A fast version of the ADSTFT

Project in FMSN35, Stationary and Non-stationary Spectral Analysis

Kasper Nordenram

May 15, 2025

The differentiable adaptive short-time Fourier transform (DASTFT or ADSTFT) was introduced by Leiber et al. [1], and allows for different window lengths for each point in the time-frequency plane, w.r.t. which the resulting spectrogram can be differentiated. The method presented in this work, referred to as the fastDSTFT, produces a similar result, but instead of employing a direct computation using different window lengths, it is based on the idea of computing discrete Fourier transforms (DFTs) using short B-spline windows, which can be combined to approximate longer windows. The spline windows can be added in a unique way for each point in the time-frequency plane to approximate different time-domain windows, and due to the linearity of the DFT, this addition directly carries over to the frequency domain. The goal of first computing the short basis function windows is to make use of the Fast Fourier Transform (FFT), instead of being forced to compute a naive DFT with specific windows for each time-frequency atom. This makes the computation faster (by over 500 times for the best cases of both methods) and more memory efficient (allowing for larger batches). The method also provides well-defined derivatives of parameters of the larger time domain window function in all cases where this function itself has well-defined derivatives of the parameters, just like the original method. This allows for optimization of window lengths using gradient-based methods, which is useful, for example, when the method is used as a layer in a neural network.

1 Method

The proposed method will now be described in more detail. For simplicity, we start by looking at a single time point, around which a symmetric segment of the signal is extracted and treated. Although Beta windows have been most prevalently used in the implementation, the explanation here will focus on the simpler Hann windows for clarity. We have a basis of B-splines

$$\mathcal{B} = \{B_i\}_i, \tag{1}$$

each normalized to sum to 1 in discrete time. The splines can be of any degree, but in this report it is always 3. Note that we only use the interior basis splines, which are all identical up to a translation. Each basis spline is defined over the entire discrete real line, being 0 outside of its support. The basis is used to estimate a Hann window with a parameter λ_{tk} at time t and frequency k as

$$W(x, \lambda_{tk}) = \sum_i \alpha_i(\lambda_k) B_i(x), \tag{2}$$

for some coefficients α_i that vary with λ_{tk}. The Greville abscissae are the points

$$\tilde{x}_i = \frac{1}{p} \sum_{j=1}^{p} t_{i+j}, \quad \text{for } i = 0, 1, \ldots, n-1, \tag{3}$$

where p is the degree of the spline and t_i is knot i—for internal splines, essentially the center of each spline. A fast and simple way to approximate a function using splines is to set the coefficient of spline i to $f(\tilde{x}_i)$, which works quite well as the splines form a partition of unity. Thus, we can easily write down closed-form expressions for these as a function of λ, making the differentiation straight forward. The window estimate is exactly

$$W(x, \lambda_{tk}) = \sum_i f(\tilde{x}_i, \lambda_{tk}) B_i(x). \tag{4}$$

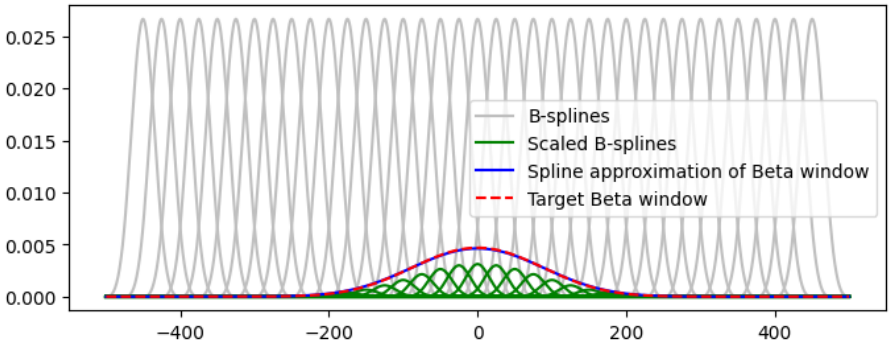

Figure 1: The spline windows used for time t, here centered at 0, and their approximation of a Beta window. Note that both the individual splines and their coefficients have been normalized to sum to 1, meaning that the window they combine to form also sums to 1.

The spline approximation is shown for a Beta window in Figure 1. Applying the window to a signal y, we get

$$y_W(x, \lambda_{tk}) = y(x) W(x, \lambda_{tk}) = \sum_i f(\tilde{x}_i, \lambda_{tk}) B_i(x) y(x) = \sum_i f(\tilde{x}_i, \lambda_{tk}) y_{B_i}(x). \tag{5}$$

which is illustrated with Beta windows in 2. We now define $\hat{s}_W$ as the discrete Fourier transform of the signal windowed by the Hann approximation, and $\hat{s}_{B_i}$ as the Fourier transform of the signal windowed by basis spline B_i, and see that due to the linearity of the transform, we have

$$\hat{s}_W(k, \lambda_{tk}) = \mathcal{F}_k\{y_W(x, \lambda_{tk})\} = \mathcal{F}_k \left\{ \sum_i f(\tilde{x}_i, \lambda_{tk}) y_{B_i}(x) \right\} = \sum_i f(\tilde{x}_i, \lambda_{tk}) \mathcal{F}_k\{y_{B_i}(x)\} = \sum_i f(\tilde{x}_i, \lambda_{tk}) \hat{s}_{B_i}(k), \tag{6}$$

This is exemplified in Figure 3. Now, using the chain rule, we want to know how the loss function is affected by changes in λ_{tk}. To do this, we seek to find $\frac{d\hat{s}_W(k, \lambda_{tk})}{d\lambda_{tk}}$. Clearly, this is just found as

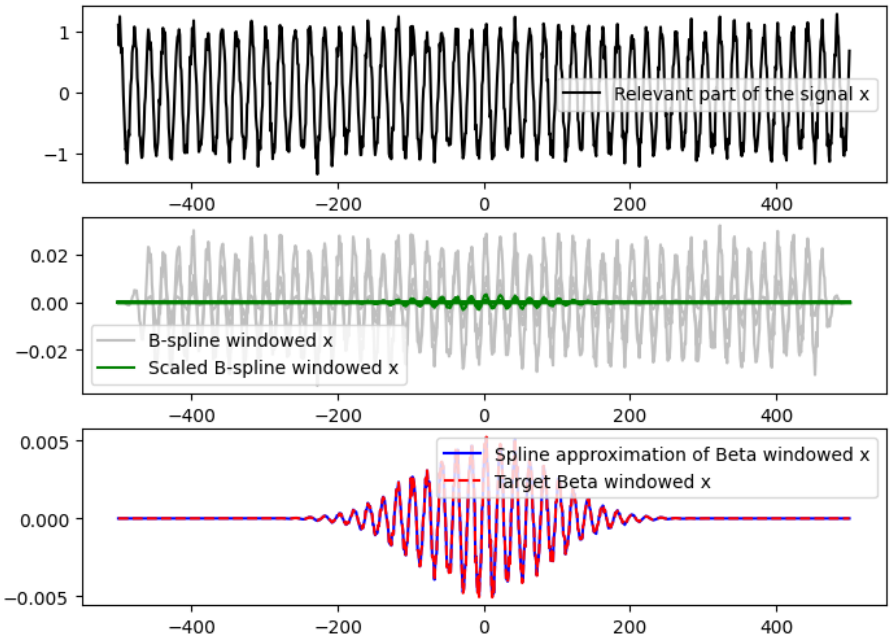

Figure 2: The windows in Figure 1, applied to a signal chunk.

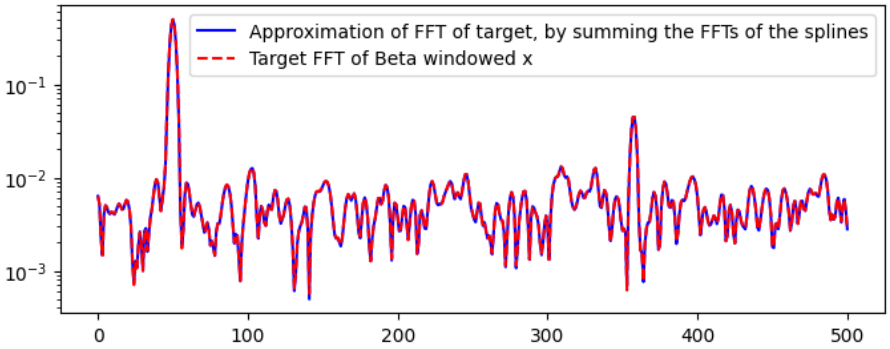

Figure 3: Approximation of the DFT for a Beta windowed segment of x centered at time t. Absolute values of the complex values are shown. The approximation is done by summing the spline windowed DFTs using the coefficients from the spline approximation of the Beta window. This works due to the linearity of the DFT.

$$\frac{d\hat{s}_W(k,\lambda_{tk})}{d\lambda_{tk}} = \sum_i \frac{\partial f(\tilde{x}_i,\lambda_{tk})}{\partial \lambda_{tk}} \hat{s}_{B_i}(k), \tag{7}$$

so to find the derivatives analytically, we will just need a window to estimate that can be easily and continuously differentiated w.r.t. λ. If we use integral normalized Hann windows,

$$H(x,\lambda_{tk}) = \begin{cases} \frac{2}{\lambda_{tk}} \cos^2\left(\frac{\pi x}{\lambda_{tk}}\right), & x \in [-\lambda_{tk}/2, \lambda_{tk}/2] \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

this derivative is simply (calling λ_{tk} just λ for legibility)

$$\frac{\partial H(x,\lambda)}{\partial \lambda} = \frac{4}{\lambda}\cos\left(\frac{\pi x}{\lambda}\right)\sin\left(\frac{\pi x}{\lambda}\right)\frac{\pi x}{\lambda^2} - \frac{2}{\lambda^2}\cos^2\left(\frac{\pi x}{\lambda}\right) = \frac{2\pi x}{\lambda^3}\sin\left(\frac{2\pi x}{\lambda}\right) - \frac{2}{\lambda^2}\cos^2\left(\frac{\pi x}{\lambda}\right) \tag{9}$$

within $[-\lambda/2, \lambda/2]$ and 0 otherwise. It should be noted that with automatic differentiation, it is only necessary that the derivative is well defined, not that it can be found analytically, so using a simple function like the Hann window is not necessarily required. It is also worth noting that any continuous function can be approximated arbitrarily well using splines with coefficients defined by evaluation at the Greville abscissae, if we use closely enough spaced knots [2]. This means that we can improve our window function approximation to be as good as we need by increasing the number of basis splines on our interval. The order of convergence of the Greville-based approximation is the knot spacing squared, if the function being approximated has at least two continuous derivatives. This is generally not as good as it could be with any choice of coefficients, however, this method has other advantages. It is known as Schoenberg's *variation diminishing* spline approximation, as it crosses any straight line at most as many times as the function it approximates. This smoothness translates to the frequency domain, where it has low levels of artefacts compared to numerically optimized splines, which can sometimes start oscillating.

2 Algorithm and computational considerations

2.1 `spline_stft`

In the previous section, we showed how we can extract a signal segment, window it with each of the splines, and then apply the DFT to the windowed signal. This is not exactly how it is done in the implementation of the fastDSTFT. Instead, we assert that neighboring splines are translations by a number of samples that divides the STFT stride evenly—this is referred to as the spline density, where 1 means that these translations are the same as the stride. By doing this, we can reuse the splines centered around one time point for some time points before and after, saving computation time. All spline-windowed DFTs are computed as the first step in the fastDSTFT algorithm. Segments corresponding to the support of each spline are extracted and zero-padded, and then the DFT is applied. To encode the temporal position of each window, a complex modulation with the start of the signal as phase reference is applied to the result of each spline-windowed DFT. The result is called a `spline_stft`, and has dimension (B, F, S)—batch size, number of frequencies, and number of spline windows.

2.2 `expanded_spline_stfts`

We then build a `PyTorch` tensor that assigns each spline to all the time steps it belongs to, creating `expanded_spline_stfts`. Since each spline can belong to several time steps, there is a lot of repetition in this tensor, which has size (B, F, s, T), where s is the number of splines per time step, and T is the number of time steps. To avoid the memory implications of realizing this large matrix, the `view_as_strided` command is used, which keeps the original memory configuration but modifies the internal view of the tensor, so it can be used as if it were expanded.

2.3 `coeffs`

To calculate the spline coefficients, we first find the Greville abscissae of the relevant splines for a time point relative to this time point. These relative positions will be the same for each time point and its associated splines. These are mapped from the range ± half of the maximum window length, to the range [0, 1], the support of the Beta distribution. The unnormalized Beta distributions is then evaluated at these points, for each of the window lengths in the time frequency plane. Each set of coefficients is then normalized to sum to one. Since very small numbers tend to arise in this computation, it is done in the logarithm domain to avoid numerical issues. The result has size $(1, F, s, T)$. This function contains a number of small steps, so it is compiled using the `@torch.compile` decorator, fusing operations resulting in a shorter computation time.

2.4 Final computation of the adaptive STFT

We finally combine `expanded_spline_stfts` with `coeffs` by broadcasting `coeffs` over all the batches and multiplying element-wise, before summing over the s splines for each batch, frequency, and time point. If done naively, this computation realizes the `expanded_spline_stfts`, and stores an intermediate result representing the element-wise product of the tensors. Since we want to avoid realizing these large tensors, this computation is placed inside a function that is compiled using the `@torch.compile` decorator. This compilation optimizes the computation, so that the inputs are mapped directly to the output, which has size (B, F, T)—a STFT for each sample in the batch.

2.5 Remarks

In the forward pass, the absolute values of the complex adaptive STFT entries are calculated, and the machine epsilon is added to avoid problems in a possible subsequent logarithm transform. Throughout all the steps of the algorithm, element-wise computations on `PyTorch` tensors are used in favor of `Python` level loops, to utilize low-level optimization and parallelization present in the `PyTorch` library. When calculating on a CPU at batch size 64, the computation of the unmodulated `spline_stft` contributes roughly 63% of the computation time (of which 96% is calculating the FFTs), modulating it contributes 6%, expanding it less than 1%, calculating the spline coefficients 4%, and calculating the final STFT contributes 33%. On the GPU, the first and final computations swap contributions.

3 Convergence

In Figure 4, the convergence of the optimization problem minimizing the Shannon entropy with regularization as formulated in Leiber et al. [1] is studied. The optimization algorithm used is Adam. The ADSTFT here is modified to use a Beta window instead of Hann, and in the way it enforces minimum and maximum window lengths, to be comparable to the fastDSTFT. The original implementation applied a filter that clamped the window lengths to the allowed range in the forward pass, without modifying the window lengths parameter. This cuts the gradient flow of window lengths outside of the allowed range to the output, thus stopping these lengths from being further updated. The modified implementation puts the window lengths outside of the allowed range back inside

after taking a step with the optimizer, enforcing the allowed range while still allowing for further learning. Both these changes result in a lower final loss. As can be seen, the convergences of the fastDSTFT and ADSTFT follow very similar curves. When the minimum window length is set to the same value for the methods, the ADSTFT achieves a slightly lower loss. This is due to the Greville evaluations producing a window approximation that is a bit too long at low lengths, so the same assigned minimum length is somewhat more restrictive for the fastDSTFT. Allowing for a slightly lower window length gives a very similar outcome. In conclusion, we can see that the fastDSTFT has very similar convergence properties to the ADSTFT, and as such is not hindered in gradient-based optimization by the use of spline approximation.

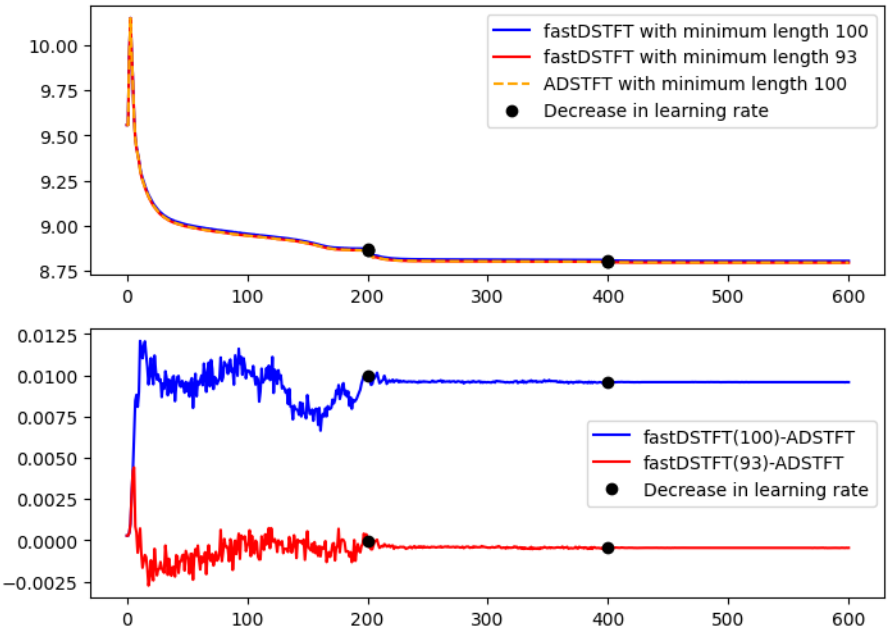

Figure 4: Comparison of the convergence of the entropy optimization problem for the fastDSTFT and the DSTFT. The spline density used here is 5, to closely emulate the original method. The learning rate is 100, 10, and 1 in the three separate sections.

4 Time complexity

The time complexity for calculating the spectrogram in the original method is

$$\mathcal{O}(TN^2), \tag{10}$$

with T being the number of time steps and N being the support, as it computes a naive DFT for each time step. With the proposed method, it consists of two parts—the initial FFT for S spline basis functions, and the combination of the windows, where TFs points are evaluated, s being the number of splines per time point, and F being the number of frequencies evaluated,

$$\mathcal{O}(SN \log_2 N + TFs). \tag{11}$$

S is approximately Td, where d is the spline density, and $F \propto N$, so we can write

$$\mathcal{O}(TdN \log_2 N + TNs). \tag{12}$$

Assuming that the constant factors in Eq. 10 and for both terms 12 are all the same, and the parameters are $N = 1000$, $d = 2$, $s = 35$, we get a reduction in computation time by a factor of around 18. This is much less than seen in practice, indicating that the constant factors of the proposed method are lower than for the original method.

5 Timing

In Figure 5, the proposed method and the original ADSTFT are timed on a MacBook Air M1 (2020) with 8GB of RAM. The time is averaged over 100 epochs for the proposed method, and 10 epochs for the original method. The best time over 3 tries is presented, as background processes on the computer can sometimes cause large slowdowns. To get a realistic timing of the backward pass, as if the methods were used as a layer in a neural network, a minimal loss function consisting of summing the spectrogram is used. Using a complicated loss function would incur overhead in the backward pass when the gradients are tracked through this function, which is unrelated to the spectrogram computation. For the fastDSTFT, a spline density of 2 is used. It approximates Beta windows, while the ADSTFT uses Hann windows. Note that the unstable speed on GPU (see for example batch sizes $2^7 - 1$, 2^7, $2^7 + 1$) might be due to `torch.compile` for Metal (the API for Apple GPUs) being an early prototype as of the date of this report.

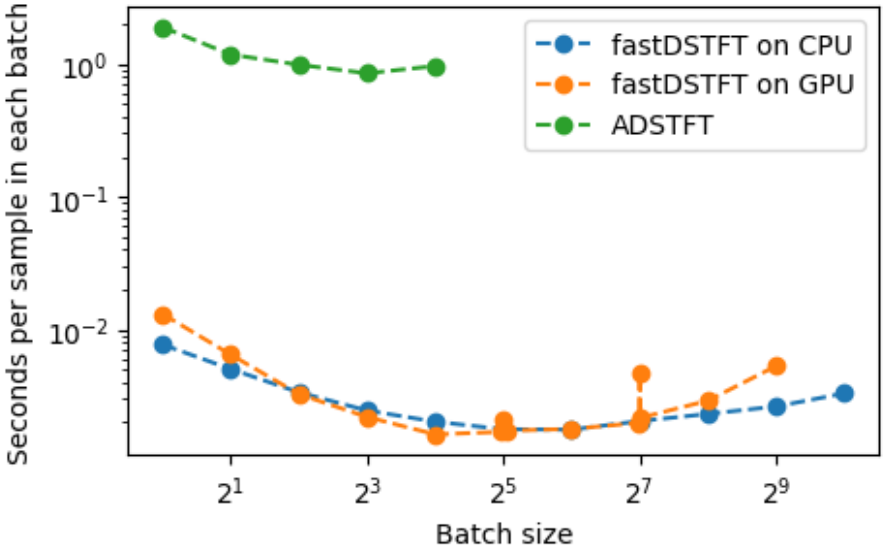

Figure 5: Timing of the proposed and original methods. The original method runs on the CPU.

6 Other notes

The proposed method does not inherently couple F and the maximum window length in the way that the original method does. Therefore, for machine learning use cases, where filtering of the frequency bands will be performed after computing the spectrogram, the number of frequency bands F can be set lower from the beginning to further decrease

computational cost. Since N is proportional to F, Eq. 12 shows that we should expect the computation time to reduce better than linearly when reducing F.

The use of Beta windows is motivated by the well-known result stating that Gaussian windows provide optimal time-frequency concentration. Beta distributions with high $a = b$ are good approximations of Gaussian distributions centered around 0.5 [3], and as such also provide a good concentration. On bounded intervals, which we are dealing with here, using the Beta distribution over truncating the Gaussian has the benefit of having the distribution go to zero along with some of its derivatives at the ends, avoiding a discontinuity that causes ripples in the frequency domain. Thus, the Beta windows provide good time-frequency concentration without the risk of artefacts due to calculating over a bounded interval.

7 Conclusion

This project introduces the fastDSTFT as a computationally fast way to achieve adaptive time-frequency resolution. It is shown to be optimizable using gradients as accurately as the ADSTFT, with around 500 times faster per-sample batched computations.

References

[1] Maxime Leiber, Yosra Marnissi, Axel Barrau, and Mohammed El Badaoui, *Differentiable Adaptive Short-Time Fourier Transform with Respect to the Window Length*,
in *Proceedings of ICASSP 2023 - IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, June 2023, pp. 1–5.
DOI: 10.1109/ICASSP49357.2023.10095245

[2] Carl de Boor, *A Practical Guide to Splines*, Applied Mathematical Sciences, Vol. 27, Springer, Berlin, 2001, Chapter XII, Example 11, pp. 149–151.

[3] EpiX Analytics. *Normal approximation to the Beta distribution*. Available at: https://modelassist.epixanalytics.com/space/EA/26575264/Normal+approximation+to+the+Beta+distribution. Accessed: May 15, 2025.