Accelerating Automatic Differentiation of Direct Form Digital Filters

Chin-Yun Yu **György Fazekas**
Centre for Digital Music
Queen Mary University of London
Mile End Road, London E1 4NS, UK
{chin-yun.yu, george.fazekas}@qmul.ac.uk

Abstract

We introduce a general formulation for automatic differentiation through direct form filters, yielding a closed-form backpropagation that includes initial condition gradients. The result is a single expression that can represent both the filter and its gradients computation while supporting parallelism. C++/CUDA implementations in PyTorch achieve at least 1000x speedup over naive Python implementations and consistently run fastest on the GPU. For the low-order filters commonly used in practice, exact time-domain filtering with analytical gradients outperforms the frequency-domain method in terms of speed.

1 Introduction

Direct form (DF) digital filters with coefficients derived directly from their rational transfer functions are widely used in signal processing. With the advent of deep learning, there is increasing interest in integrating these filters into neural networks as an inductive bias and jointly optimising them using gradient-based methods. Differentiable implementations have been used for equaliser matching [1], filter design[2], virtual analogue modelling [3, 4], or bigger end-to-end systems like differentiable music tracks mixing [5, 6, 7]. All-pole filters, a special case of DF filters, are fundamental building blocks for voice modelling [8] and their differentiable version is used in neural vocoders and voice synthesis [9, 10, 11, 12, 13]. For a comprehensive review of differentiable filters, refer to Table 2 in [14]. Outside audio applications, differentiable DF filters have also been used in system identification [15] and state-space sequence models [16].

Efficiency is a significant issue when training filters in popular automatic differentiation (AD) frameworks such as PyTorch [17]. At the time of writing, PyTorch native operators only support filters with no recursions, such as convolution. Evaluating recursive DF filters using the available differentiable operators is slow due to the added up function call overheads in Python [4, 12, 13]. This is not the case for other AD frameworks that have recursion operators, such as `jax.lax.scan` in JAX [18] and `tf.scan` in TensorFlow [19]. To address this limitation, a common workaround is to approximate the filter using frequency sampling (FS) [14], thereby leveraging the fast Fourier transform (FFT) available in AD frameworks. Nevertheless, sampling creates time-domain aliasing [20] where its infinite impulse response (IIR) is folded. To reduce this mismatch error, one can increase the FFT resolution [21] or sample the transfer function outside the unit circle [22]. Parnichkun et al. [16] show that the tail of the IIR can be removed entirely in FS by subtracting it with a decayed and delayed version of itself. In addition, some works perform filtering directly using FS [4, 5, 23, 24, 25, 26], which results in circular convolution with the folded response, introducing more errors.

In this paper, we propose registering filters as low-level operators in PyTorch and deriving analytical gradients for backpropagation, thereby bypassing the overheads of AD frameworks. This approach has been used for specific filter types [7, 12, 13, 15], but we generalise it to all possible direct form

1st Workshop on Differentiable Systems and Scientific Machine Learning@ EurIPS 2025.

filters. This general form includes the transposed direct form (TDF), which has better numerical properties than DF and is the standard implementation in scientific computing libraries, thus better aligning with the needs of the open science community. It also allows exploration of acceleration techniques on parallel hardware, such as GPUs, which are not discussed in prior work. We stick with the time-domain implementation since filters used in practice are usually low-order, and the FS method is not necessarily faster in this case. We also show the analytical form of the gradients for the initial conditions, which can be beneficial when optimising on short sequences [27]. Our implementation is published in the open-source package `philtorch`.[1]

2 Background

An M^{th}-order time-invariant linear system has the following transfer function in the z-domain:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^{-M}} \tag{1}$$

where b_i, a_i are the feed-forward and feedback coefficients, respectively. To simplify the notation, we assume the numerator and denominator have the same order (padding if necessary). Directly converting the filtered signal $Y(z) = H(z)X(z)$ into difference equations results in the so-called *Direct-Form*[2] implementation:

$$v(n) = x(n) - a_1 v(n-1) - a_2 v(n-2) - \ldots - a_M v(n-M) \tag{2}$$
$$y(n) = b_0 v(n) + b_1 v(n-1) + \ldots + b_M v(n-M) \tag{3}$$

where $x(n)$ and $y(n)$ are the input and output signals, respectively. However, it is more common to see the *Transposed-Direct-Form* implementation in practice, such as in SciPy [28] and MATLAB, because the interleaved b_i coefficients provide compensating attenuation, making them more numerically robust [29]. To see how DF and TDF are related, let us utilise the following state-space filter:

$$\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{B}x(n) \tag{4}$$
$$y(n) = \mathbf{C}^\top \mathbf{v}(n) + Dx(n) \tag{5}$$

where $\mathbf{v}(n) \in \mathbb{R}^M$ is the state vector, $\mathbf{A} \in \mathbb{R}^{M \times M}$ is the transition matrix, $\mathbf{B}, \mathbf{C} \in \mathbb{R}^M$ are the input and output matrices, and $D \in \mathbb{R}$ is the direct path coefficient. The initial state $\mathbf{v}(0)$ gives an entry point to initiate the recursion. When $\mathbf{A}$ is the companion matrix of the polynomial in the denominator of Eq. (1), $\mathbf{C} = [b_1 - a_1 b_0, b_2 - a_2 b_0, \ldots, b_M - a_M b_0]^\top$, $\mathbf{B} = [1, 0, \ldots, 0]^\top$, and $D = b_0$, Eq. (4) and Eq. (5) are equivalent to Eq. (2) and Eq. (3). TDF traverses the signal flow of DF in reverse, which does not alter the transfer function [30]. Representing TDF in state-space form is simply replacing $\mathbf{A}$ with its transpose and swapping $\mathbf{B}$ and $\mathbf{C}$ [31].

Efficient AD through DF was developed separately by Forgione [15] and Yu [12, 13]. They treat Eq. (3) and Eq. (2) as separate filters and derive analytical gradients for the latter. This idea was later extended to parameter-varying all-pole filters and improved in [32, 33], where the backpropagation is further simplified. The latter version is implemented in TorchAudio [34]. However, in TDF, the feed-forward and feedback coefficients are interleaved in the difference equations [30], making it non-trivial to apply the same technique. In this work, we start from the state-space form, as it represents the coefficients in two matrices, making it easier to derive gradients.

3 Methodology

3.1 Backpropagation through state-space filters

Since we are implementing the filter without AD for optimal speed, we need to compute the filter's gradient analytically. We outline the results in this section. For detailed derivations, please refer to Appendix A. Given a scalar value $\mathcal{L} = f(y(0), y(1), \ldots)$ we wish to minimise, computed via a differentiable function f, we aim to calculate the gradients of the parameters we are interested in with respect to it. The parameters can be the filter variables or other parameters that the variables

[1] github.com/yoyolicoris/philtorch

[2] For simplicity, we assume type-II (transposed) direct forms throughout this paper.

depend on. In reverse-mode AD, the gradients with respect to the output $\frac{\partial \mathcal{L}}{\partial y(n)}$ are provided. Our task is to compute $\frac{\partial \mathcal{L}}{\partial x(n)}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{C}}$, and $\frac{\partial \mathcal{L}}{\partial D}$. The instantaneous gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{C}}$, $\frac{\partial \mathcal{L}}{\partial D}$ are trivial to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)} = \frac{\partial \mathcal{L}}{\partial y(n)}\mathbf{C}^\top, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{C}} = \sum_{n\geq 0} \frac{\partial \mathcal{L}}{\partial y(n)}\mathbf{v}(n)^\top, \quad \frac{\partial \mathcal{L}}{\partial D} = \sum_{n\geq 0} \frac{\partial \mathcal{L}}{\partial y(n)}x(n). \tag{6}$$

For $\frac{\partial \mathcal{L}}{\partial x(n)}$, let us denote $\mathbf{z}(n) = \mathbf{B}x(n)$ so Eq. (4) becomes $\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{z}(n)$. Given the instantaneous gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)}$, the gradients with respect to $\mathbf{z}(n)$ can be computed via *backpropagation-through-time* (BPTT) algorithm [35] and have recursive definition:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)} = \left(\mathbf{A}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n+1)}^\top + \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)}^\top\right)^\top = \left(\mathbf{A}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n+1)}^\top + \mathbf{C}\frac{\partial \mathcal{L}}{\partial y(n+1)}\right)^\top. \tag{7}$$

The rest of the gradients can be computed as:

$$\frac{\partial \mathcal{L}}{\partial x(n)} = \mathbf{B}\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}^\top + D\frac{\partial \mathcal{L}}{\partial y(n)}, \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}(-1)}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \sum_{n\geq 0} \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}x(n), \quad \frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \sum_{n\geq 0} \mathbf{v}(n)\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}. \tag{9}$$

Notice that Eq. (7) and Eq. (8) combined form a TDF filter as we have seen in Section 2, but running backwards in time since n is decreasing in Eq. (7). Similarly, if we derive the gradients for the TDF filter, we obtain a DF filter for backpropagation. We verified that our analytical gradients match the numerical gradients.

3.2 Implementation considerations

As we have mentioned in Section 1, recursions like Eq. (4) and Eq. (7) are better made as native operators in AD frameworks. Given the relation between TDF and DF, we should implement both and register the backpropagation of one using the other to save efforts. Alternatively, we can implement the state-space filter and register its backpropagation using itself with reparametrised arguments and reverse-time filtering. The state-space form also enables parallelisation across the time dimension, which is not possible with scalar difference equations.

The recursion Eq. (4) can be accelerated using the *associative scan* algorithm [36], which is implemented in JAX and TensorFlow but still work-in-progress for PyTorch [37]. The trick is to express the recursion as associative operations that can be computed in parallel. Let us use $\oplus$ to denote a binary operator that merges two tuples $(\mathbf{A}, \mathbf{z}) \oplus (\mathbf{A}', \mathbf{z}') \mapsto (\mathbf{A}'\mathbf{A}, \mathbf{A}'\mathbf{z} + \mathbf{z}')$. Then, we can express Eq. (4) as:

$$(\mathbf{0}, \mathbf{v}(n)) = (\mathbf{0}, \mathbf{v}(0)) \oplus (\mathbf{A}, \mathbf{z}(0)) \oplus (\mathbf{A}, \mathbf{z}(1)) \oplus \ldots \oplus (\mathbf{A}, \mathbf{z}(n-1)). \tag{10}$$

Using a parallelised associative scan, $\mathbf{v}(n)$ only needs $O(\frac{N}{p} + \log(p))$ time to be computed for all n instead of $O(N)$ time, where N is the sequence length and p is the number of parallel workers. However, the extra multiplications in $\oplus$ can introduce significant overhead when M is large. Yu et al. [7] proposed decomposing $\mathbf{A}$ into diagonal and invertible matrices, reducing matrix multiplications to element-wise multiplications. Similar ideas have been widely used to accelerate linear RNNs [38] and state-space sequence models such as Mamba [39], LRU [40], and S5 [41]. It is only applicable when $\mathbf{A}$ is diagonalisable (i.e., no repeated poles).

4 Experiments

To examine the proposed method, we benchmarked the recursion Eq. (4) with $M = 2$ in PyTorch.[3] We choose second-order because it is the minimum order for a real filter to have complex poles, and higher-order filters are often composed of many second-order sections to reduce numerical errors.

[3] Specifically, the recursion of $\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{z}(n)$ as AD through $\mathbf{B}\mathbf{x}(n)$ can be done via PyTorch.

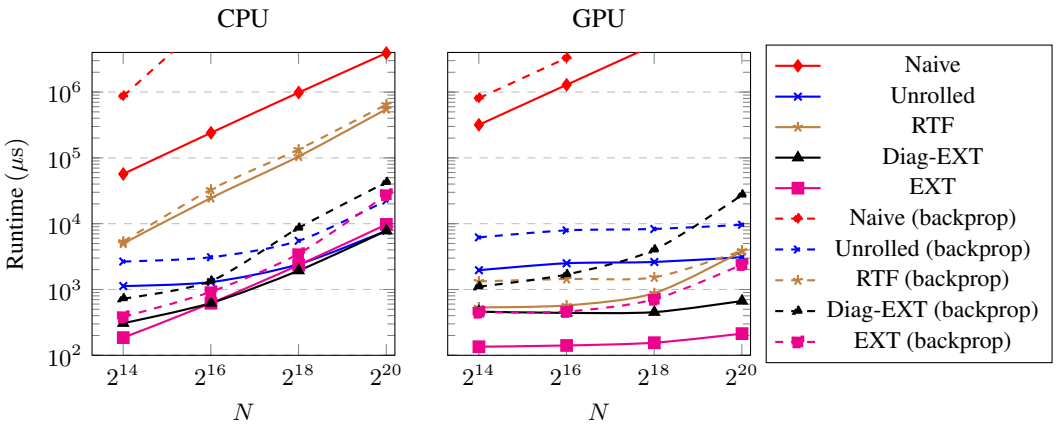

Figure 1: Runtime of the recursion Eq. (4) and its backpropagation with various signal lengths N and implementations.

The signal length N varies from 2^{14} to 2^{20}, corresponding to 1 to 60 seconds of audio at 16 kHz sample rate, which covers most of the range of audio processing tasks. We implement the recursion in C++ and CUDA as custom extensions (**EXT**) and register the backpropagation based on the equations in Section 3.1. In the C++ implementation, recursion is implemented sequentially, whereas for the CUDA kernel, we utilise the associative scan from the CUDA Core Compute Libraries.[4] The diagonalised version of EXT is denoted as **Diag-EXT**. The baselines we compare with are all implemented using PyTorch's Python API. The first one, **Naive**, is just a simple for-loop. The second baseline **Unrolled** implements the parallel scan using differentiable matrix multiplications [42]. The third one **RTF** evaluates the truncated impulse response and performs convolution in the frequency domain using FFT [16].

The benchmarks were run with PyTorch 2.8 and CUDA 12.9, using single precision, on a Ubuntu 25.04 machine equipped with an Intel i7-7700K CPU and an NVIDIA RTX 5060 Ti GPU. We limited the CPU benchmarks to a single thread to see the speed-up from having custom extensions without parallel acceleration. The results are shown in Figure 1. Our methods are the most efficient on both platforms, and EXT consistently spends the least time computing gradients in most configurations. The naive implementation is remarkably slower than others and quickly becomes impractical when N increases. On the CPU, the unrolled method has higher overhead for smaller N but becomes comparable to the other two methods for large N. The RTF method is only slightly faster than Naive. On the GPU, RTF becomes comparable to Diag-EXT in smaller values of N but grows faster as N increases and exceeds Unrolled at $N = 2^{20}$. The gap between Diag-EXT and EXT on the GPU is likely due to the extra eigen-space projection in Diag-EXT, and we expect the gap to disappear as M increases.

5 Conclusion

We presented the general form of automatic differentiation via (transposed) direct form filters in the state-space domain. The derivations yield a closed-form backward pass that is itself a state-space (DF/TDF) recursion, and include gradients with respect to initial conditions. This view enables a single low-level operator whose backpropagation is implemented by the same kernel. Furthermore, the state-space form naturally supports parallelisation. Our custom PyTorch extensions that implement the proposed recursion substantially outperform both the naive and frequency-sampling baselines. These results reinforce the conclusion that exact time-domain filtering with analytical gradients is preferable for low-order filters. Additionally, we notice that the presented derivation can be easily extended to parameter-varying cases and applies to a broader class of linear state-space models. Moreover, a forward-mode differentiation can be derived similarly, which is helpful for second-order optimisation methods. How the closed-form solution affects the numerical accuracy of gradients is also an interesting direction to explore. We left these extensions to future work.

[4] nvidia.github.io/cccl/

Acknowledgments and Disclosure of Funding

Chin-Yun Yu is a research student at the UKRI Centre for Doctoral Training in Artificial Intelligence and Music, supported jointly by UK Research and Innovation (grant number EP/S022694/1) and Queen Mary University of London. György Fazekas's research on Knowledge-driven Deep Learning for Music Informatics was supported by the Leverhulme Trust and the Royal Academy of Engineering under the RAEng / Leverhulme Trust Research Fellowships scheme.

References

[1] Shahan Nercessian. Neural parametric equalizer matching using differentiable biquads. In *International Conference on Digital Audio Effects*, pages 265–272, Vienna, Austria, 2020.

[2] Joseph T Colonel, Christian J Steinmetz, Marcus Michelen, and Joshua D Reiss. Direct design of biquad filter cascades with deep learning by sampling random polynomials. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3104–3108, Singapore, 2022. IEEE.

[3] Boris Kuznetsov, Julian D Parker, and Fabián Esqueda. Differentiable IIR filters for machine learning applications. In *International Conference on Digital Audio Effects*, pages 297–303, Vienna, Austria, 2020.

[4] Shahan Nercessian, Andy Sarroff, and Kurt James Werner. Lightweight and interpretable neural modeling of an audio distortion effect using hyperconditioned differentiable biquads. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 890–894, Toronto, Canada, 2021. IEEE.

[5] Christian J Steinmetz, Nicholas J Bryan, and Joshua D Reiss. Style transfer of audio effects with differentiable signal processing. *Journal of the Audio Engineering Society*, 70(9):708–721, 2022.

[6] S. Lee, M. A. Martínez-Ramírez, W.-H. Liao, S. Uhlich, G. Fabbro, K. Lee, and Y. Mitsufuji. Reverse engineering of music mixing graphs with differentiable processors and iterative pruning. *JAES*, 73:344–365, June 2025.

[7] Chin-Yun Yu, Marco A. Martínez-Ramírez, Junghyun Koo, Ben Hayes, Wei-Hsiang Liao, György Fazekas, and Yuki Mitsufuji. DiffVox: A differentiable model for capturing and analysing vocal effects distributions. In *Proc. DAFx*, pages 334–341, 2025.

[8] John D. Markel and Augustine H. Gray. *Linear Prediction of Speech*, volume 12 of *Communication and Cybernetics*. Springer, Berlin, Heidelberg, 1976.

[9] Jean-Marc Valin and Jan Skoglund. LPCNet: Improving neural speech synthesis through linear prediction. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5891–5895, Brighton, UK, 2019. IEEE.

[10] Krishna Subramani, Jean-Marc Valin, Umut Isik, Paris Smaragdis, and Arvindh Krishnaswamy. End-to-end LPCNet: A neural vocoder with fully-differentiable LPC estimation. In *Interspeech 2022*, pages 818–822, Incheon, Korea, 2022.

[11] Kilian Schulze-Forster, Gaël Richard, Liam Kelley, Clement SJ Doire, and Roland Badeau. Unsupervised music source separation using differentiable parametric source models. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31:1276–1289, 2023.

[12] Chin-Yun Yu and György Fazekas. Singing Voice Synthesis Using Differentiable LPC and Glottal-Flow-Inspired Wavetables. In *Proceedings of the 24th International Society for Music Information Retrieval Conference*, pages 667–675, Milan, Italy, November 2023. ISMIR.

[13] Chin-Yun Yu and György Fazekas. GOLF: A Singing Voice Synthesiser with Glottal Flow Wavetables and LPC Filters. *TISMIR*, Dec 2024.

[14] Ben Hayes, Jordie Shier, György Fazekas, Andrew McPherson, and Charalampos Saitis. A review of differentiable digital signal processing for music and speech synthesis. *Frontiers in Signal Processing*, 3:1284100, 2024.

[15] Marco Forgione and Dario Piga. dynoNet: A neural network architecture for learning dynamical systems. *International Journal of Adaptive Control and Signal Processing*, 35(4):612–626, 2021.

[16] Rom Parnichkun, Stefano Massaroli, Alessandro Moro, Jimmy T.H. Smith, Ramin Hasani, Mathias Lechner, Qi An, Christopher Re, Hajime Asama, Stefano Ermon, Taiji Suzuki, Michael Poli, and Atsushi Yamashita. State-free inference of state-space models: The transfer function approach. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 39834–39860. PMLR, 21–27 Jul 2024.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, pages 8026 – 8037, 2019.

[18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax, 2018.

[19] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[20] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT)*. http://ccrma.stanford.edu/~jos/mdft/, accessed (2024-10-25). online book.

[21] Sungho Lee, Hyeong-Seok Choi, and Kyogu Lee. Differentiable artificial reverberation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:2541–2556, 2022.

[22] Gloria Dal Santo, Gian Marco De Bortoli, Karolina Prawda, Sebastian J Schlecht, and Vesa Välimäki. FLAMO: An open-source library for frequency-domain differentiable audio processing. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2025.

[23] Lauri Juvela, Bajibabu Bollepalli, Junichi Yamagishi, and Paavo Alku. GELP: GAN-excited liner prediction for speech synthesis from mel-spectrogram. In *Proc. INTERSPEECH*, pages 694–698, 2019.

[24] Suhyeon Oh, Hyungseob Lim, Kyungguen Byun, Min-Jae Hwang, Eunwoo Song, and Hong-Goo Kang. ExcitGlow: Improving a WaveGlow-based neural vocoder with linear prediction analysis. In *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 831–836, Auckland, New Zealand, 2020. IEEE.

[25] Alec Wright and Vesa Valimaki. Grey-box modelling of dynamic range compression. In *International Conference on Digital Audio Effects*, pages 304–311, Vienna, Austria, 2022.

[26] Alistair Carson, Simon King, Cassia Valentini Botinhao, and Stefan Bilbao. Differentiable grey-box modelling of phaser effects using frame-based spectral processing. In *DAFx*, pages 219–226, 2023.

[27] Marco Forgione and Dario Piga. Continuous-time system identification with neural networks: Model structures and fitting criteria. *European Journal of Control*, 59:69–81, 2021.

[28] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[29] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*, chapter Numerical Robustness of TDF-II. https://ccrma.stanford.edu/~jos/fp/Numerical_Robustness_TDF_II.html, accessed (2025-10-07). online book.

[30] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*, chapter Transposed Direct-Forms. https://ccrma.stanford.edu/~jos/fp/Transposed_Direct_Forms.html, accessed (2025-10-07). online book.

[31] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*, chapter Transposition of a State Space Filter. https://ccrma.stanford.edu/~jos/fp/Transposition_State_Space_Filter.html, accessed (2025-10-07). online book.

[32] Chin-Yun Yu and György Fazekas. Differentiable time-varying linear prediction in the context of end-to-end analysis-by-synthesis. In *Interspeech 2024*, pages 1820–1824, Kos, Greece, 2024.

[33] Chin-Yun Yu, Christopher Mitcheltree, Alistair Carson, Stefan Bilbao, Joshua D. Reiss, and György Fazekas. Differentiable all-pole filters for time-varying audio systems. In *DAFx*, pages 345–352, 2024.

[34] Jeff Hwang, Moto Hira, Caroline Chen, Xiaohui Zhang, Zhaoheng Ni, Guangzhi Sun, Pingchuan Ma, Ruizhe Huang, Vineel Pratap, Yuekai Zhang, Anurag Kumar, Chin-Yun Yu, Chuang Zhu, Chunxi Liu, Jacob Kahn, Mirco Ravanelli, Peng Sun, Shinji Watanabe, Yangyang Shi, and Yumeng Tao. TorchAudio 2.1: Advancing speech recognition, self-supervised learning, and audio processing components for PyTorch. In *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 1–9, Taipei, Taiwan, 2023.

[35] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[36] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[37] Yidi Wu, Thomas Ortner, Richard Zou, Edward Z. Yang, Adnan Akhundov, Horace He, and Yanan Cao. Control Flow Operators in PyTorch. In *Championing Open-source DEvelopment in ML Workshop @ ICML25*, 2025.

[38] Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. In *ICLR*, Vancouver, Canada, 2018.

[39] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

[40] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[41] Jimmy T.H. Smith, Andrew Warrington, and Scott Linderman. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations*, 2023.

[42] Chin-Yun Yu. Block-based Fast Differentiable IIR in PyTorch. https://iamycy.github.io/posts/2025/06/28/unroll-ssm/, June 2025. Accessed: 2025-10-08.

A Derivation details

A.1 $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}$

This section provides the derivation details for Eq. (7) where $\mathbf{z}(n) = \mathbf{B}x(n)$. Let us unroll Eq. (4) so only $\mathbf{v}(0)$ and $\mathbf{z}(n)$ are on the right-hand side for any non-negative integer n:

$$\mathbf{v}(n+1) = \mathbf{A}^{n+1}\mathbf{v}(0) + \sum_{m=0}^{n} \mathbf{A}^{n-m}\mathbf{z}(m). \tag{11}$$

Let $\mathbf{v}(0) = \sum_{m=-\infty}^{-1} \mathbf{A}^{-m-1}\mathbf{z}(m)$, which is always valid, as a solution of $\mathbf{z}(m)$ for $m < 0$ always exists that produces the same $\mathbf{v}(0)$ with any $\mathbf{A}$. Then, we can eliminate $\mathbf{v}(0)$ so Eq. (11) is simplified as:

$$\mathbf{v}(n) = \mathbf{z}(n-1) + \sum_{m=-\infty}^{n-2} \mathbf{A}^{n-m-1}\mathbf{z}(m). \tag{12}$$

This form does not need to account for boundary conditions, which makes the following derivation cleaner. We can interpret Eq. (12) as the response of the system to an infinite-length input sequence.

Then, it is easy to see that the Jacobian of $\mathbf{v}(n)$ with respect to $\mathbf{z}(n)$ is:

$$\frac{\partial \mathbf{v}(n)}{\partial \mathbf{z}(m)} = \begin{cases} \mathbf{A}^{n-m-1} & m < n-1 \\ \mathbf{I} & m = n-1 \\ \mathbf{0} & m \geq n \end{cases}, \tag{13}$$

Given the instantaneous gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)}$ computed in Eq. (6), we can compute $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}$ using the chain rule:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)} &= \sum_{m=-\infty}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)} \frac{\partial \mathbf{v}(m)}{\partial \mathbf{z}(n)} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)} + \sum_{m=n+2}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)} \mathbf{A}^{m-n-1} \\ &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)}^{\top} + \sum_{m=n+2}^{\infty} \left(\mathbf{A}^{\top}\right)^{m-n-1} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)}^{\top} \right)^{\top} \\ &= \left(\mathbf{A}^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n+1)}^{\top} + \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)}^{\top} \right)^{\top}. \end{aligned} \tag{14}$$

The last step is obtained by noticing that the term in the parentheses can be converted from Eq. (12), with $\mathbf{A}$ replaced by $\mathbf{A}^{\top}$, $\mathbf{z}(m)$ replaced by $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)}^{\top}$, and m replaced by $2n - m$. The $m \mapsto 2n - m$ reparametrisation implies reverse-time filtering. In practice, we cannot evaluate infinite sequences but up to a finite length N (from $y(0)$ to $y(N-1)$). Thus, $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}$ is zeros for $n \geq N-1$ since $\mathbf{z}(n)$ are not evaluated beyond $n = N-2$. We can set $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(N-1)} = \mathbf{0}$ to initiate the evaluation of Eq. (14) from $n = N-1$ to $n = 0$.

A.2 $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$

To derive $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$, let us utilise the property of the Kronecker product that $\mathbf{A}\mathbf{v}(n) = \text{vec}(\mathbf{A}\mathbf{v}(n)) = (\mathbf{v}(n)^{\top} \otimes \mathbf{I})\text{vec}(\mathbf{A})$, where $\text{vec}(\cdot)$ denotes the vectorisation operator that stacks the columns of a matrix into a vector. The Jacobian of $\mathbf{v}(n)$ with respect to $\text{vec}(\mathbf{A})$ can be expressed as:

$$\begin{aligned} \frac{\partial \mathbf{v}(n)}{\partial \text{vec}(\mathbf{A})} &= \frac{\partial \mathbf{v}(n)}{\partial \mathbf{v}(n-1)} \frac{\partial \mathbf{v}(n-1)}{\partial \text{vec}(\mathbf{A})} + \mathbf{v}(n-1)^{\top} \otimes \mathbf{I} \\ &= \mathbf{A} \frac{\partial \mathbf{v}(n-1)}{\partial \text{vec}(\mathbf{A})} + \mathbf{v}(n-1)^{\top} \otimes \mathbf{I} \\ &= \sum_{m=0}^{n-1} \mathbf{A}^{n-m-1} \left(\mathbf{v}(m)^{\top} \otimes \mathbf{I} \right). \end{aligned} \tag{15}$$

Then, applying the chain rule, we have:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \text{vec}(\mathbf{A})} &= \sum_{m=1}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)} \frac{\partial \mathbf{v}(m)}{\partial \text{vec}(\mathbf{A})} \\
&= \sum_{m=1}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)} \sum_{n=0}^{m-1} \mathbf{A}^{m-n-1} \left(\mathbf{v}(n)^\top \otimes \mathbf{I} \right) \\
&= \left(\sum_{m=1}^{\infty} \left(\sum_{n=0}^{m-1} \left(\mathbf{v}(n) \otimes \mathbf{I} \right) \left(\mathbf{A}^\top \right)^{m-n-1} \right) \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)}^\top \right)^\top \\
&= \left(\sum_{n=0}^{\infty} \left(\mathbf{v}(n) \otimes \mathbf{I} \right) \sum_{m=n+1}^{\infty} \left(\mathbf{A}^\top \right)^{m-n-1} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)}^\top \right)^\top \\
&= \left(\sum_{n=0}^{\infty} \left(\mathbf{v}(n) \otimes \mathbf{I} \right) \left(\frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)}^\top + \sum_{m=n+2}^{\infty} \left(\mathbf{A}^\top \right)^{m-n-1} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(m)}^\top \right) \right)^\top \\
&= \left(\sum_{n=0}^{\infty} \left(\mathbf{v}(n) \otimes \mathbf{I} \right) \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}^\top \right)^\top \\
&= \left(\sum_{n=0}^{\infty} \text{vec} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}^\top \mathbf{v}(n)^\top \right) \right)^\top \\
&= \text{vec} \left(\sum_{n=0}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{z}(n)}^\top \mathbf{v}(n)^\top \right)^\top .
\end{aligned}
\tag{16}
$$

The conversion from the fifth to the sixth step is based on Eq. (14). The second last step utilises the same property we mentioned before that $\text{vec}(\mathbf{UW}) = (\mathbf{W}^\top \otimes \mathbf{I})\text{vec}(\mathbf{U})$. After removing the vectorisation operator on both sides, we obtain $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ shown in Eq. (9).

A.3 $\frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)}$

From Eq. (11), it is easy to see that the Jacobian of $\mathbf{v}(n)$ with respect to $\mathbf{v}(0)$ is:

$$
\frac{\partial \mathbf{v}(n)}{\partial \mathbf{v}(0)} = \mathbf{A}^n. \tag{17}
$$

Apply the chain rule, we have:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)} &= \sum_{n=1}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)} \frac{\partial \mathbf{v}(n)}{\partial \mathbf{v}(0)} + \frac{\partial \mathcal{L}}{\partial y(0)} \frac{\partial y(0)}{\partial \mathbf{v}(0)} \\
&= \sum_{n=1}^{\infty} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)} \mathbf{A}^n + \frac{\partial \mathcal{L}}{\partial y(0)} \mathbf{C}^\top \\
&= \left(\sum_{n=1}^{\infty} \left(\mathbf{A}^\top \right)^n \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)}^\top + \mathbf{C} \frac{\partial \mathcal{L}}{\partial y(0)} \right)^\top \\
&= \left(\mathbf{A}^\top \left(\frac{\partial \mathcal{L}}{\partial \mathbf{v}(1)}^\top + \sum_{n=2}^{\infty} \left(\mathbf{A}^\top \right)^{n-1} \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n)}^\top \right) + \mathbf{C} \frac{\partial \mathcal{L}}{\partial y(0)} \right)^\top \\
&= \left(\mathbf{A}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}(0)}^\top + \mathbf{C} \frac{\partial \mathcal{L}}{\partial y(0)} \right)^\top \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{z}(-1)}.
\end{aligned}
\tag{18}
$$

The last three steps are based on Eq. (14) and Eq. (6).

B Example code

We provide a minimal Python code below to demonstrate how to implement the recursion $\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{z}(n)$ as a differentiable operator in PyTorch. The forward call is just for reference and not optimised.

```
import torch
from torch import Tensor
from typing import Tuple
from torch.autograd import Function

class LTIMatrixRecurrence(Function):
    @staticmethod
    def forward(A: Tensor, v0: Tensor, z: Tensor) -> Tensor:
        # A: (M, M)
        # v0: (M,)
        # z(0:N-1): (N, M)
        # returns v(1:N): (N, M)
        # Ideally, implement the forward pass using C++/CUDA extension
    for efficiency.
        # Here is an inefficient reference implementation in Python.
        v = z.clone()
        v_n = v0
        for n in range(z.shape[0]):
            v[n, :] += A @ v_n
            v_n = v[n, :]
        return v

    @staticmethod
    def setup_context(ctx, inputs, output):
        A, v0, _ = inputs
        v = output
        ctx.save_for_backward(A, v0, v)

    @staticmethod
    def backward(
        ctx, grad_v: Tensor
    ) -> Tuple[Tensor, Tensor, Tensor]:
        A, v0, v = ctx.saved_tensors

        flipped_grad_z = LTIMatrixRecurrence.apply(
            A.T, torch.zeros_like(v0), grad_v.flip(0)
        )

        grad_z = flipped_grad_z.flip(0)
        grad_v0 = A.T @ grad_z[0, :]
        padded_v = torch.cat([v0.unsqueeze(0), v[:-1]], dim=0)
        grad_A = grad_z.T @ padded_v
        return grad_A, grad_v0, grad_z
```