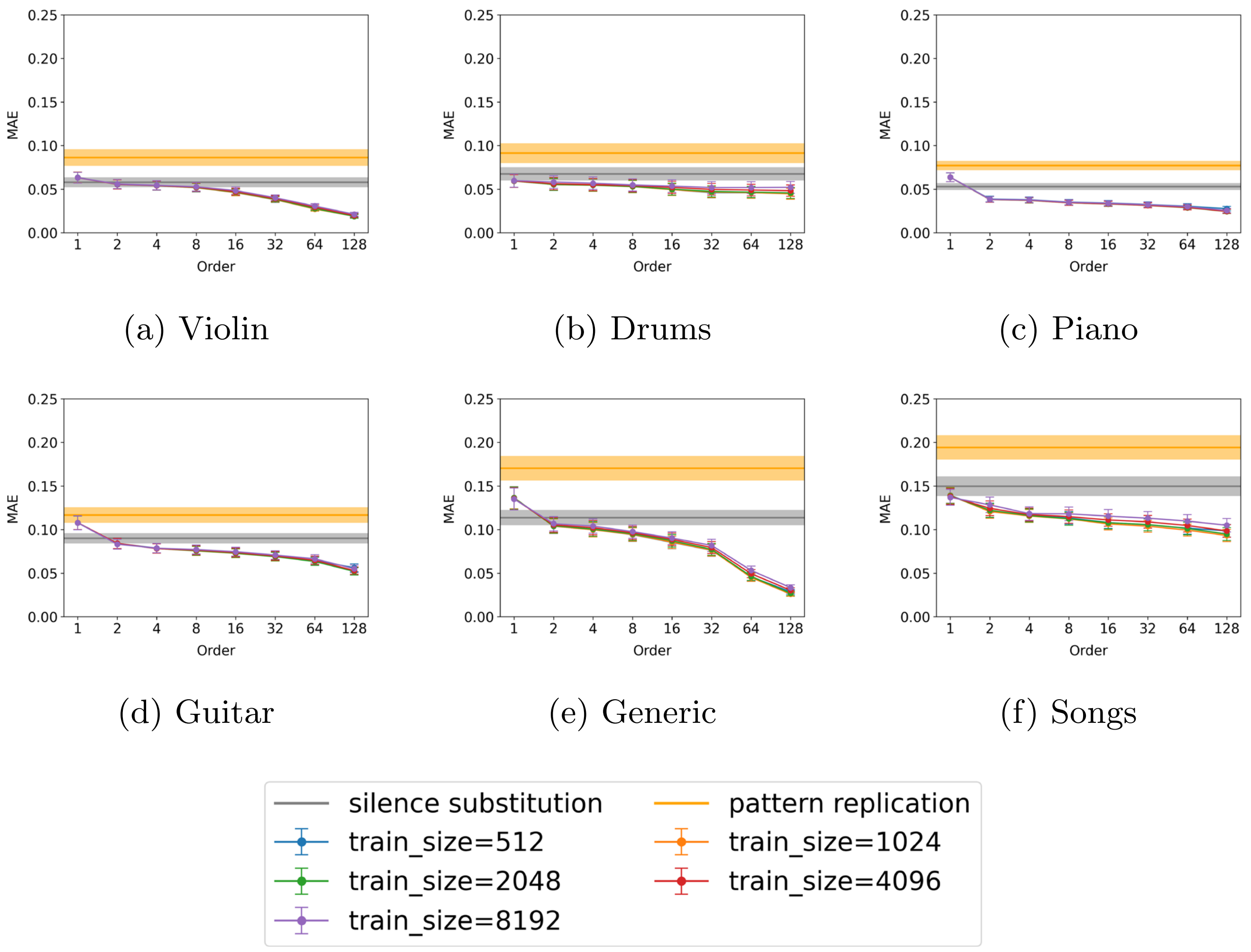
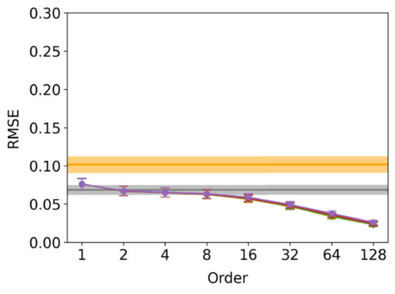


Fig. 1 Performance of AR models in terms of MAE

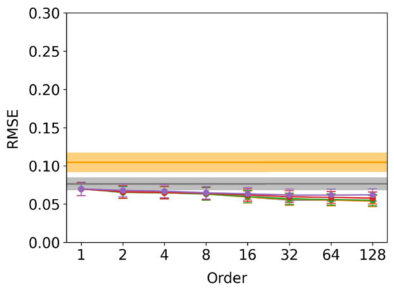
packets to be memorized, which reduces memory consumption. Second, in Burg's method, to calculate the numerator and denominator of (6), we compute the dot products of f and b. Since f and b depend on n, smaller n means smaller size vectors to multiply, hence lower computational cost.

Results show that ρ is the most impacting parameter to tune: generally, the higher ρ, the lower the error. However, increasing ρ means increasing also the computational cost of both fitting the model and predicting the next samples. Thus, in a real scenario, a trade-off emerges between prediction quality and computational cost. Note that, depending on the input signal, it is not always necessary to select high values of ρ, since in some cases the improvement in prediction quality is negligible, while the increase in computational cost is significant. Thus, to use AR models for PLC, we first need to find the best configuration of n and ρ, according to our specific runtime context (i.e., computational resources, timing constraints). Then, we could use a criterion such as AIC [17] to allow for a potential early stop of the fitting phase, with the aim of reducing computational cost for easy-to-predict signals.

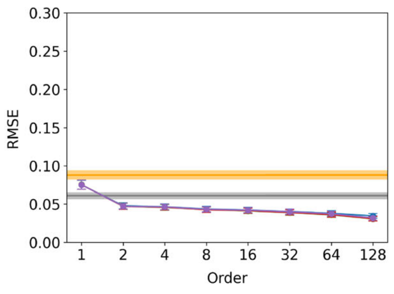
In general, we can see that, the more sustained the sound is, the better AR models are able to reconstruct missing samples. Conversely, the more transient-oriented the sound is, the less effective AR models are, showing a trend very similar to the one obtained by silence substitution. This can be clearly observed in Figs. 1 and 2, where in the categories Violin, Piano, and Generic, all of which contain more sustain-oriented sounds with less abrupt transitions, AR models show a greater improvement w.r.t. silence substitution and pattern replication. Instead, transient-oriented sounds like those in the categories Guitar (which contains some



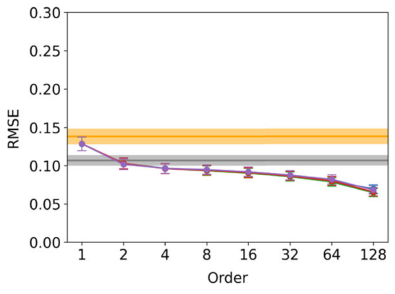
(a) Violin



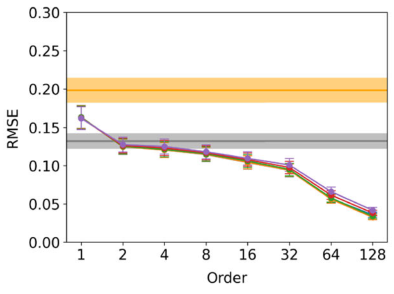
(b) Drums



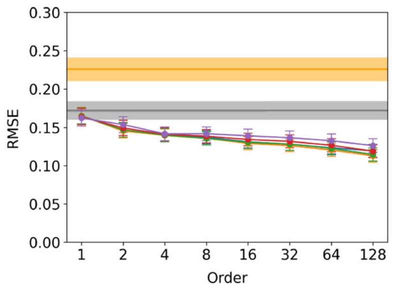
(c) Piano



(d) Guitar



(e) Generic



(f) Songs

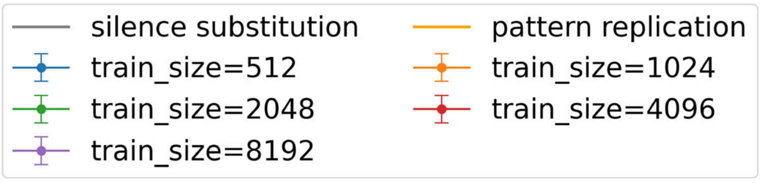
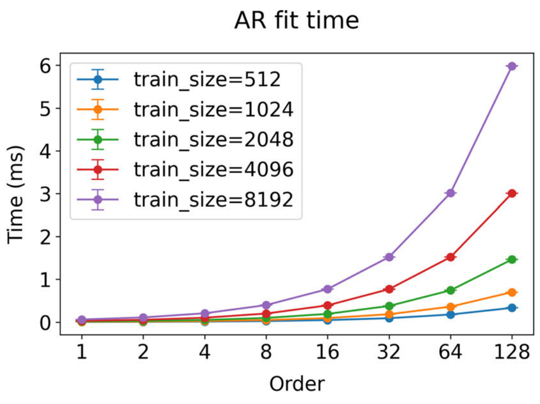
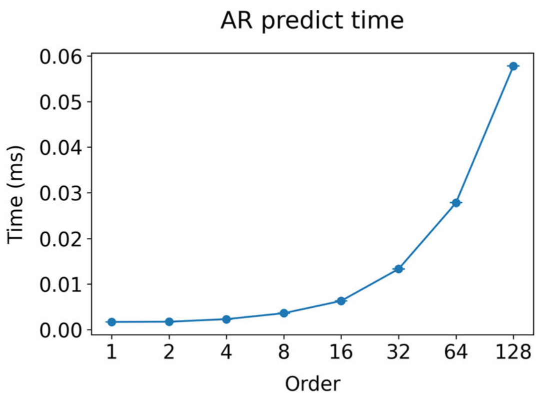


Fig. 2 Performance of AR models in terms of RMSE



(a) Fitting time trend for the traditional Burg's method (Sec 4.2.1) with doubles and no additional compilation flags



(b) Prediction time trend for the traditional Burg's method (Sec 4.2.1) with doubles and no additional compilation flags

Fig. 3 AR models—computational time trend

Table 1 Time (ms) required to fit the specified algorithm on a training set of size n, with ρ set to 64 and data type set to `double`

Algorithm	Fit time (ms)				
	n=512	n=1024	n=2048	n=4096	n=8192
Burg's method	0.178	0.362	0.746	1.520	3.024
Denominator optimization	0.091	0.182	0.378	0.771	1.528
Hybrid denominator	0.102	0.205	0.424	0.866	1.720
Burg's method (`-ffast-math`)	0.114	0.228	0.477	1.007	1.985
Den. opt. (`-ffast-math`)	0.069	0.136	0.292	0.624	1.227
Hybrid den. (`-ffast-math`)	0.075	0.148	0.315	0.672	1.320
Burg's method (compensated)	0.795	1.628	3.327	6.706	13.385
Den. opt. (compensated)	0.473	0.962	1.967	3.945	7.878
Hybrid den. (compensated)	0.515	1.049	2.139	4.292	8.567

strumming tracks), Songs (which features multiple instruments, among which drums), and, most notably, Drums (i.e., the most transient-oriented sources), the improvement is far less noticeable. Especially in Drums, we can see that AR models have a trend very close to that of silence substitution. This is due to the fact that AR models are not able to learn any significant pattern, and the resulting predictions often quickly decay to zero. For a thorough discussion on these aspects, as well as for a presentation of the results of subjective tests comparing the performance of AR models to that of the two traditional PLC methods used in NMP, the interested reader is referred to our previous work [4]. The dataset used is the same one used in this paper. While the parameter estimation method for AR models adopted in this study is slightly different, results are rather similar. Additionally, the reconstructed audio files using the three PLC techniques (silence substitution, pattern replication, and AR models with the hybrid denominator implementation, n=2048 and ρ=[64, 128]) are stored in the publicly accessible GitHub repository.[11]

Notably, Figs. 1 and 2 show that, while the improvement in prediction quality varies from category to category, AR models tend to outperform the two benchmark approaches already with $\rho = 4$. This means that, in a real scenario, there is a fairly wide range of configurations we can choose from; in other words, we can adapt in real time the AR model parameters to the current runtime conditions.

5.6.2 Evaluation of computational requirements

Given a number of samples n_s and a sampling rate f, to compute the time interval in which the n_s samples are generated/consumed, we can simply divide n_s by f:

$$t_s = \frac{n_s}{f} \tag{13}$$

In our specific case, $n_s = n_{test} = 128$ and $f = 44100$Hz. Thus, the NMP application generates and consumes packets every $\frac{128}{44,100}$ s ≈ 2.90 ms. This means that, in order to apply our PLC solution, we need to fit the AR model and predict the next n_{test} samples within 2.90 ms.

The general trend is shown in Fig. 3a, in terms of computational time for fitting the model, and in Fig. 3b in terms of computational time for predicting the next n_{test} samples. Both plots were generated using the traditional Burg's method (Sect. 4.2.1) with double-precision floating point and no `-ffast-math` compiler flag, but similar trends were observed for the other variants (Sect. 4.2.2-4.2.4) and configurations.

In general, replacing the traditional Burg method (Algorithm 1) with Burg's method with denominator optimization (Algorithm 3) reduces the fitting time of a factor between 1.5 and 2. Instead, substituting the traditional Burg method (Algorithm 1) with our proposed hybrid method (Algorithm 4) reduces the fitting time by a factor between 1.3 and 1.8 for$\rho \geq 16$ and has the same fitting time for $\rho \leq 8$. As expected, the prediction trend instead does not change among the three variants.

The error-compensated versions of the algorithm are 4 to 5 times slower during the fitting phase than their corresponding non-error-compensated version, and 2 times slower during the prediction phase. The quadruple-precision floating-point (`long double`) version of the algorithms instead is between 80 and 100 times slower during the fit phase and about 40 times slower during the predict phase than their double-precision floating-point (`double`) equivalent.

[11] https://github.com/matteosacchetto/burg-implementation-experiments

Table 2 Time (ms) required to fit the specified algorithm on a training set of size n, with ρ set to 128 and data type set to `double`

Algorithm	Fit time (ms)				
	n=512	n=1024	n=2048	n=4096	n=8192
Burg's method	0.335	0.703	1.466	3.009	5.996
Denominator optimization	0.172	0.352	0.743	1.519	3.018
Hybrid denominator	0.188	0.385	0.807	1.649	3.280
Burg's method (`-ffast-math`)	0.216	0.444	0.939	1.994	3.933
Den. opt. (`-ffast-math`)	0.132	0.267	0.573	1.230	2.423
Hybrid den. (`-ffast-math`)	0.141	0.283	0.606	1.296	2.552
Burg's method (compensated)	1.505	3.170	6.563	13.305	26.637
Den. opt. (compensated)	0.905	1.882	3.886	7.826	15.665
Hybrid den (compensated)	0.962	2.000	4.122	8.300	16.611

Regarding compilation flags, `-ffast-math` helped reduce the computation cost by a factor between 1.2 and 1.4. Note that this holds only for the double-precision floating-point version of the algorithms, whereas for the quadruple-precision floating-point version it had no impact. Since on the Raspberry Pi 4B the quadruple-precision floating-point data type is library provided, and it is not directly backed by hardware, it is very expensive to operate on such device, and additional compiler optimizations provided by `-ffast-math` are not possible. In our tests, though using quadruple-precision floating-point parameters may help the AR models to reduce the prediction error, running them in real-time on such devices is not feasible. With $n = 512$ and $\rho = 16$, we already reached the 2.90 ms constraint, with only Burg's denominator optimized algorithm (Alg. 3) being able to stay within our deadline, with a timing of 2.47 ms for the fitting phase and 0.21 ms for the prediction phase. The same algorithm, with the most demanding configuration ($n = 8192$ ans $\rho = 128$), required 305 ms for the fitting phase and 1.65 ms for the prediction phase. So, from now on, we will focus only on the double-precision floating-point version of the algorithms.

Table 3 Time (ms) required to predict the next n_{test} (128) samples, with ρ set to 64 and 128 and data type set to `double`

Algorithm	Prediction time (ms)	
	ρ=64	ρ=128
Burg's method	0.028	0.058
Denominator optimization		
Hybrid denominator		
Burg's method (`-ffast-math`)	0.022	0.050
Den. opt. (`-ffast-math`)		
Hybrid den. (`-ffast-math`)		
Burg's method (compensated)	0.060	0.124
Den. opt. (compensated)		
Hybrid den. (compensated)		

Tables 1 and 2 show the computational time required to fit the various models for different values of n, whereas ρ is set to 64 and 128, respectively. Table 3 shows instead the computational time required by the various models to predict the next n_{test} (128) samples, with ρ set to either 64 or 128. All results shown in the aforementioned tables have an absolute error $\leq \pm 3\%$.

Numerical results show that 2048 is a good value for the training set size, because it allows us to freely change the value of ρ in the range 1 to 128 while not exceeding the 2.90 ms deadline. The denominator optimized version (Sect. 4.2.2) and the hybrid version (Sect. 4.2.3) are very similar in terms of computational time and are about 40% faster during fitting than the traditional implementation (Sect. 4.2.1), thus allowing for larger values of n. In terms of compilation flags, `-ffast-math` showed to not be necessary to meet the 2.90 ms deadline, but it is helpful in cases where additional optimization is needed, at the expense of breaking IEEE compliance on floating-point representation.

5.7 Evaluation of the prediction error on long predictions

Figure 4 shows, for every AR model algorithm and various values of n, the MAE over a long prediction of 20 packets of 128 samples each (2560 samples in total), for different values of ρ. In the plots, where the MAE goes above the y-axis range, the model diverged with that configuration, drastically exceeding the interval $[-1, 1]$ used for audio floating-point representation. Based on the plots, it emerges that, while error compensation may help on longer predictions, this effect is not deterministic. For example, in Fig. 4e, error compensation helped to avoid model divergence with $n = 2048$, which was instead present in Fig. 4b, but the error compensated model ended up diverging for $n = 1024$. Thus, in general, the benefits that error compensation may bring are negligible when compared to the additional computational cost.

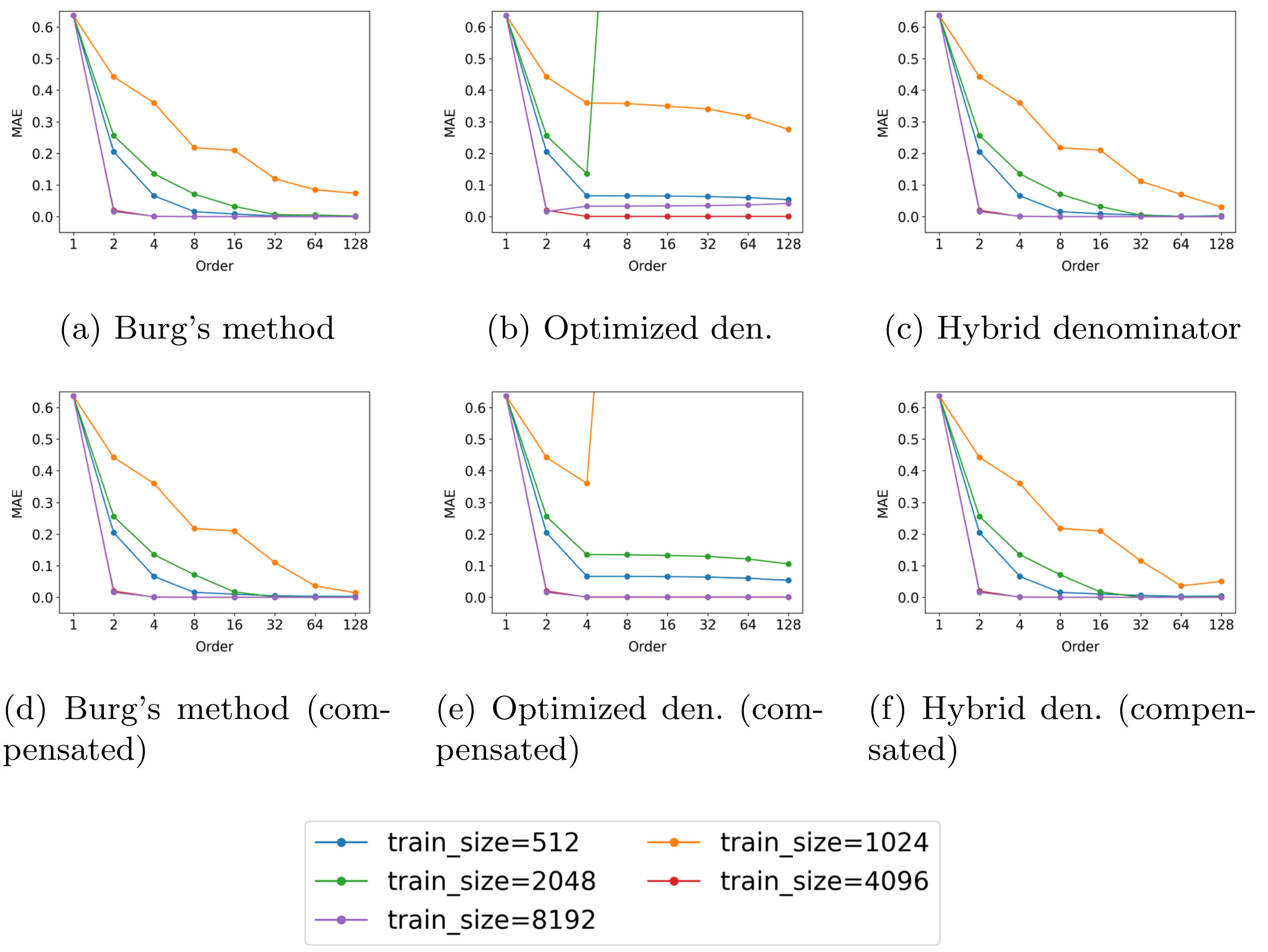


Fig. 4 AR models—long prediction MAE

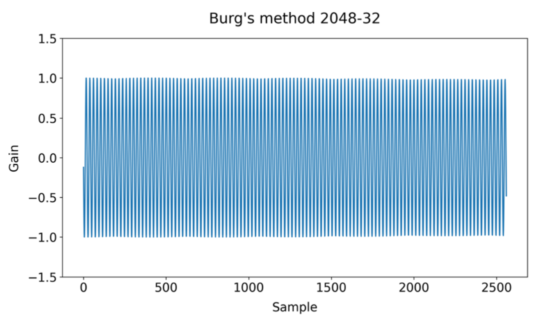
Moreover, depending on the implementation, the parameters, and the training data, numerical stability may not be granted.

Figure 4 b and e show another interesting effect. In Burg's denominator optimized algorithm (Algorithm 3), for every value of n, the model does not improve the prediction error beyond a certain value of ρ. This is a consequence of the errors made in estimating the initial k_i reflection coefficients, which lead to progressively smaller improvements with the increase of the model order. Interestingly, this does not happen for Burg's hybrid denominator version (Fig. 4c and f), which follows more closely the trend of the traditional Burg method (Fig. 4a and d).

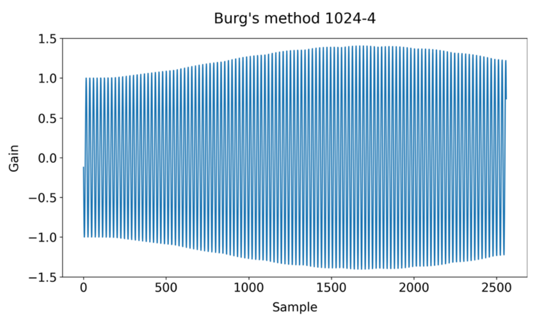
Thus, the proposed Burg hybrid algorithm (Algorithm 4) has the advantage of following more closely the error propagation trend of the traditional Burg method, for longer predictions, while incurring in a computational cost only slightly higher than that of Burg's denominator optimized algorithm (Algorithm 3), allowing for significant savings in comparison to the traditional method.

5.8 Limitations

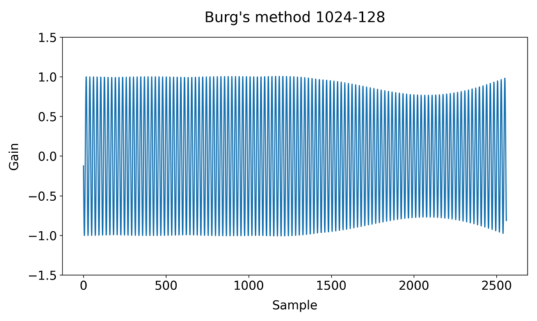
While AR models using Burg's method proved to outperform the two benchmark solutions, they have some limitations. The first one is in terms of computational requirements: though AR models are pretty light in terms of memory, they are more demanding in terms of CPU, especially during the fitting phase (the prediction phase is about one to two orders of magnitude lighter). Through algorithm optimizations, like the denominator optimization illustrated in Sect. 4.2.2, and the usage of the appropriate compilation flags provided by the specific language compiler in use, CPU usage can be tamed. However, even if optimized, they are still computationally intensive, which means that the scalability of this solution is not as good as the one of the two benchmark PLC solutions. Thus, if used in the context of a NMP application, some trade-offs emerge. In a real deployment, a possible approach could be to set an upper bound on the maximum number of AR models to be executed in parallel, associated with the network



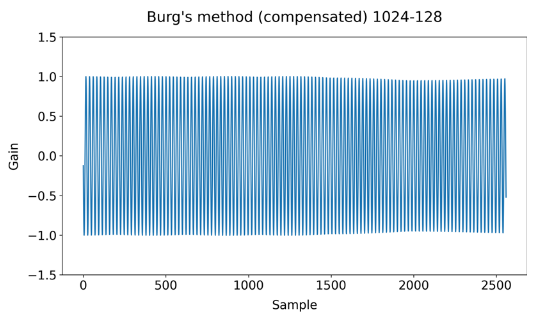
(a) Burg's method with n=2048 and ρ=32



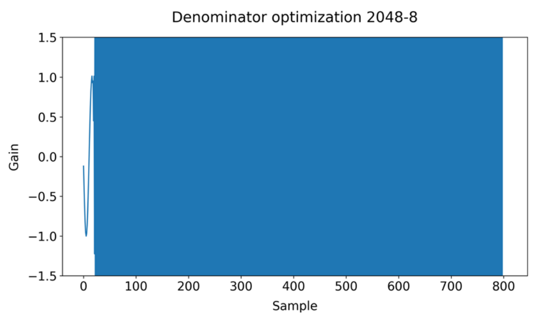
(b) Burg's method with n=1024 and ρ=4



(c) Burg's method with n=1024 and ρ128



(d) Burg's method (compensated) with n=1024 and ρ=128



(e) Optimized denominator with n=2048 and ρ=8

Fig. 5 Resulting waveform for AR-based predictions of 2560 samples (20 packets of 128 samples) of a 2-kHz sine wave

channels that are more likely to experience packet losses (this choice could be done based on packet loss statistics collected over the past history and updated in real-time).

The second limitation, which is specific to Burg's method implementation, is that every time a new packet is received, the whole model needs to be fitted again on the new training set. This can be mitigated, e.g., by fitting the model upon detection of a lost packet and then using it to predict the next M packets (without re-fitting the model on a per-packet basis), where M should be carefully chosen so that the signal stationarity is likely to be maintained within such time window. The investigation of solutions that make possible incremental incorporation of new audio samples without requiring a new fitting procedure from scratch on the whole training set is left for future work.

The third limitation of AR models is that numerical stability may not be granted, as discussed in Sect. 5.7, and it depends on the implementation, the parameters, and the training data. To show this effect, we fit different implementations and configurations on a sine wave with a fixed frequency of 2 kHz. We then used the various models to predict the next 2560 samples (20 packets of 128 samples). In Fig. 5, we report some examples of how the different implementations and configurations behaved. Sometimes, error compensation may help, as reported in Fig. 5c and d. These two plots were obtained with the same algorithm and parameters, with the only difference that in Fig. 5d, error-compensated floating-point transformations were adopted. Nevertheless, as shown in Fig. 5a, b, and e, numerical divergence heavily depends on the AR model algorithm and parameters used.

In Fig. 5a, we obtained an almost perfect reconstruction of the input signal, while Fig. 5b reports a sine wave which exceeds the range $[-1, 1]$, which, once played back, would result in auditory distortions due to audio converters clipping. Figure 5e, instead, shows that after 20 samples amplitude values ultimately explode to $[-\infty, +\infty]$, and after 800 samples, the algorithm aborted. Countermeasures to avoid the numerical stability problems need to be applied. A possible approach could be to pre-check the generated samples and, if they exceed the range $[-1, 1]$, but are still limited (like in Fig. 5b), an amplitude rescaling could be applied. Conversely, if they diverge to $[-\infty, +\infty]$, a fallback solution could be to use silence substitution, to avoid generating loud distortions which could possibly damage the equipment or the ears of the listeners.

The last consideration about AR models is that, while they have the benefit of creating a prediction which ensures continuity with the past samples, they have no guarantees of continuity with future samples. Thus, in order to avoid creating some discontinuities between the predicted sections and the next actual section, which result in audio glitches, some strategies need to be implemented. A well-known strategy to smooth out the discontinuities is to predict a number of samples which is greater than the ones in the missing section and then use a simple cross-fade on the overlapping samples to fade out the predicted samples and fade in the actual samples of the next section [24–26].

6 Conclusions

In this paper, we propose a PLC solution based on AR models and targeted towards NMP applications. In particular, we focus on exploring different implementations of AR models based on Burg's method for AR model fitting and propose a novel variant that mitigates error propagation during the recursive execution of the prediction algorithm, while preserving computational scalability. Since many NMP solutions are also built to run on a Raspberry Pi SBC, the focus of this paper is to propose a PLC solution which is able to run on the latest Raspberry Pi SBC currently available (i.e., the Raspberry Pi 4B). Experiments were performed considering various categories of audio signals, exploring various choices of parameters and different implementations. Results showed that, even with small orders, AR models outperform the prediction performance of the two benchmark PLC techniques in terms of MAE and RMSE. Concerning computational timings, results showed that a training set of 2048 lags is the highest value which guarantees the completion of the model fitting and prediction of the next 128 samples within 2.90 ms, for model orders up to 128. Moreover, using either the denominator optimization or the hybrid denominator versions of Burg's method reduces the computation cost by a factor of 1.3 to 2, thus increasing the scalability of the proposed solution.

Finally, we want to acknowledge that, while this PLC method and its implementation were optimized and tested with NMP as target application, it could be employed in any other scenario where low-latency streaming of music/audio material is involved.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement. This work has been supported by the Italian Ministry for University and Research under the PRIN program (grant n. 2022CZWWKP).

Data Availability The dataset analyzed during the current study, all the code developed for and presented in this study, results, and processed audio files are publicly available on GitHub at https://github.com/matteosacchetto/burg-implementation-experiments

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Sacchetto M, Gastaldi P, Chafe C, Rottondi C, Servetti A (2022) Web-based networked music performances via WebRTC: a low-latency PCM audio solution. J Audio Eng Soc 70(11):926–937. https://doi.org/10.17743/jaes.2022.0021
2. Rottondi C, Chafe C, Allocchio C, Sarti A (2016) An overview on networked music performance technologies. IEEE Access 4:8823–8843. https://doi.org/10.1109/ACCESS.2016.2628440
3. Burg, JP (1975) Maximum entropy spectral analysis. PhD thesis, Stanford University. Available at http://sepwww.stanford.edu/data/media/public/oldreports/sep06/

4. Sacchetto M, Huang Y, Bianco A, Rottondi C (2022) Using autoregressive models for real-time packet loss concealment in networked music performance applications. In: Proceedings of the 17th international audio mostly conference. AM '22, pp 203–210. Association for Computing Machinery, New York, USA. https://doi.org/10.1145/3561212.3561226
5. Sanneck H, Stenger A, Younes KB, Girod B (1996) A new technique for audio packet loss concealment. In: Proc. of GLOBECOM'96. 1996 IEEE Global telecommunications conference, pp 48–52
6. Guoqiang Z, Kleijn WB (2008) Autoregressive model-based speech packet-loss concealment. In: 2008 IEEE International conference on acoustics, speech and signal processing. international conference on acoustics speech and signal processing (ICASSP), pp 4797–4800. https://doi.org/10.1109/ICASSP.2008.4518730. QC20100830
7. Foster P, Klapuri A, Plumbley MD (2011) Causal prediction of continuous-valued music features. In: ISMIR, pp 501–506
8. Maezawa A (2019) Deep linear autoregressive model for interpretable prediction of expressive tempo. Proc SMC, 364–371
9. Chang C-J, Lee C-Y, Yang Y-H (2021) Variable-length music score infilling via XLNet and musically specialized positional encoding. arXiv:2108.05064
10. Vos K, Sørensen KV, Jensen SS, Valin J-M (2013) Voice coding with Opus. In: Audio Engineering Society Convention 135. Audio Engineering Society
11. Vos K (2013) A fast implementation of Burg's method. OPUS codec
12. Lazzaro J, Wawrzynek J (2001) A case for network musical performance. In: Proceedings of the 11th international workshop on network and operating systems support for digital audio and video. NOSSDAV '01, pp 157–166. Association for Computing Machinery, New York, USA. https://doi.org/10.1145/378344.378367
13. Virolainen J, Laine P (2005) Methods and apparatus for transmitting MIDI data over a lossy communications channel. Google Patents. US Patent 6,898,729
14. Nelson JR, Heidorn AJ, Cox RJ (2017) Reliable real-time transmission of musical sound control data over wireless networks. Google Patents. US Patent 9,601,097
15. Verma P, Mezza AI, Chafe C, Rottondi C (2020) A deep learning approach for low-latency packet loss concealment of audio signals in networked music performance applications. In: 2020 27th Conference of open innovations association (FRUCT), pp 268–275. IEEE
16. Briot J-P (2021) From artificial neural networks to deep learning for music generation: history, concepts and trends. Neural Comput & Applic 33(1):39–65
17. Akaike H (1969) Fitting autoregressive models for prediction. Ann Inst Stat Math 21(1):243–247. https://doi.org/10.1007/BF02532251
18. Levinson N (1946) The wiener (root mean square) error criterion in filter design and prediction. J Math Phys 25(1–4):261–278. https://doi.org/10.1002/sapm1946251261
19. Faber LJ (1986) Commentary on the denominator recursion for Burg's block algorithm. Proc IEEE 74(7):1046–1047. https://doi.org/10.1109/PROC.1986.13584
20. Ogita T, Rump S, Oishi S (2005) Accurate sum and dot product. SIAM J Sci Comput 26:1955–1988. https://doi.org/10.1137/030601818
21. Graillat S, Ménissier-Morain V (2007) Error-free transformations in real and complex floating point arithmetic. In: International symposium on nonlinear theory and its applications (NOLTA'07), pp 341–344
22. Sims CA (1980) Macroeconomics and reality. Econometrica J Econometric Society 1–48
23. Zivot, E., Wang, J.: Vector autoregressive models for multivariate time series. Modeling financial time series with S-PLUS®, 385–429 (2006)
24. Wasem OJ, Goodman DJ, Dvorak CA, Page HG (1988) The effect of waveform substitution on the quality of PCM packet communications. IEEE Trans Acoustics Speech Signal Process 36(3):342–348. https://doi.org/10.1109/29.1530
25. Fink M, Holters M, Zölzer U (2013) Comparison of various predictors for audio extrapolation. In: Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), pp 1–7
26. Pedersen CS, Zhou M, Møller MB, de Koeijer NEM, Østergaard J (2023) AR model for low latency packet loss concealment for wireless sound zones at low frequencies. In: Audio engineering society convention 154. Audio Engineering Society