Stage ARIA — Centre Borelli

Large Scale Convolutional Dictionary Learning For Multivariate Time Series

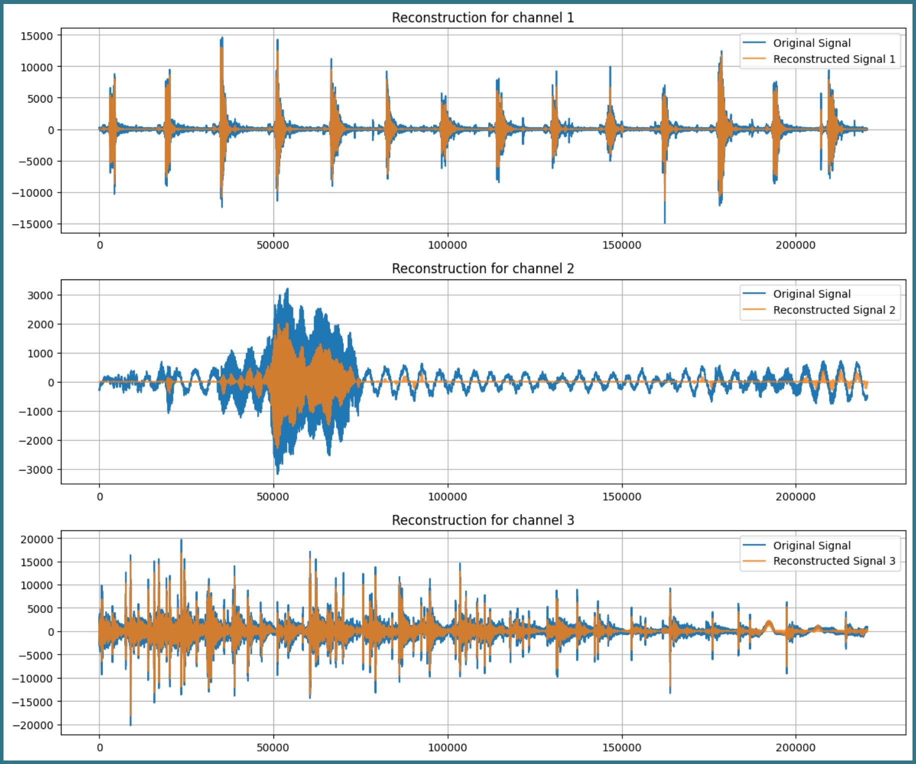


Figure 1: Reconstruction, Denoising and compression of 3 audio signals of 5 seconds each, using convolutional dictionary learning

Rémi Al Ajroudi

Encadrant : Pr. Charles Truong

November 2024 - January 2025

Abstract

CDL (Convolutional Dictionary Learning) is a popular framework for modeling multivariate time series by decomposing signals into convolutional combinations of atoms and activations that in our case our sparse. However, scaling CDL to large datasets remains challenging due to, firstly, the non-convex nature of the optimization problem and secondly, the high computational cost of processing high-dimensional data. In this work, we present an accelerated CDL framework that integrates iterative solvers such as CG (Conjugate Gradient), GMRES (Generalized Minimal Residual), and BiCGStab (Biconjugate Gradient Stabilized) with memory-efficient "LinearOperator" implementations to avoid explicit matrix construction. By exploiting the inherent sparsity of the activation matrix and employing JIT (Just-In-Time Compilation), our method significantly reduces computational time. When applied to audio denoising on the ESC-50 dataset, our approach achieves compression ratios of up to 98.95% while effectively reducing noise. Benchmark results demonstrate the superior efficiency of Conjugate Gradient and BiCGStab in updating the dictionary, particularly as signal lengths and atom sizes increase. Overall, our scalable solution strikes an effective balance between accuracy and computational feasibility. However at this time, it is still not adapted for real-time applications. Additionally, the public availability of our code ensures reproducibility and paves the way for further developments.

Contents

List of Figures

1 Introduction

Time series analysis often involves identifying recurring patterns to enable tasks like denoising, compression, and feature extraction. Convolutional Dictionary Learning (CDL) addresses this by representing signals as sparse combinations of temporally shifted atoms. A method that is particularly effective for capturing transient or periodic structures in data. While traditional CDL frameworks alternate between optimizing a dictionary D and activation coefficients Z, scaling these methods to large datasets remains challenging due to the non-convexity of the joint optimization and the prohibitive memory requirements of direct solvers.

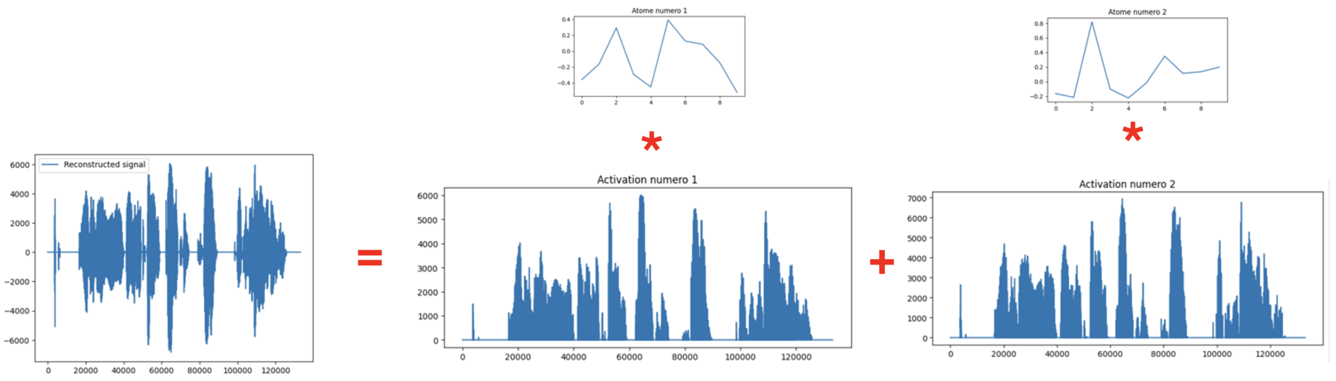


Figure 2: Illustration of the reconstruction of signals using convolutional dictionary learning. The target signal (blue) is modeled as the sum of convolutions between atoms and activations.

As shown in Fig. 2, the target signal is reconstructed using two atoms and their corresponding activations. This corresponds to the sum of the convolutions between the atoms D_k and activation coefficients Z. Formally, the observed signal Y is approximated as:

$$Y_n \approx \sum_{k=1}^{n_{\text{atoms}}} D_k * Z_{n,k}$$

where $*$ denotes the convolution operator.
Existing approaches, such as the ADMM (Alternating Direction Method of Multipliers) for sparse coding, face bottlenecks in dictionary updates where iterative solvers are needed to handle large-scale convolutions. Direct matrix inversion becomes infeasible as signal lengths grow, necessitating the use of "LinearOperators" to implicitly represent convolution operations. Furthermore, real-world applications like audio processing demand not only accuracy but also computational efficiency, especially for deployment on resource-constrained devices (e.g., Raspberry Pi).
This work introduces a scalable CDL framework that integrates three key innovations:

- Iterative Solvers with "LinearOperators": Conjugate Gradient (CG) and BiCGStab (Biconjugate Gradient Stabilized) methods solve least-squares problems without constructing dense matrices, reducing memory usage.
- Sparsity Exploitation: By focusing computations on non-zero activations, convolution and correlation operations are accelerated, enabling efficient handling of 99% sparse Z.
- Compiler Optimizations: The library Numba provide the decorator "njit"[1] that accelerates critical functions, bridging the performance gap between Python and compiled languages.

Validated on noisy audio signals from the ESC-50 [2] dataset, our approach demonstrates robust denoising capabilities and high compression ratios (98.95%). Systematic benchmarks analyze the impact of signal length, atom size, and sparsity on solver performance, revealing BiCGStab's advantages in convergence speed and stability. This work

[1]The `njit` decorator in Numba (Just-In-Time compilation) compiles Python functions into optimized machine code at runtime, significantly improving execution speed by avoiding Python interpreter overhead.

[2]ESC-50 is a publicly available dataset consisting of 2000 environmental audio recordings, categorized into 50 semantic classes (e.g., dog barking, rain, engine noise). It is widely used for benchmarking audio classification and denoising algorithms. `https://github.com/karolpiczak/ESC-50`

contributes a practical toolkit for large-scale CDL, balancing accuracy with computational feasibility for real-time applications.

2 Problem Statement & Challenges

A fundamental challenge in dictionary learning lies in the simultaneous optimization of the dictionary D and the activations Z. The cost function to minimize is typically formulated as a regularized least squares problem:

$$\min_{D,Z} \sum_{n=0}^{nbSignals-1} \frac{1}{2} \left\| Y_n - \sum_{k=0}^{nbAtoms-1} D_k * Z_{n,k} \right\|_2^2 + \lambda \operatorname{Reg}(Z) \quad \text{s.t.} \quad \forall k, \quad \|D_k\|_2 \leq 1.$$

where:

- Y represents the observed data matrix with dimensions $(\text{nbSignals}, T)$.
- D is the dictionary matrix containing the atoms, with dimensions $(\text{nbAtoms}, T_d)$.
- Z contains the activation coefficients, with dimensions $(\text{nbSignals}, \text{nbActivations}, T_z)$.
- $D * Z$ denotes the convolution of D with Z.
- $\|\cdot\|_2$ denotes the l_2-norm.
- $\operatorname{Reg}(Z)$ is a regularization function promoting desirable properties (e.g., sparsity in Z); for example, it could be the l_1 or l_2-norm of Z.
- $\lambda > 0$ is a regularization parameter.

When both D and Z are variables, the optimization problem is non-convex. This non-convexity makes finding the global minimum challenging and computationally expensive. However, if either D or Z is fixed, the problem becomes convex with respect to the other variable.
A widely used approach is to alternate between optimizing D and Z by iteratively updating one while keeping the other fixed (cf. Fig. 3).

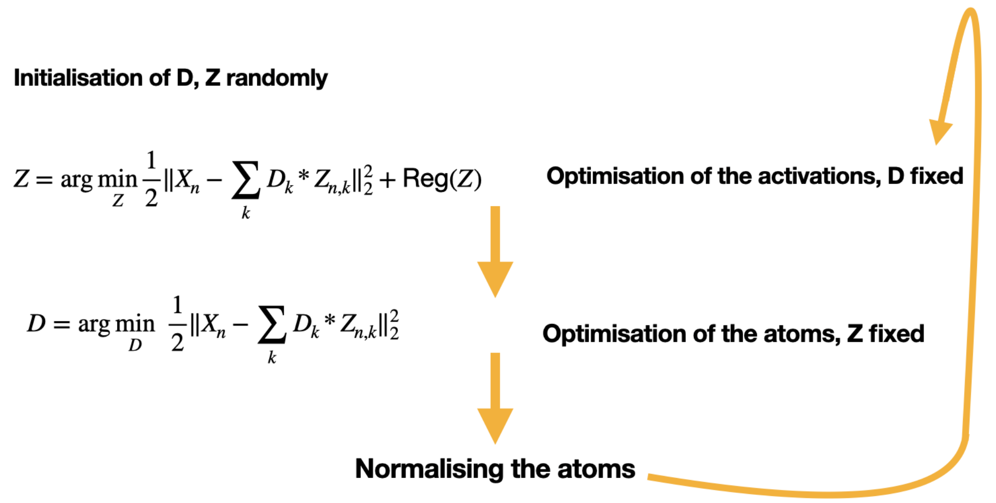

Initialisation of D, Z randomly
$Z = \arg\min_Z \frac{1}{2}\|X_n - \sum_k D_k * Z_{n,k}\|_2^2 + \operatorname{Reg}(Z)$
Optimisation of the activations, D fixed
$D = \arg\min_D \frac{1}{2}\|X_n - \sum_k D_k * Z_{n,k}\|_2^2$
Optimisation of the atoms, Z fixed
Normalising the atoms

Figure 3: Diagram illustrating how alternating optimization works

The second challenge is related to memory cost. Figure 4 illustrates the memory hierarchy in a computer system.

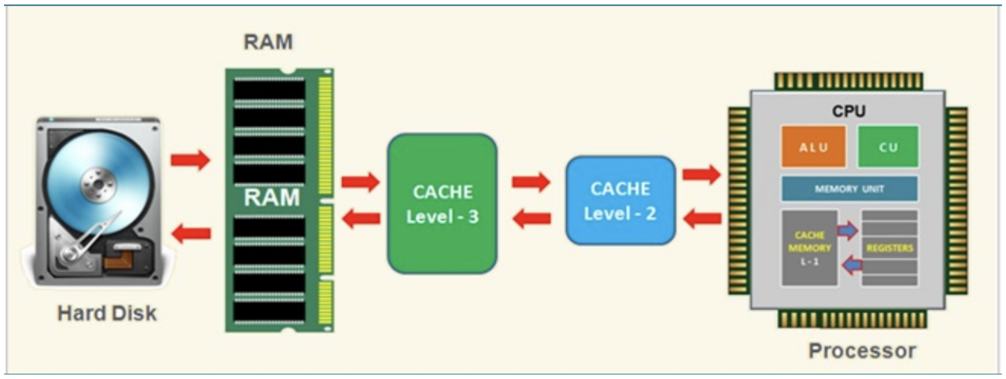


Figure 4: Memory hierarchy of a computer system

As shown in this diagram, the hard disk (on the left) can store the largest amount of data (e.g., PB, TB) but requires the longest access time, whereas the cache (on the right) is located directly on the chip, stores the smallest amount of data (e.g., kB), and offers the fastest access time. In between lies the Random Access Memory (RAM), which can store a relatively large amount of data (e.g., GB) and has an access time longer than that of the cache but much shorter than that of the hard disk. For example, when coding our convolutional dictionary method in Python, the variables are stored in RAM.
When using our CDL framework, we store many variables: those related to the signals to be processed (i.e., all the activations in the matrix Z and all the atoms in D) as well as those related to the optimization (e.g., the gradient of Z or matrices such as $(Z^T Z)^{-1}$). For instance, consider an audio signal sampled at $fs = 48\,\text{kHz}$, where each sample requires 4 bytes of storage. In just a few dozen minutes or hours (depending on the available RAM), the memory can become saturated. We will explore our approach to address this challenge in the context of multivariate sparse coding.
In order to solve the cost function, we will use different linear regression solvers. There are two types of solvers: direct solvers, which use a closed-form solution (e.g., for the least squares problem

$$D = \arg\min_{D} \|AD - Y\|_2^2,$$

where the dictionary matrix is given by

$$\hat{D} = (A^T A)^{-1} A^T Y,$$

and iterative solvers, such as the conjugate gradient or biconjugate gradient stabilized method.
Direct solvers, such as those implemented in "scipy.linalg", do not support "LinearOperator" objects because they typically require explicit access to the entire matrix to perform operations like Gaussian elimination or LU decomposition. These methods rely on the matrix being fully stored in memory, whether in a dense or sparse format. In contrast, iterative solvers (e.g., CG or GMRES) only require the ability to perform matrix-vector products, making them compatible with "LinearOperator", which defines this operation without explicitly storing the matrix. This distinction allows iterative methods to handle large-scale problems efficiently, where explicitly constructing the matrix would be computationally or memory prohibitive.
Therefore, for our study, we will use the iterative methods provided by the "scipy.sparse.linalg" library.

3 Proposed Approach

In convolutional dictionary learning, a major challenge is handling large batches of signals. To optimize the dictionary, we seek to minimize the following cost function:

$$D^\star = \arg\min_{D} \frac{1}{2} \sum_{n=0}^{nbSignals-1} \left\| Y_n - \sum_k D_k * Z_{n,k} \right\|_2^2 .$$

The solution to this least squares problem often involves inverting a large matrix. However, directly inverting or storing this matrix in memory is often infeasible for large datasets due to memory constraints. This is where the

”LinearOperator” class becomes crucial. Instead of explicitly forming and storing the matrix, a ”LinearOperator” allows us to define matrix-vector products via a function.
Given a vector v, the ”LinearOperator” can compute $A * v$ without ever constructing the full matrix A.
By defining a function ”matvec(v)” that returns the result of $A * v$, we can create a ”LinearOperator” object:

$$A_{LinOp} = \text{LinearOperator}(\text{shape} = (m, n), \text{matvec}).$$

This approach is useful when using iterative solvers like conjugate gradient or GMRES, as it significantly reduces memory. This method allows us to numerically solve large-scale problems without fully loading the matrix into memory.
We are solving $A \cdot D = Y$; however, since A is rectangular and certain algorithms require A to be square, we can multiply both sides by A^T to obtain:

$$(A^T A) \cdot D = A^T Y.$$

Thus, we also need to define a function for the transpose of the linear operator, denoted A^T. We call this function ”rmatvec”, such that:

$$A_{LinOp} = \text{LinearOperator}(\text{shape} = (m, n), \text{matvec}, \text{rmatvec}).$$

In the context of convolutional dictionary learning, the linear operator reconstructs the signals Y from a dictionary D, such that

$$A : D \to S, \quad A(D) = Y.$$

Thus, the linear operator maps from the atom space to the signal space (cf. Figure 5).

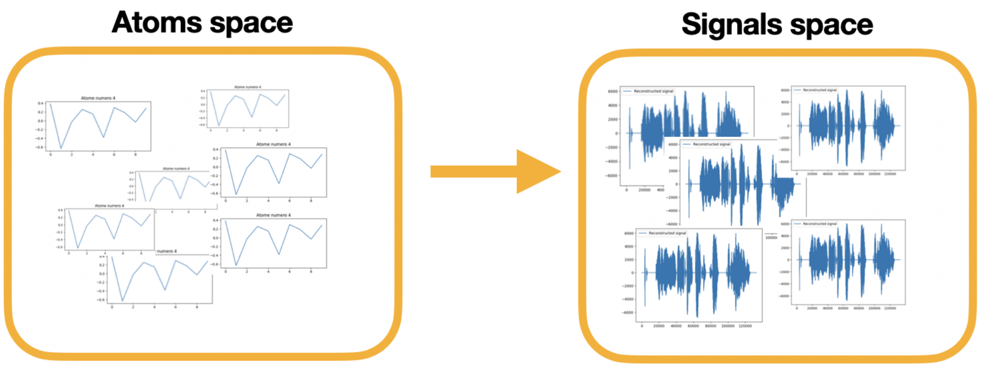


Figure 5: Illustration of the action of the linear operator

Conversely, the transpose of the linear operator, A^T, reconstructs the atoms of the dictionary D from the target signals Y, such that

$$A : S \to D, \quad A(Y) = D.$$

Thus, A^T maps from the signal space to the atom space (cf. Figure 6).

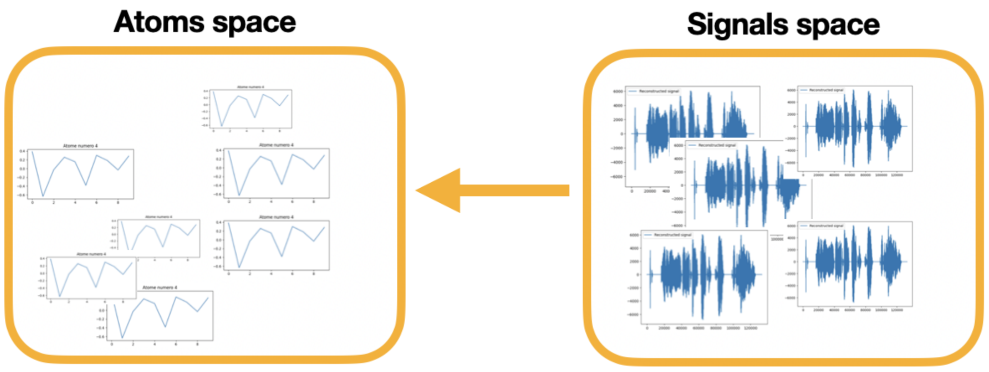


Figure 6: Illustration of the action of the transpose of the linear operator

3.0.1 How to Create the Linear Operator

You can run the code to find the function of the transpose of the linear operator by accessing the python file in the github repository using this link : Link to the code in the github repository.
First, we need to determine the operation we want to implement. In our case, it is the convolution $Z_{n,k} * D_k$. Let the linear operator be A, such that $A \cdot D = Z_{n,k} * D_k$. The linear operator A will be defined using a function we named 'matvec' (A represents a matrix and D a vector), which applies the convolution. Next, we need to define the transpose of the linear operator, A^T, to determine how it operates on a vector.
To understand this, we can use an example with matrices D and Z (in this case Z is simply a matrix because $n = 1$ signal, otherwise it is a tensor). Suppose:

$$D = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad Z = \begin{pmatrix} 0 & 1 & 0 & -1 \end{pmatrix}$$

The convolution formula is given as:

$$y[n] = \sum_{k=0}^{N_h - 1} x[n-k]h[k]$$

A generalistic code written in Python that you can apply for different atoms and activations is available in the appendix.
For the first row of Z and the first column of D:

$$Z * D_1 = \begin{pmatrix} 0 & 1 & 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 0 \\ 1 \cdot 0 + 1 \cdot 0 + 0 \cdot 0 \\ 1 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 \\ 0 \cdot (-1) + 0 \cdot 0 \\ 0 \cdot (-1) \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 0 \end{pmatrix}^T$$

Repeating this for all rows of Z and columns of D, we obtain:

$$A \cdot D_1 = \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \\ d_1 & d_2 & d_3 \\ e_1 & e_2 & e_3 \\ f_1 & f_2 & f_3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \\ f_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ -1 \\ 0 \\ 0 \end{pmatrix}$$

We do the same thing for the 2 other atoms D_2 and D_3, and we find the other coefficients of A :

$$A \cdot D_2 = \begin{pmatrix} a_1 + a_2 \\ b_1 + b_2 \\ c_1 + c_2 \\ d_1 + d_2 \\ e_1 + e_2 \\ f_1 + f_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ -1 \\ -1 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ e_2 \\ f_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

$$A \cdot D_3 = \begin{pmatrix} a_3 \\ b_3 \\ c_3 \\ d_3 \\ e_3 \\ f_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ -1 \end{pmatrix}$$

Therefore we find A:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

We see that the matrix A is a diagonal-block matrix. Comes the transpose of the Linear Operator A^T :

$$A^T = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix}$$

The linear operator A allows to go from the space of atoms to the space of signals with the operation AD = y. Therefore the transpose A^T allows to go from the space of signals to the space of atoms with the operation $A^T y = D$.

$$A^T y = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 2 & 2 & -1 \\ 0 & 2 & 0 \\ -1 & -1 & 2 \end{pmatrix}$$

We found the value of $A^T y$ and we also know that $A^T y = f(Z, y)$ with f a function that we need to determine. We find that this function is the correlation, in fact we perform the correlation between Z and y. For the correlation, we shift D across Z and compute the dot product at each step.
For the first row of Z:

$$\gamma(Z, y_1)[n] = \sum_k Z \cdot y_1[n + k] = \begin{pmatrix} 2 & 0 & -1 \end{pmatrix}$$

Performing this for all columns of y gives us the result found previously, $A^T y$ as:

$$A^T \cdot y = \gamma(Z, y) = \begin{pmatrix} 2 & 0 & -1 \\ 2 & 2 & -1 \\ -1 & 0 & 2 \end{pmatrix}$$

Thus, A^T is the correlation between the activation Z and the signals y.
This detailed example shows how the convolution operation defines the linear operator A, and how the transpose A^T corresponds to the correlation operation.

3.1 Algorithms for Convolutional Sparse Coding Update

In this section, we describe two well-known algorithms for sparse coding: the Alternating Direction Method of Multipliers (ADMM) and the ISTA (Iterative Shrinkage Thresholding Algorithm). These algorithms are used to update the activation matrix Z. However, since the primary focus of this study is on accelerating the dictionary update in large-scale convolutional dictionary learning rather than on the sparse coding component, we provide only a brief overview.

3.1.1 Alternating Direction Method of Multipliers (ADMM)

The explanation of ADMM in this section is based on [1]. Our study primarily addresses the challenges related to optimizing the dictionary D; therefore, we do not focus on the optimization of the activations Z. Nonetheless, for the reader's benefit, we provide an overview of how ADMM works, as it is one of the most commonly used algorithms in recent literature on convolutional sparse coding.
The Alternating Direction Method of Multipliers (ADMM) is an optimization technique that is particularly well-suited for problems with separable objectives and constraints. In the context of convolutional sparse coding, ADMM updates the activation matrix Z by decomposing the problem into simpler subproblems that are solved iteratively. This approach balances fidelity to the observed data with sparsity constraints on Z, making it an effective tool in this setting.

Optimization Problem: The sparse coding problem is formulated as:

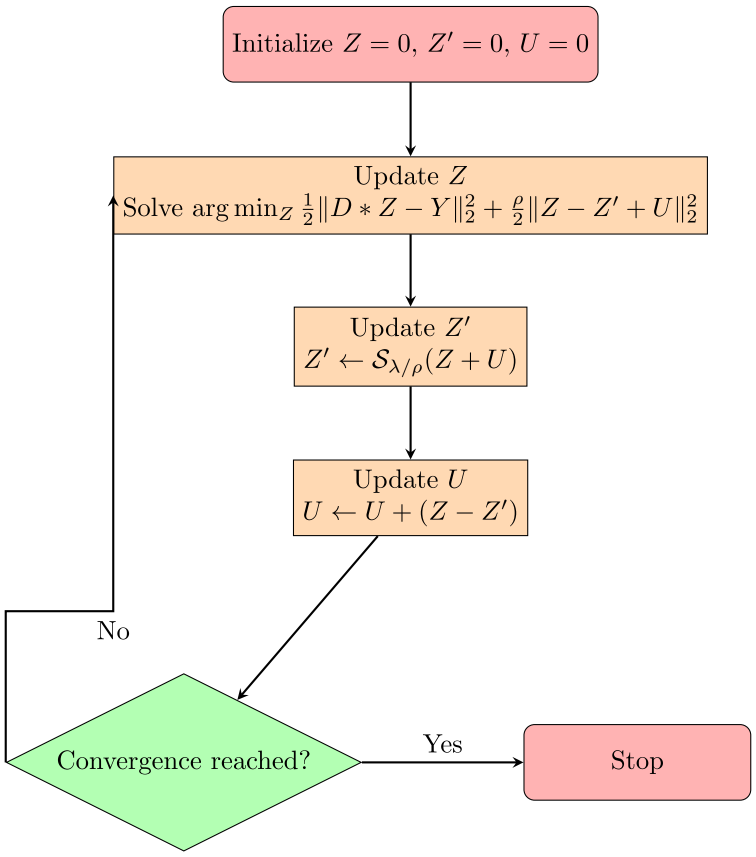
$$\min_{Z} \frac{1}{2}\|D * Z - Y\|_2^2 + \lambda\|Z\|_1,$$

where D is the dictionary, Y is the observed data, and λ controls the sparsity of Z. To apply ADMM, we reformulate the objective by introducing an auxiliary variable Z', yielding:

$$\min_{Z,Z'} \frac{1}{2}\|D * Z - Y\|_2^2 + \lambda\|Z'\|_1, \quad \text{subject to } Z = Z'.$$

Algorithm 1 ADMM for Convolutional Sparse Coding

```
1: Initialize $Z = 0$, $Z' = 0$, $U = 0$, $\rho > 0$, and tolerances $\varepsilon_{\text{prim}}, \varepsilon_{\text{dual}}$. not converged
2: Update $Z$ by solving:
       $Z \leftarrow \arg\min_{Z} \frac{1}{2}\|D * Z - Y\|_2^2 + \frac{\rho}{2}\|Z - Z' + U\|_2^2.$
3: Update $Z'$ using soft-thresholding:
       $Z' \leftarrow \mathcal{S}_{\lambda/\rho}(Z + U).$
4: Update $U$:
       $U \leftarrow U + (Z - Z').$
5: **Return** $Z$.
```



Initialize $Z = 0$, $Z' = 0$, $U = 0$
Update Z
Solve $\arg\min_Z \frac{1}{2}\|D * Z - Y\|_2^2 + \frac{\rho}{2}\|Z - Z' + U\|_2^2$
Update Z'
$Z' \leftarrow \mathcal{S}_{\lambda/\rho}(Z + U)$
Update U
$U \leftarrow U + (Z - Z')$
No
Convergence reached?
Yes
Stop

Figure 7: Flowchart of the ADMM algorithm for convolutional sparse coding.

In summary, ADMM effectively decomposes the sparse coding problem into manageable subproblems that are solved iteratively. This approach ensures the sparsity in the activation matrix Z.

3.2 Algorithms for Dictionary Update

In this section, we describe two well-known algorithms for updating the dictionary in convolutional dictionary learning: Conjugate Gradient (CG) method and the BiConjugate Gradient Stabilized (BiCGStab) method. These algorithms are used to optimize the dictionary D of atoms. Below (cf. Fig. 8) is a conceptual diagram of dictionary learning to understand the mechanisms behind the optimization of the dictionary.

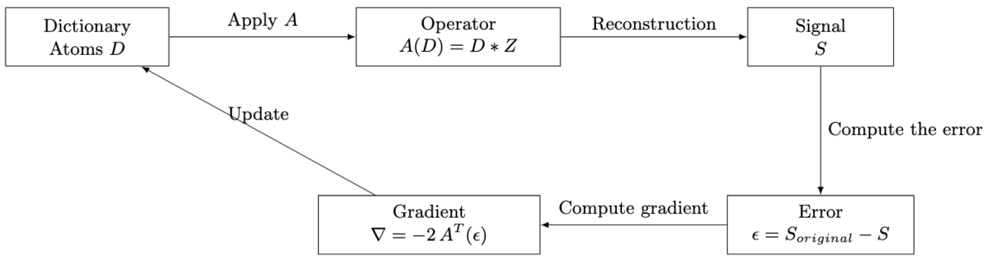


Figure 8: Conceptual diagram of dictionary learning

Since the focus of this study is on accelerating the dictionary update process in large convolutional dictionary learning, we explore the key ideas behind these algorithms and their application to the dictionary learning problem.

3.2.1 Conjugate Gradient

The Conjugate Gradient (CG) method is an iterative algorithm used to solve systems of linear equations where the matrix is symmetric and positive-definite. CG minimizes a quadratic function over a sequence of search directions that are conjugate to each other with respect to the matrix A.
In the context of our problem, which is updating the dictionary of atoms, the CG method aims to minimize the following quadratic objective function:

$$f(D) = \|A \cdot D - Y\|_2^2$$

where A is a linear operator of dimension ($Td * nbAtoms$ x $T * nbSignals$) (that we guarantee symmetric positive-definite matrix by letting A = $A_{old}^T A_{old}$ and $Y = A_{old}^T Y_{old}$), Y of dimension ($T * nbSignals$) is the target data, and D of dimension ($Td * nbAtoms$) represents the dictionary. The optimization problem is to find the dictionary D that minimizes the residual between the dictionary and the data.
The CG method avoids directly calculating the inverse of A, making it much more efficient for large-scale problems compared to direct methods. In the CG method, the concept of conjugate directions is essential. Given a symmetric positive-definite matrix A, two vectors u and v are said to be A-conjugate if:

$$u^\top A v = 0$$

This relationship ensures that the vectors are mutually orthogonal with respect to the inner product defined by A. The CG algorithm constructs a sequence of conjugate directions, ensuring that each new direction is A-conjugate to all previous ones, thus improving the convergence rate of the algorithm. The CG method seeks to minimize the quadratic objective function:

$$J(D) = \frac{1}{2} D^\top A D - b^\top D$$

The gradient of this function is:

$$\nabla J(x) = AD - b$$

At the optimal solution $D_\star$, the gradient is identically zero, i.e., $\nabla J(D_\star) = 0$. At each iteration k, the residual vector r_k is given by:

$$r_k = b - AD_k$$

The residual represents the direction of steepest descent for the objective function. The search direction p_k is updated at each step to ensure that it is A-conjugate to all previous directions, thereby accelerating the convergence. The pseudo-code is as follows:

Algorithm 2 Conjugate Gradient Algorithm for Dictionary Update

```
1: **Input:** Linear operator $A$ (symmetric positive-definite), target data $Y$, initial dictionary $D_0$, tolerance tol,
   maximum iterations max_iter
2: **Initialization:**
3: $r_0 = Y - AD_0$ {Initial residual}
4: $p_0 = r_0$ {Initial search direction}
5: $k = 0$
       $\|r_k\| >$ tol and $k <$ max_iter
6: Compute $q_k = Ap_k$
7: Compute $\alpha_k = \frac{r_k^\top r_k}{p_k^\top q_k}$ {Step size}
8: Update dictionary $D_{k+1} = D_k + \alpha_k p_k$
9: Update residual $r_{k+1} = r_k - \alpha_k q_k$ $\|r_{k+1}\| \leq$ tol
10: **break**
11: Compute $\beta_k = \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$
12: Update search direction $p_{k+1} = r_{k+1} + \beta_k p_k$
13: $k = k + 1$
14: **Output:** Approximate dictionary $D_k$
```

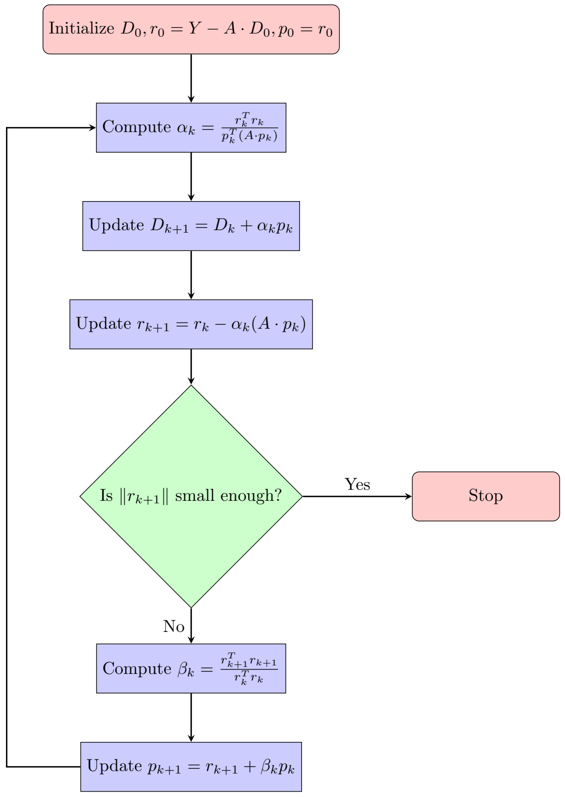

Initialize $D_0, r_0 = Y - A \cdot D_0, p_0 = r_0$
Compute $\alpha_k = \frac{r_k^T r_k}{p_k^T (A \cdot p_k)}$
Update $D_{k+1} = D_k + \alpha_k p_k$
Update $r_{k+1} = r_k - \alpha_k (A \cdot p_k)$
Is $\|r_{k+1}\|$ small enough?
Yes
Stop
No
Compute $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
Update $p_{k+1} = r_{k+1} + \beta_k p_k$

Figure 9: Flowchart of the Conjugate Gradient Algorithm with Explicit Notations

3.2.2 BiConjugate Gradient Stabilized (BiCGStab)

The BiConjugate Gradient Stabilized (BiCGStab) method is an extension of the Conjugate Gradient method, designed to handle non-symmetric and ill-conditioned systems. It improves stability and convergence in such cases.

Algorithm 3 BiConjugate Gradient Stabilized for Dictionary Update

```
1: Initialize $D_0$, residual $r_0 = Y - A \cdot D_0$, and auxiliary vector $\hat{r}_0 = r_0$
2: Set $\rho_0 = \alpha = \omega = 1$, $v_0 = p_0 = 0$ $k = 0, 1, \ldots, K$
3: $\rho_{k+1} = \hat{r}_0^T r_k$ $\rho_{k+1} = 0$
4: Break (breakdown)
5: $\beta = \frac{\rho_{k+1}}{\rho_k} \frac{\alpha}{\omega}$
6: $p_{k+1} = r_k + \beta(p_k - \omega v_k)$
7: $v_{k+1} = A \cdot p_{k+1}$
8: $\alpha = \frac{\rho_{k+1}}{\hat{r}_0^T v_{k+1}}$
9: $s = r_k - \alpha v_{k+1}$ $\|s\|$ is small enough
10: Update $D_{k+1} = D_k + \alpha p_{k+1}$
11: Break
12: $t = A \cdot s$
13: $\omega = \frac{t^T s}{t^T t}$
14: Update $D_{k+1} = D_k + \alpha p_{k+1} + \omega s$
15: Update $r_{k+1} = s - \omega t$ $\|r_{k+1}\|$ is small enough, break
```

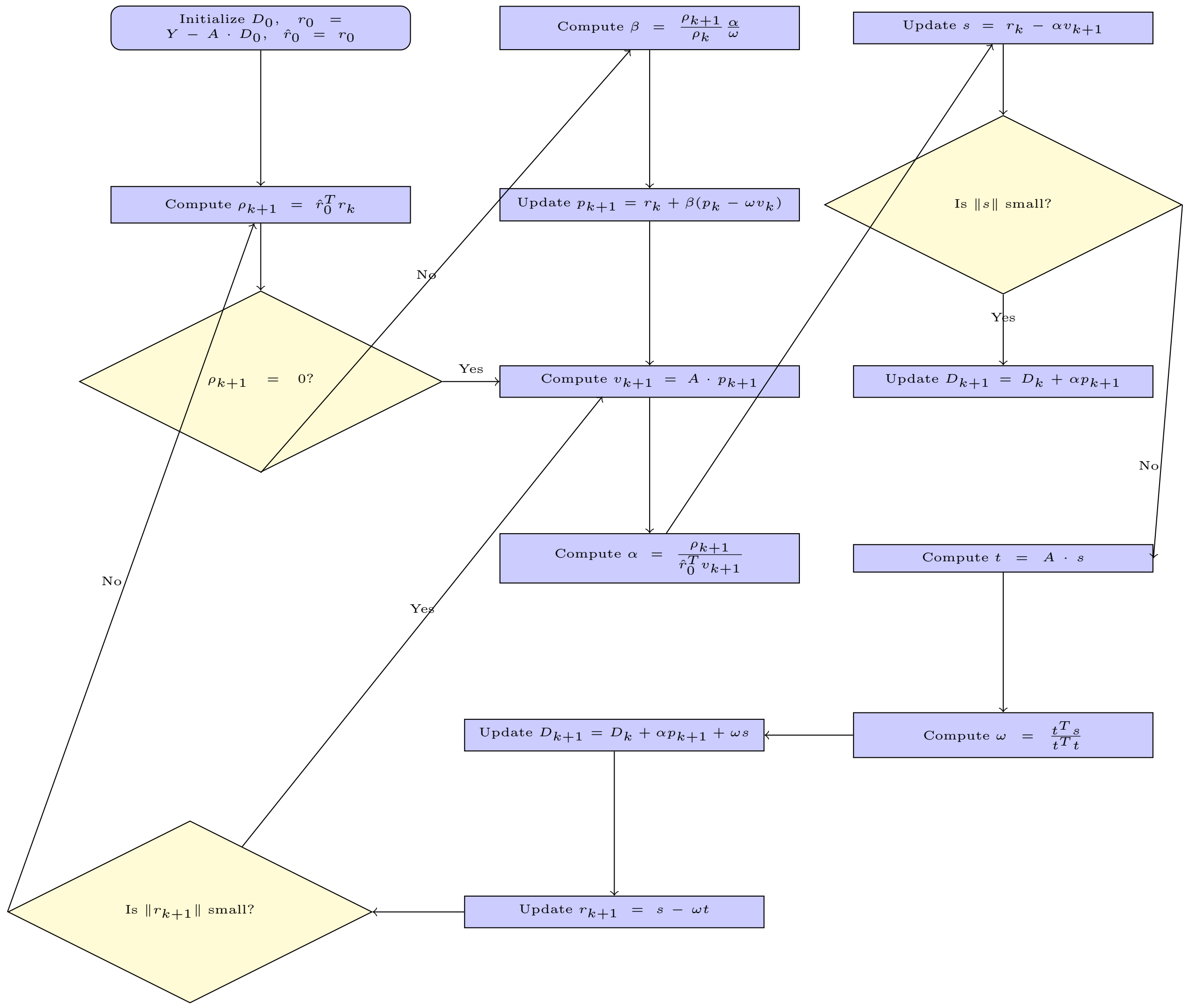

Initialize D_0, $r_0 = Y - A \cdot D_0$, $\hat{r}_0 = r_0$
Compute $\beta = \frac{\rho_{k+1}}{\rho_k} \frac{\alpha}{\omega}$
Update $s = r_k - \alpha v_{k+1}$
Compute $\rho_{k+1} = \hat{r}_0^T r_k$
Update $p_{k+1} = r_k + \beta(p_k - \omega v_k)$
Is $\|s\|$ small?
No
Yes
$\rho_{k+1} = 0$?
Yes
Compute $v_{k+1} = A \cdot p_{k+1}$
Update $D_{k+1} = D_k + \alpha p_{k+1}$
No
Compute $\alpha = \frac{\rho_{k+1}}{\hat{r}_0^T v_{k+1}}$
Compute $t = A \cdot s$
No
Yes
Update $D_{k+1} = D_k + \alpha p_{k+1} + \omega s$
Compute $\omega = \frac{t^T s}{t^T t}$
Is $\|r_{k+1}\|$ small?
Update $r_{k+1} = s - \omega t$

Figure 10: Flowchart of the BiConjugate Gradient Stabilized algorithm for updating the dictionary

3.2.3 Normalization of the atoms

After the optimal dictionary has been found, we normalize the atoms to ensure consistency and prevent ambiguity during the optimization of the activations Z. The normalization is performed as follows:

$$D_k^{\text{normalized}} = \frac{D_k}{\|D_k\|}, \tag{1}$$

where $\|D_k\|$ is the Euclidean norm of the atom D_k.

This step ensures that the optimization process does not result in arbitrary scaling between the dictionary D and the activation matrix Z. Without normalization, the algorithm could converge to solutions where the atoms D_k capture the overall amplitude or shape of the target signal, while the activations Z adjust to compensate for this scaling. Such behavior may lead to overfitting, where the model reproduces the training data too closely and fails to generalize to unseen signals.

3.2.4 How to accelerate the process

One of the main challenges that we want to solve is the computational cost associated with large-scale data. The linear operator A, defined as $A \cdot D = Z * D$, involves convolutions between the activation matrix Z and the dictionary D. Similarly, its transpose A^T, used to compute correlations, requires significant computational resources, especially when Z is large. To address this challenge, we leverage the sparsity of Z.

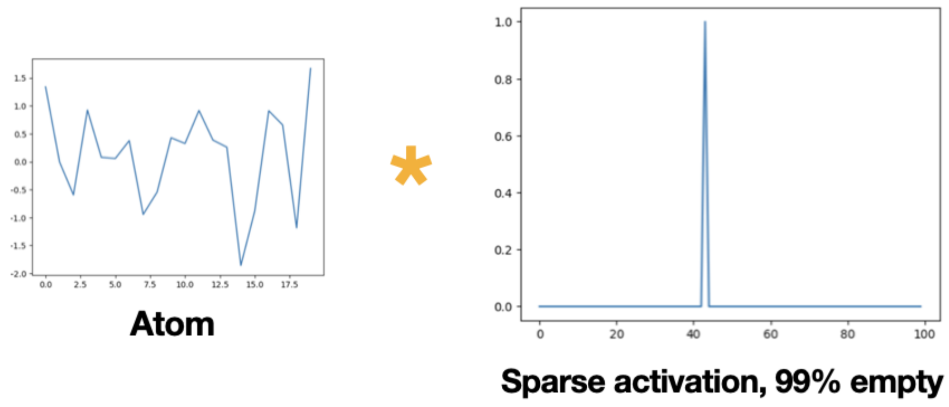


Figure 11: Illustration of the convolution between an atom and sparse activations

To enable efficient computation of the matrix-vector product $A \cdot D$ and its transpose $A^T \cdot s$, we implemented the functions 'matvec' and 'rmatvec'. The matvec function computes the convolution $Z * D$ by iterating over the non-zero elements of Z and updating only the relevant indices of the resulting vector s.
The rmatvec function computes the correlation $A^T \cdot s$, which is equivalent to $Z^T * s$, by propagating back the contributions of s to the atoms of D based on the indices of non-zero values in Z.

These functions are designed to bypass the explicit construction of the full matrices A and A^T, thus enabling efficient memory usage and reduced computational overhead.

Given that Z is sparse, with only a small percentage of its coefficients being non-zero, performing convolution and correlation on all elements would result in unnecessary computations (cf. Fig. 11). To address this, we preprocess Z to extract its non-zero values and their associated indices into a compact representation called non_zero_array. This representation allows us to: Iterate only over the significant entries of Z. Compute convolutions and correlations selectively, avoiding operations on zero entries.

For instance, if Z is 99% sparse, 99% of the time would be spent performing operations on zero entries if the sparsity were not leveraged. This would result in unnecessary computations that do not contribute to the final result. By focusing only on the non-zero entries, we ensure that every operation is meaningful, drastically improving computational efficiency.

This method not only reduces computational complexity but also ensures compatibility with iterative solvers like Conjugate Gradient (CG), where only matrix-vector products are required. The optimized matvec and rmatvec functions ensure that the solver operates efficiently, even when dealing with large-scale problems.

3.2.5 Using a Decorator to Accelerate the Process

One effective method to enhance the speed of the dictionary update algorithm is by using the ”njit” decorator from the numba library. A decorator in Python is a special function that modifies the behavior of another function or method. It acts as a wrapper, allowing additional functionality to be applied without altering the original function's code.

The ”njit” decorator enables JIT (Just-In-Time Compilation) compilation, converting Python code into efficient machine code. Python, being an interpreted language, executes each line of code by compiling it to machine code at runtime.This process incurs additional computational costs, making Python slower than compiled languages like C++ and less suitable for time-sensitive operations such as real-time data processing.
By applying the ”njit” decorator to critical functions like ”matvec” and ”rmatvec”, the code is compiled to machine code upon its first execution. Subsequent calls to these functions bypass the interpretation process, leading to performance gains. This optimization is particularly beneficial in scenarios involving large-scale computations or repeated function calls, where small inefficiencies can accumulate into substantial delays.

3.3 Compression of the signals

In this section, we discuss how signal compression can be achieved using our method. By leveraging the sparsity of the activation matrix Z, which can reach 90% or even 99% of zero coefficients, we can efficiently reconstruct and transmit signals at a significantly reduced cost. Instead of transmitting the full signal, we only need to transmit the non-zero indices of Z, their associated values, and the dictionary D.
Let us break down the calculation:
Consider a signal of length $T = 100,000$. Suppose we have a dictionary D composed of $n_{\text{atoms}} = 5$ atoms, each of size $T_d = 10$, and that Z contains 99% zero entries. To reconstruct the signal, we only need:

- $n_{\text{atoms}} \times T_d = 5 \times 10 = 50$ values for the dictionary D,
- $\text{nnz}(Z) = \frac{1}{100} \times T = 1,000$ non-zero values from Z, along with their indices.

Thus, the total number of indices to transmit is:

$$n_{\text{atoms}} \times T_d + \text{nnz}(Z) = 50 + 1,000 = 1,050.$$

This represents a reduction from the original $T = 100,000$ indices, corresponding to a compression ratio of:

$$\frac{T - (n_{\text{atoms}} \times T_d + \text{nnz}(Z))}{T} \times 100 = \frac{100,000 - 1,050}{100,000} \times 100 = 98.95\%.$$

By focusing only on the non-zero entries of Z, this method avoids redundant computations and unnecessary data transmission. The sparsity of Z is thus a key factor enabling such high compression ratios, especially for large-scale signals.

3.4 Algorithmic Complexity

Let’s examine the algorithmic complexity of the regression solver both without our method and when exploiting the sparsity of the activation matrix.
In the dense case—that is, when all convolutions are computed even if the activation coefficients are zero—the complexity is bounded by

$$O(N \cdot K \cdot T_z \cdot T_d).$$

In contrast, when using our method and if the activation matrix Z is 99% sparse, the complexity is bounded by

$$O(\#\text{Non-zeros} \cdot T_d).$$

For a concrete example, consider a 2-hour audio signal sampled at $f_s = 48\,\text{kHz}$ with a dictionary of 100 atoms and $T_d = 512$, yielding $T_z = 345600$ (i.e., 345600 activation steps) with Z being 99% sparse. We choose $T_d = 512$ because the atom length is not arbitrary—it must be selected according to the context of the signals studied. In our example, since

$$\frac{T_d}{f_s} = \frac{512}{48000} \approx 10\,\text{ms},$$

this represents a reasonable time interval to capture interesting audio dynamics. The algorithmic complexity is then bounded by:

- $O(10^{10})$ in the dense case,
- $O(10^8)$ in the optimized case.

Furthermore, RAM usage would be prohibitive just for storing the coefficients, not to mention the additional variables needed for optimization. For instance, if each sample requires 4 bytes of storage, then storing all coefficients would require

$$N \cdot n_{\text{atoms}} \cdot T_z \cdot 4\,\text{B} = 1 \cdot 100 \cdot 345600 \cdot 4 \approx 132\,\text{GB},$$

which is very high for computers that typically have 8, 16, or even 32 GB of RAM. In contrast, with the optimized method, one would need to store

$$N \cdot n_{\text{atoms}} \cdot \Big((1\% \cdot T_z) \cdot \dim(Z)\Big) \cdot 4\,\text{B} = 1 \cdot 100 \cdot \Big((1\% \cdot 345600) \cdot 3\Big) \cdot 4 \approx 4\,\text{MB}.$$

To summarize, consider the following table:

Approach	Impact of Z on RAM	Computation Time $A(D)$	Complexity
Dense (no optimization)	$1 \cdot 100 \cdot 345600 \cdot 4\,\text{B} = 132\,\text{GB}$	$O(10^{10})$	$O(N \cdot K \cdot T_z \cdot T_d)$
99% Sparse Z (optimized)	$1 \cdot 100 \cdot \Big((1\% \cdot 345600) \cdot 3\Big) \cdot 4\,\text{B} = 4\,\text{MB}$	$O(10^8)$	$O(\#\text{Non-zeros} \cdot T_d)$

4 Use case

In this section we detail how we are going to apply our convolutional dictionary learning framework to a real use-case.

4.1 Audio signals

The use case selected for this study focuses on audio signals. Specifically, we extracted 10 audio clips of 5 seconds each from the publicly available dataset "ESC-50" (cf. [3]). To simulate noisy scenarios, Gaussian noise with zero mean and a standard deviation corresponding to 10% of the maximum amplitude of each signal was added to these audio clips.
The idea is to leverage our convolutional dictionary learning framework to optimize a dictionary D and an activation matrix Z such that the original signals can be reconstructed. Since the reconstruction will not be perfect, the reconstructed signals are expected to contain significantly less noise, or ideally, no noise at all. This process effectively results in a form of noise reduction.
To evaluate the performance of the framework, we quantify the quality of the reconstructed signals using several metrics. These metrics are computed by comparing:

- the reconstructed signal to the original clean signal (without noise),
- the reconstructed signal to the noisy signal.

This dual comparison allows us to assess both the fidelity of the reconstruction and the extent of noise reduction achieved.

4.2 Real time usage

To validate deployment on resource-constrained devices, the framework was tested on a Raspberry Pi 5 (8GB RAM, ARM Cortex-A72).
Experiments (cf. section Results) shows that the computation time required to perform CDL on 5 seconds long audio signals sampled at a frequency of 44.1 kHz still requires a time superior to 5 seconds despite the great improvements with the initial unoptimized methods. Therefore the real time processing on an embedded system like a Raspberry Pi is not possible with the framework. It would require further optimization or a use of parallel computation that is avaible on specific embbeded systems such that the Nvidia Jetson nano that embark a GPU of 128 threads.

5 Results

In this section, we analyze the results obtained through the various experiments discussed in the preceding sections.

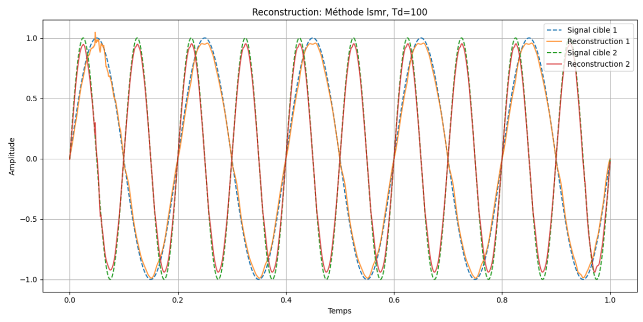


Figure 12: Exemple of the reconstruction of the target signals with CDL during the benchmark

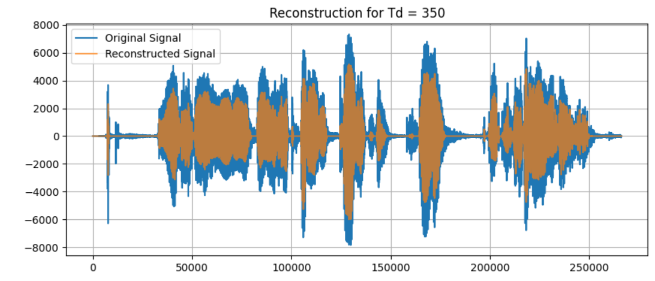


Figure 13: Reconstructed audio signal using CDL

5.1 Benchmarking Solvers

To identify the most efficient iterative solver for optimizing the dictionary D given a fixed activation matrix Z, we benchmark several solvers, namely, Conjugate Gradient (CG), Least Squares Minres (LSMR), Least Squares QR (LSQR), Biconjugate Gradient Stabilized (BICGSTAB), Generalized Minimal Residual (GMRES) in their non-optimized forms.
We perform two types of benchmarks. The first quantifies the performance gains achieved by accelerating the optimization methods for computing the optimal dictionary. The solver performance is evaluated by conducting multiple benchmarks in which key parameters of the convolutional dictionary learning framework are varied while others are held constant. Each benchmark is executed under two configurations: non-optimized and optimized solvers. The investigated parameters include:

- Signal Length (T): Varying the length of the signal (and activations, denoted T_z), with computation time analyzed as a function of T. Longer signals often lead to increased computational complexity, and this parameter assesses how solvers handle scalability with respect to input size.
- Atom Length (T_d): Varying the length of the dictionary atoms, with computation time plotted against T_d. Larger atoms may involve more complex operations, thus affecting solver performance.
- Number of Atoms and Activations ($n_{\text{atoms}}, n_{\text{activations}}$): Varying the size of the dictionary D and the activations Z, with computation time analyzed as a function of n_{atoms}. A larger dictionary generally impacts the sparsity and the complexity of the optimization problem, influencing convergence speed and computational time.

Performance metrics include:

- Convergence Speed: The number of iterations required for the solver to converge within a given tolerance. Faster convergence is a desirable property, particularly when dealing with large-scale problems.
- Computational efficiency: Total time required to reach convergence. This criterion evaluates the computational cost of solvers, a crucial aspect in applications where execution speed is critical.

This benchmarking process highlights the trade-offs between computational efficiency and scalability (capacity of handling large-scale multivariate signals) for both non-optimized and optimized solvers in the context of convolutional dictionary learning.
The second type of benchmark evaluates how well the target signal is reproduced after optimization. Various metrics are used to quantify the quality of the reconstruction, that were found in research papers on audio signals reconstruction in the context of convolutional dictionary learning (cf. [7] [8]), including:

- Peak Signal-to-Noise Ratio (PSNR): Evaluates the quality of the reconstructed signal by comparing it to the original in terms of signal fidelity. It is defined as:

$$\text{PSNR} = 20 \log_{10} \left(\frac{\max(|x|)}{\sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \hat{x}_i)^2}} \right)$$

where x_i and $\hat{x}_i$ are the original and reconstructed signal values, respectively, N is the number of samples, and $\max(|x|)$ represents the maximum absolute value of the original signal. A higher PSNR indicates a more accurate reconstruction, as it reflects a lower distortion level.

- Signal-to-Interference Ratio (SI-SDR): Quantifies the quality of the reconstructed signal by comparing it to the interference (for e.g. noise or distortion). A higher SI-SDR value indicates a better separation between the target signal and the interference. It is defined as:

$$\text{SI-SDR} = -10 \log_{10} \left(\frac{\|\hat{s} - \alpha s\|^2}{\|\alpha s\|^2} \right)$$
$$\text{with } \alpha = \frac{\langle \hat{s}, s \rangle}{\|s\|^2}$$

where s is the original signal and $\hat{s}$ is the reconstructed signal. A higher SI-SDR value indicates better reconstruction, as it reflects a smaller discrepancy between the original and reconstructed signals.

- Short-Time Fourier Transform (STFT) Loss: This metric measures the difference in frequency content between the original and reconstructed signals by comparing their spectrograms. The STFT is a method of transforming a signal into a representation of its frequency content over time, thus it is especially adapted for non-stationary signals. The STFT loss is calculated as the difference in the magnitude of the STFT of the original and reconstructed signals. A smaller STFT loss indicates that the reconstructed signal preserves the frequency components of the original signal better.

- Mel Frequency Cepstral Coefficients (MEL) Loss: This metric compares the Mel-frequency spectrograms of the original and reconstructed signals. The Mel scale approximates human auditory perception, making this metric particularly useful for speech or audio signals. The Mel loss is computed by comparing the Mel spectrograms of the original and reconstructed signals, with a lower loss indicating a reconstruction that better preserves perceptual features. The Mel spectrogram is computed by transforming the signals into Mel scale features, which are then compared to quantify the difference.

When conducting these benchmarks, we require a well-tested baseline trusted by the scientific community for effective comparison with our proposed methods. In our study, we adopt the optimization techniques implemented in the Python library AlphaCsC. Specifically, we use the ISTA method for optimizing the activation matrix Z and employ projected gradient descent as the baseline for optimizing the dictionary D.

5.2 Benchmark 1: Optimizing D with Fixed Z

In the first experiment, the signal length T is varied between 2500 and 4000, which means that we vary the length of the activation map, Tz. The activation matrix Z is manually set, such that it is 99% sparse and that it has 10 atoms, and the dictionary D is optimized accordingly, to this given Z, thus, without alternating optimization. This setup isolates the computational efficiency of the solvers without evaluating reconstruction accuracy.

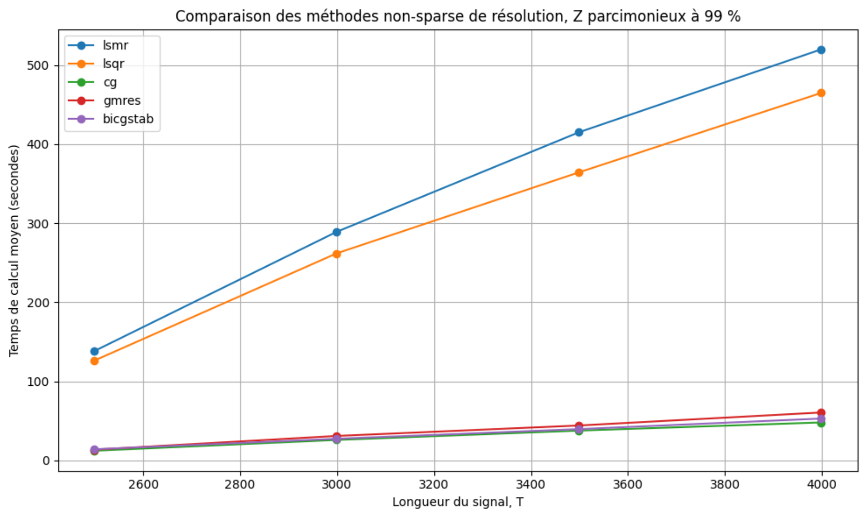
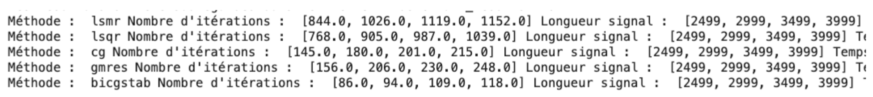


Figure 14: Computation time using non-optimized solvers.



```
Méthode :  lsmr Nombre d'itérations :  [844.0, 1026.0, 1119.0, 1152.0] Longueur signal :  [2499, 2999, 3499, 3999]
Méthode :  lsqr Nombre d'itérations :  [768.0, 905.0, 987.0, 1039.0] Longueur signal :  [2499, 2999, 3499, 3999] Te
Méthode :  cg Nombre d'itérations :  [145.0, 180.0, 201.0, 215.0] Longueur signal :  [2499, 2999, 3499, 3999] Temps
Méthode :  gmres Nombre d'itérations :  [156.0, 206.0, 230.0, 248.0] Longueur signal :  [2499, 2999, 3499, 3999] Te
Méthode :  bicgstab Nombre d'itérations :  [86.0, 94.0, 109.0, 118.0] Longueur signal :  [2499, 2999, 3499, 3999]
```

Figure 15: Number of iterations required for each solver.

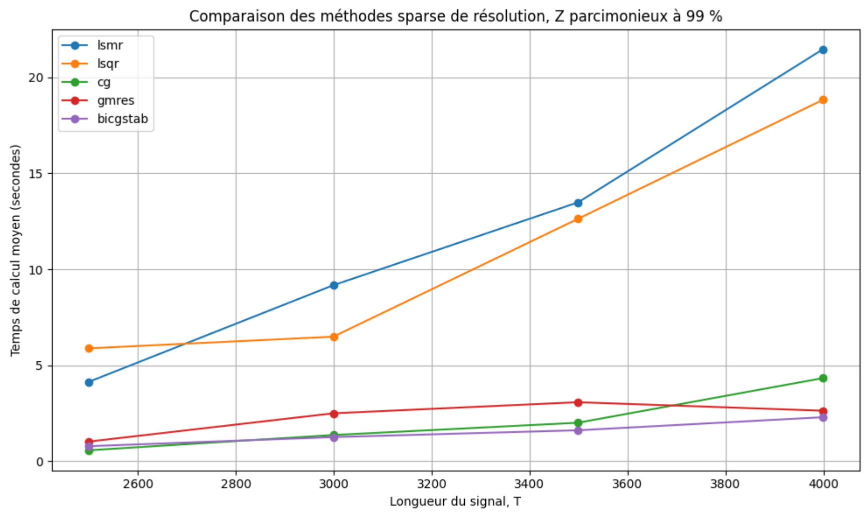
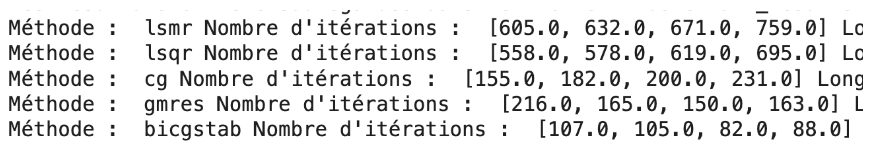


Figure 16: Computation time using optimized solvers.



```
Méthode :  lsmr Nombre d'itérations :  [605.0, 632.0, 671.0, 759.0] Lo
Méthode :  lsqr Nombre d'itérations :  [558.0, 578.0, 619.0, 695.0] Lo
Méthode :  cg Nombre d'itérations :  [155.0, 182.0, 200.0, 231.0] Long
Méthode :  gmres Nombre d'itérations :  [216.0, 165.0, 150.0, 163.0] L
Méthode :  bicgstab Nombre d'itérations :  [107.0, 105.0, 82.0, 88.0]
```

Figure 17: Number of iterations required for each optimized solver.

The figure 15 shows the computational time for each of the methods LSMR (Least Squares Minimal Residual), LSQR (Least Squares QR), CG (Conjugate Gradient), BiCGSTAB (Biconjugate Gradient Stabilized), GMRES (Generalized Minimal Residual) to find the optimal dictionary D, when they are unoptimized, meaning that the activations Z and the atoms D are performed over all the coefficients of Z, including all the zeros, which represent 99% of the activation values. Meanwhile, the figure 17 shows the computational time for the same methods to find the optimal dictionary D, when they are optimized with our method, meaning only the useful activation coefficients are used and the functions associated with the linear operator and its transpose are compiled using the njit decorator. Firstly, for both case scenarios, we can clearly discriminate a fast group and a slow group of methods. The LSMR and LSQR methods require more iterations to converge and are slower than the CG, GMRES and BICGSTAB methods. Secondly, while the global computational time to run all the benchmarks took around 1 hour for the unoptimized methods, the second benchmark took 3,40 minutes, therefore we succesfully accelerated the benchmark by a factor of 17,6.

5.3 Benchmark 2: Alternating Optimization

In the second benchmark, alternating optimization is performed between the dictionary D (using our methods) and the activation matrix Z (using the Iterative Shrinkage-Thresholding Algorithm (ISTA) method provided by AlphaCSC). The ISTA implementation remains unoptimized, while our contributions focus solely on improving dictionary updates. Each benchmark uses default parameters, varying one to assess its impact on computational efficiency and reconstruction accuracy.

5.3.1 Varying the Atom Length (T_d)

This benchmark is performed by varying the length of the atoms in the dictionary D. The atom length is an important parameter that must be set wisely, as it depends on the context. For example, in our case we are reconstructing audio signals sampled at 48 kHz. Thus, if we want the atom length T_d to capture the relevant dynamics in the signal, it must be neither too short nor too long. For instance, if $T_d = 500$, then

$$\frac{T_d}{f_s} \approx 10\,\text{ms},$$

which is a relevant time interval to capture the dynamics in an audio signal. For the benchmark, we set a default configuration and then vary only one parameter, namely T_d. The default parameters are:

- Number of atoms: 5
- $T_z \approx 266240$
- Z sparsity: 97%
- Number of alternations in the optimization of D and Z: 4

The atom length T_d is varied between 5 and 1000, and the impact on computational performance is analyzed.

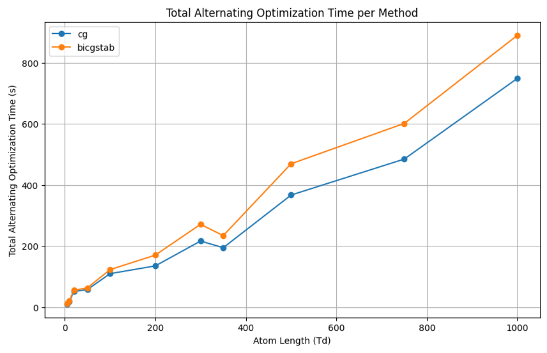


Figure 18: Computation time using optimized solvers.

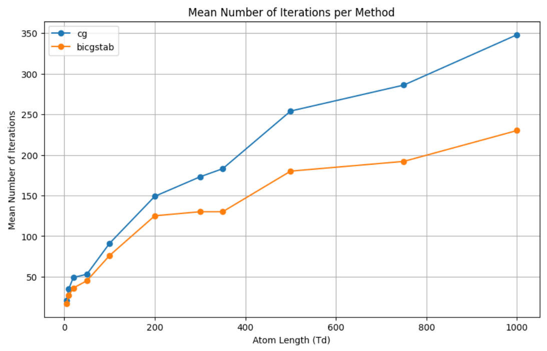


Figure 19: Number of iterations required for convergence.

Figure 18 shows the computational time and Figure 19 the number of iterations required when using the CG (Conjugate Gradient) and the BiCGSTAB (Biconjugate Gradient Stabilized) methods to find the optimal dictionary D. The computational time also includes the time needed by the ISTA method to compute the optimal activation matrix Z, with a total of 4 alternations (i.e., D and Z are optimized 4 times). Only these two methods were retained for the benchmark since they proved to be the fastest in the previous tests, and the purpose of this benchmark is now to determine the optimal hyperparameters for reconstructing the audio signal.
As can be seen, although the number of iterations required to converge increases more for CG than for BiCGSTAB as the atom length increases, the overall computational time for CG remains lower than that for BiCGSTAB, indicating that CG is more time-efficient for solving the problem.

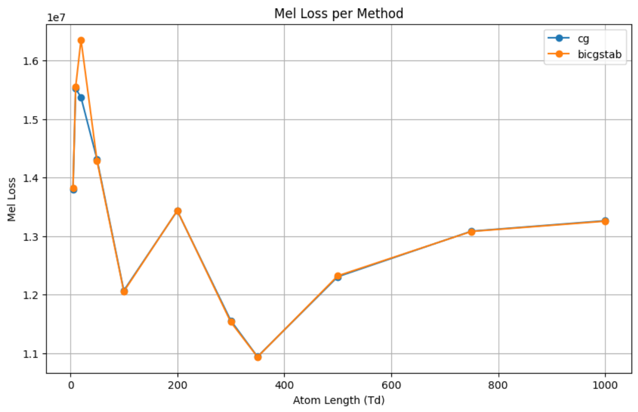


Figure 20: Variation of the MEL loss metric as T_d increases.

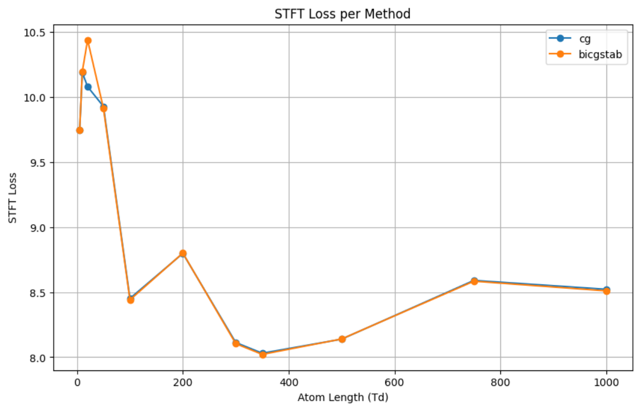


Figure 21: Variation of the STFT loss metric as T_d increases.

Figure 21 illustrates the evolution of the STFT loss for each atom length T_d used during the benchmark, while Figure 20 shows the evolution of the MEL loss. The overall structure of both graphs is similar, with a peak at $T_d = 5$, a minimum at $T_d = 350$, and then a steady increase. Since these loss metrics are such that lower values indicate better performance, the optimum atom length is achieved at $T_d = 350$. This is a reasonable choice because, as mentioned earlier, the atom length is responsible for capturing the dynamics of the audio signal; indeed, we have

$$\frac{T_d}{f_s} = 7.3\,\text{ms},$$

which corresponds to an appropriate time interval for capturing the relevant audio dynamics.

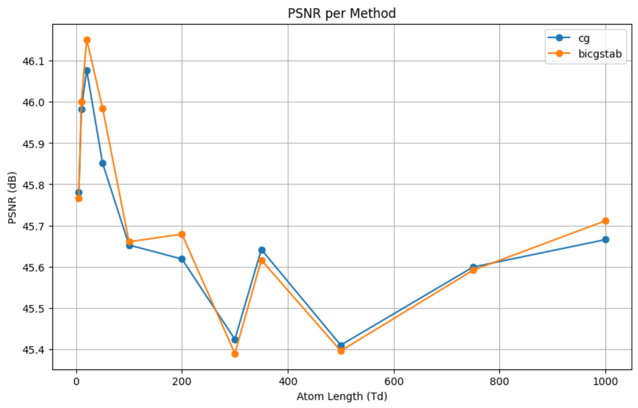


Figure 22: Variation of the PSNR metric as T_d increases.

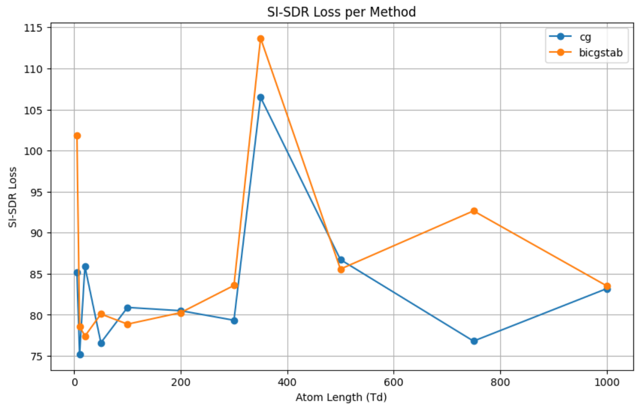


Figure 23: Variation of the SI-SDR metric as T_d increases.

Figure 22 shows the evolution of the Peak Signal-to-Noise Ratio (PSNR) for each atom length T_d used in the benchmark, while Figure 23 shows the evolution of the Scale-Invariant Signal-to-Distortion Ratio (SI-SDR). The trends are more complex: in Figure 22, the PSNR initially reaches a maximum at $T_d = 5$, then decreases to a global minimum around $T_d = 300$, reaches a local maximum at $T_d = 350$, decreases again at $T_d = 500$ to the same global minimum, and finally rises. In Figure 23, the metric oscillates moderately for T_d between 5 and 100, then increases significantly to peak at $T_d = 350$ before decreasing.
One interpretation is that for low T_d values, the atoms capture only very short, local details, which can lead to an overfitting effect on small segments and result in a deceptively high PSNR without capturing the overall signal dynamics. Therefore, the optimal value of T_d is more likely to be at the local maximum corresponding to $T_d = 350$, since an atom length in that range better captures the interesting dynamics of audio signals. Moreover, the SI-SDR metric, which indicates the level of distortion in the reconstructed signal, reaches a global maximum at $T_d = 350$, further confirming that this is the ideal atom length for reconstruction.

5.3.2 Comparison with the Baseline

We conducted the experiment under the same conditions using the baseline; the results are presented below.

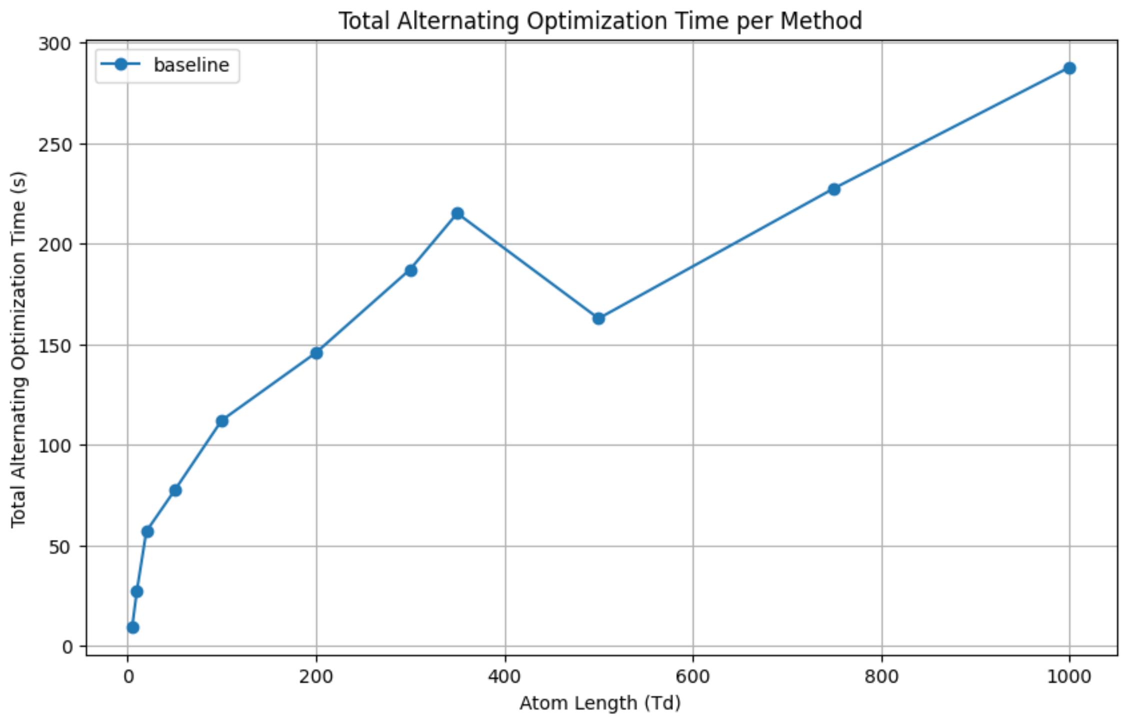


Figure 24: Computation time using the baseline.

Figure 24 shows the computation time taken by the baseline for the optimization. The overall trend is an increase in computation time as the atom length T_d increases. We notice that our optimized methods, especially the optimized Conjugate Gradient, perform slightly better when T_d is less than or equal to 350, which, in our case, is the optimal atom length.

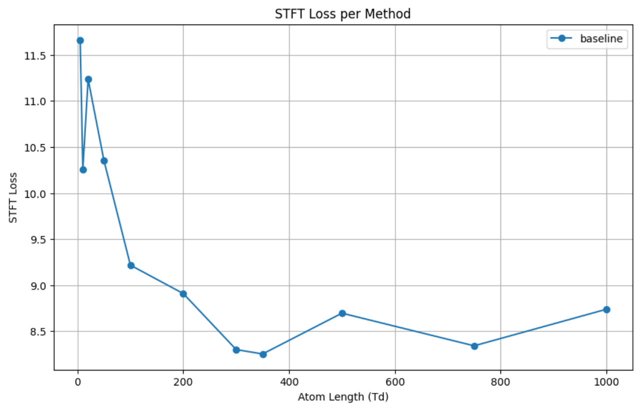


Figure 25: Variation of the STFT metric as T_d increases.

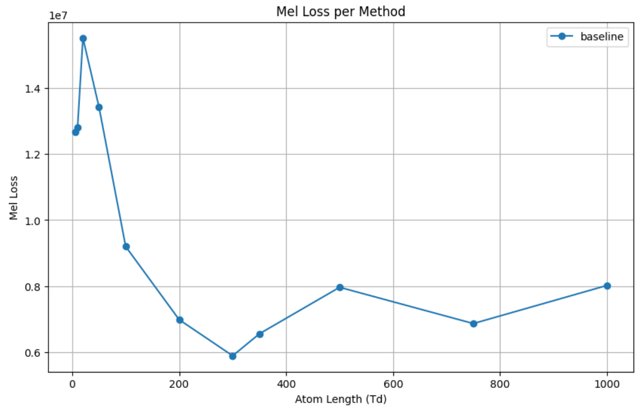


Figure 26: Variation of the MEL metric as T_d increases.

At $T_d = 350$, the MEL loss is around 0.65 for the baseline, compared to approximately 1.1 for our methods indicating that the baseline performs better in terms of MEL loss. However, regarding the STFT loss, our methods achieve a value of 8, while the baseline reaches 8.25, so our approach is superior on this metric.

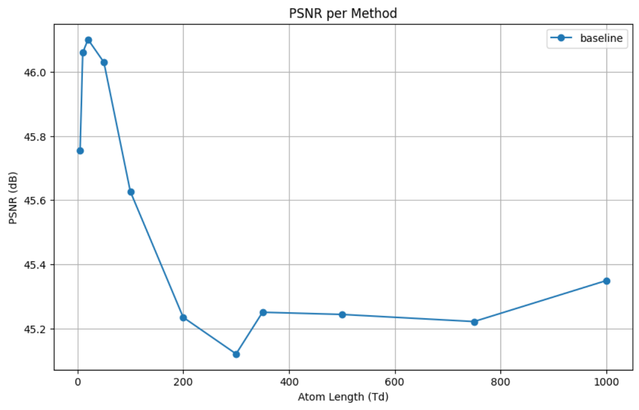


Figure 27: Variation of the PSNR metric as T_d increases.

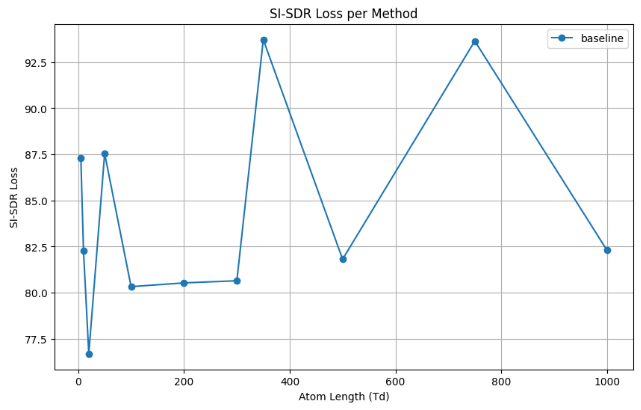


Figure 28: Variation of the SI-SDR metric as T_d increases.

Again, at $T_d = 350$, our methods achieved a PSNR of 45.4 and an SI-SDR of 79, whereas the baseline yielded a PSNR of 45.1 and an SI-SDR of 94. Thus, our PSNR is slightly higher, but the baseline exhibits a better SI-SDR. In summary, our methods and the baseline are equivalent in terms of reconstruction quality (both in frequency and shape/distortion) and computational time for the selected optimal atom length $T_d = 350$.

5.3.3 Varying the Number of Atoms

This benchmark is performed by varying the number of atoms in the dictionary D. The number of atoms is a crucial parameter since it determines the variety of forms that the dictionary can replicate. For this benchmark, we fix a default configuration and vary only the number of atoms. We use $T_d = 350$ as determined in the previous benchmark. The default parameters are:

- $T_d = 350$
- $T_z = 266240$
- Z sparsity: 97%
- Number of alternations in the optimization of D and Z: 4

The benchmark evaluates three methods: Conjugate Gradient (CG), Biconjugate Gradient Stabilized Method (BICGSTAB), and Generalized Minimal Residual Method (GMRES). Although we initially retained only CG and BICGSTAB, GMRES was added to assess potential improvements, despite a significant increase in computation time. The number of atoms is varied between 2 and 20.

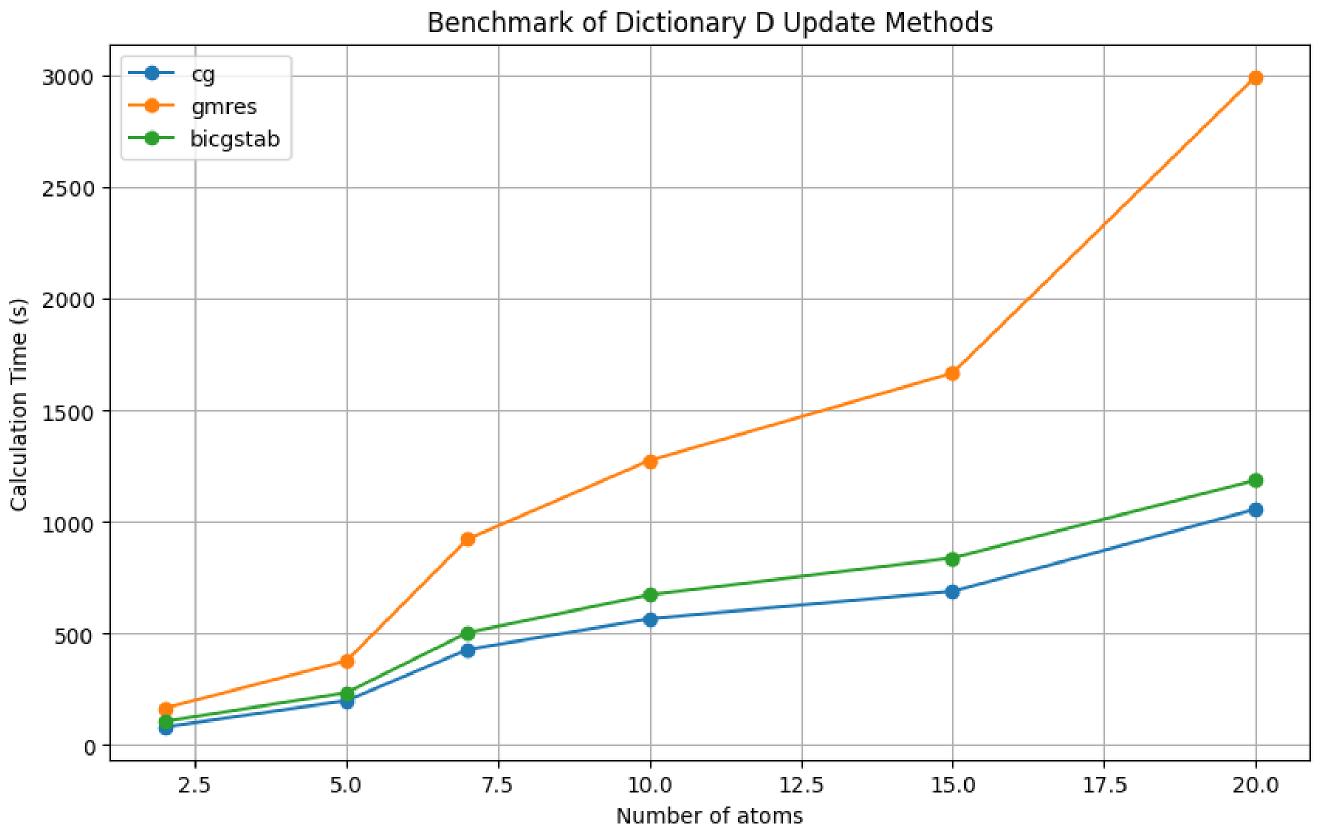


Figure 29: Computation time using optimized methods.

As shown in Figure 29, the computation time increases in a piecewise linear manner, being lowest for 2 atoms and highest for 20 atoms.

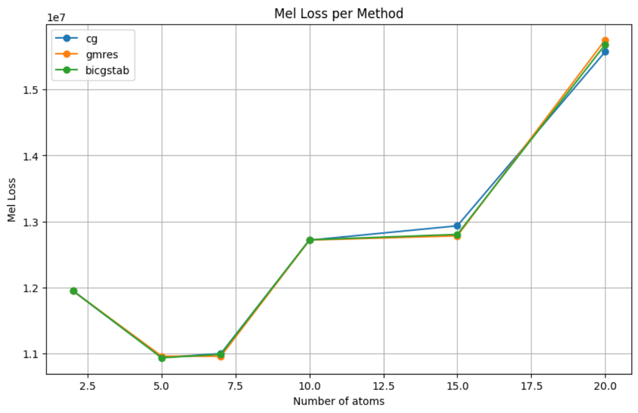


Figure 30: Variation of the MEL loss metric as the number of atoms increases.

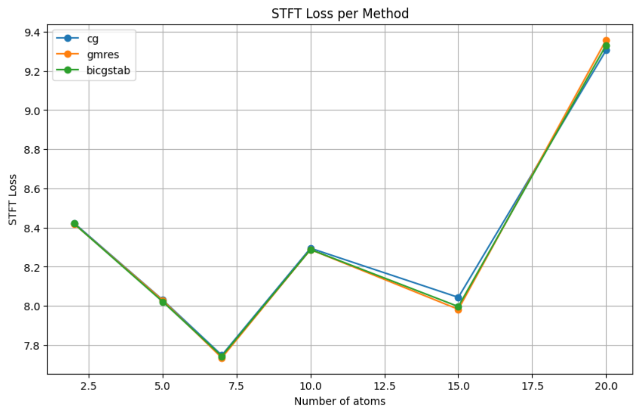


Figure 31: Variation of the STFT loss metric as the number of atoms increases.

In Figure 30, the MEL loss initially decreases to a global minimum for a number of atoms between 5 and 7, then continuously increases. In contrast, the STFT loss (Figure 31) first decreases to a global minimum at 7 atoms, then increases and oscillates before rising again. Based on these metrics, the framework appears to best reproduce the audio signal when the number of atoms is around 7, which is reasonable since it provides sufficient variety in atom shapes without excessively increasing computation time.

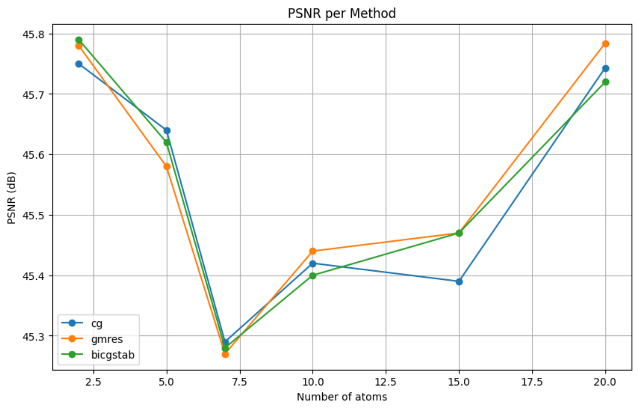


Figure 32: Variation of the PSNR metric as the number of atoms increases.

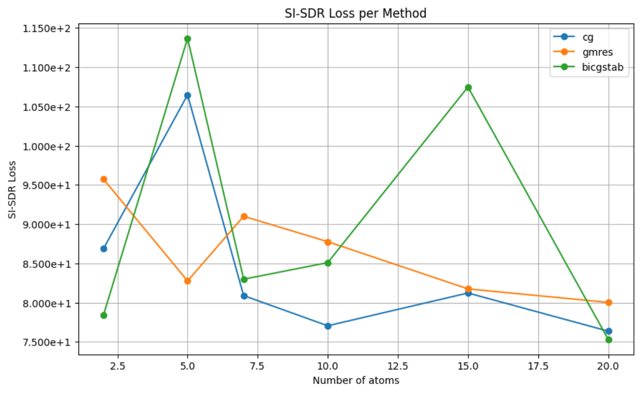


Figure 33: Variation of the SI-SDR metric as the number of atoms increases.

Figure 32 shows that the PSNR is highest for 2 atoms, decreases to a global minimum at 7 atoms, and then rises again, reaching a global maximum at 20 atoms. In contrast, Figure 33 indicates that the SI-SDR starts at a minimum for 2 atoms—likely because too few atoms lead to a highly distorted reconstruction due to insufficient representational variety—then reaches a global maximum at 5 atoms before decreasing as the number of atoms increases.

For the PSNR and SI-SDR metrics, higher values indicate better performance. We place more importance on the SI-SDR than on the PSNR because a high PSNR combined with a low SI-SDR can occur when the reconstruction minimizes local errors (resulting in a high PSNR) while failing to capture the overall signal structure, thereby producing significant distortion (reflected by a low SI-SDR). Therefore, the optimal number of atoms is determined to be 5, which is consistent with the MEL and STFT loss observations.

5.3.4 Comparison with the Baseline

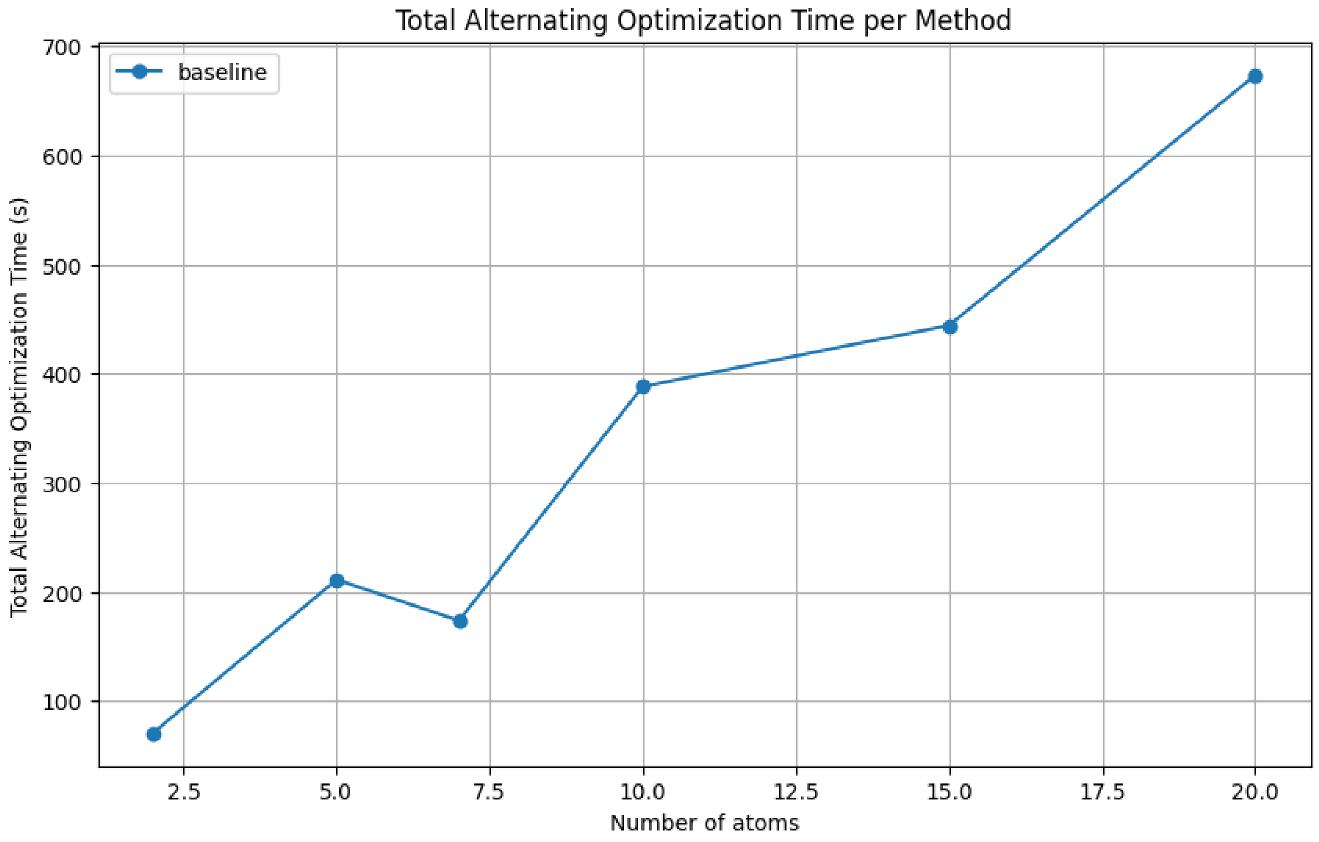


Figure 34: Computation time using the baseline.

Figure 34 illustrates that the computation time for optimizing the dictionary and activations increases with the number of atoms, as expected.

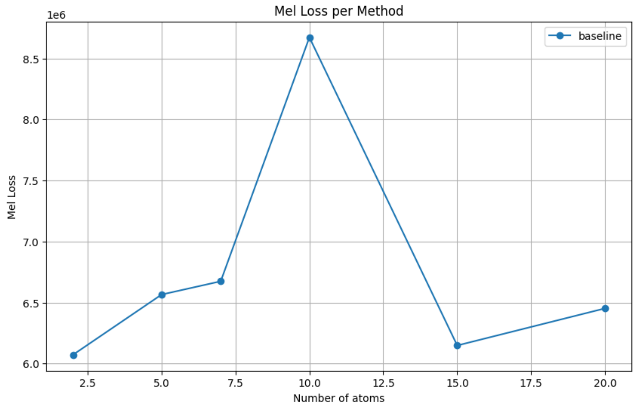


Figure 35: Variation of the MEL loss metric as the number of atoms increases.

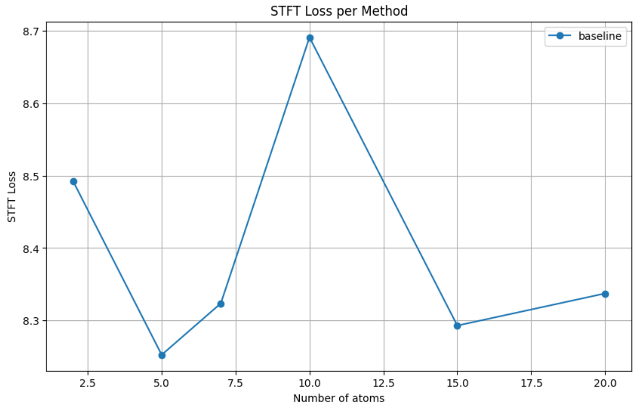


Figure 36: Variation of the STFT loss metric as the number of atoms increases.

Figures 35 and 36 show the evolution of the MEL and STFT losses with the number of atoms. Both exhibit a similar overall structure, with a peak around 10 atoms. However, the STFT loss first decreases to a global minimum at 5 atoms.

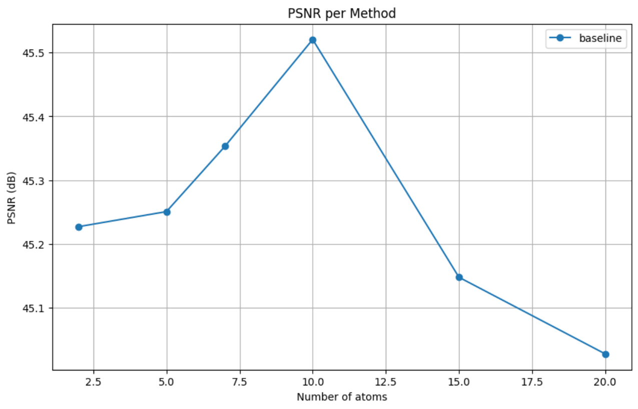


Figure 37: Variation of the PSNR metric as the number of atoms increases.

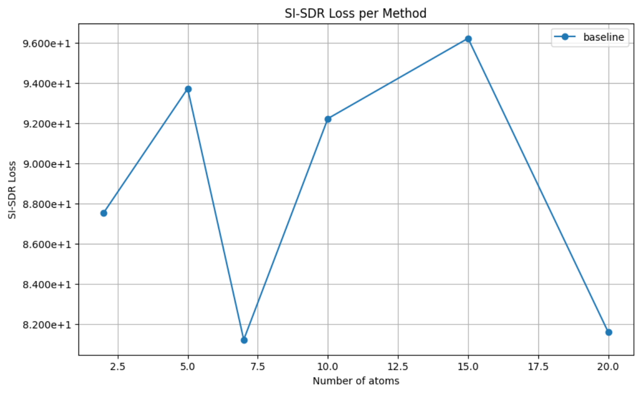


Figure 38: Variation of the SI-SDR metric as the number of atoms increases.

Finally, Figures 37 and 38 indicate that the PSNR reaches a global maximum at 10 atoms before decreasing, while the SI-SDR first increases to a local maximum at 5 atoms, then falls to a global minimum at 7 atoms, and finally rises to a global maximum at 15 atoms. Thus, with the baseline, the optimal number of atoms could be either 5 or 15. However, considering the computation time, $n_{\text{atoms}} = 5$ is preferable.
For our optimized methods, the optimal value was also $n_{\text{atoms}} = 5$. Comparing both approaches, our methods achieved a MEL loss of approximately 1.1 and an STFT loss of 7.7, while the baseline recorded 6.5 and 8.22, respectively, indicating that our methods perform better on frequency-related metrics. Regarding the PSNR and SI-SDR, our methods yielded values of 45.65 and 114, compared to 45.52 and 94 for the baseline, demonstrating superior performance on waveform-related metrics. Finally, for $n_{\text{atoms}} = 5$, our methods required around 250 seconds, while the baseline took approximately 210 seconds, making the baseline slightly more time-efficient.
In conclusion, although our methods reproduce the target signals more accurately, they require a little longer optimization time.

5.3.5 Varying the Number of Alternations

The final benchmark is performed by varying the number of alternations in the optimization of the dictionary D and the activations Z. For instance, if we perform 2 alternations, it means that we optimize D for a fixed Z, then Z for the fixed D, and repeat this sequence once more. We set $T_d = 350$ and the number of atoms to 5, as these were the best values found in the previous benchmark. The default parameters are:

- $T_d = 350$
- $T_z = 266240$
- Z sparsity: 97%
- Number of atoms: 5

The benchmark evaluates three methods: Conjugate Gradient (CG), Biconjugate Gradient Stabilized Method (BICGSTAB), and Generalized Minimal Residual Method (GMRES). Although we initially retained only CG and BICGSTAB, GMRES was added to assess potential improvements, despite a significant increase in computation time. The number of alternations is varied between 2 and 20.

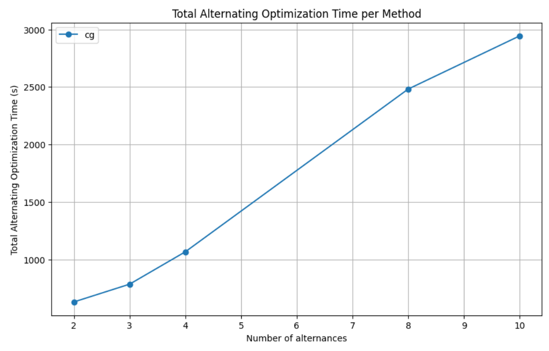


Figure 39: Computation time using optimized solvers.

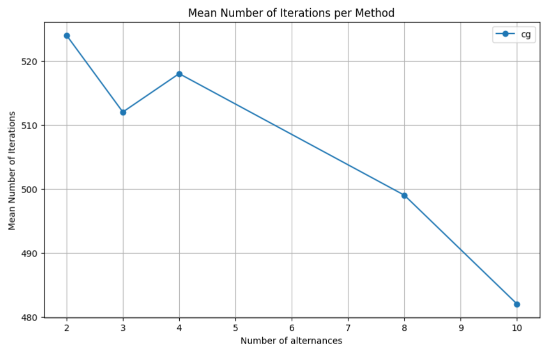


Figure 40: Number of iterations required for convergence.

As shown in Figure 39, as the number of alternations increases, the computation time grows almost linearly, an expected behavior since each alternation adds a full optimization step for both Z and D. Meanwhile, the number of iterations required to optimize the dictionary D decreases. This can be explained by the progressive refinement of Z, which becomes more structured and better adapted to X, leading to a better-conditioned optimization problem for D and thus requiring fewer iterations for convergence.

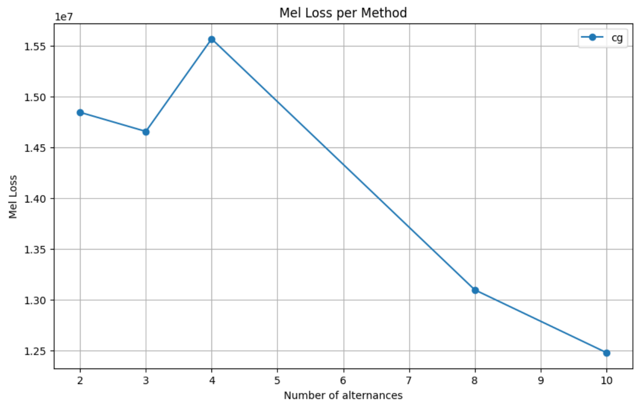


Figure 41: Variation of the MEL loss metric as the number of alternations increases.

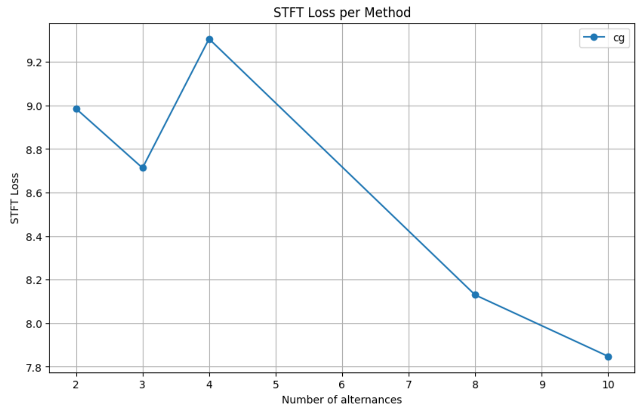


Figure 42: Variation of the STFT loss metric as the number of alternations increases.

Figure 41 shows that the MEL loss decreases, reaching a local minimum at 3 alternations, then increases at 4 alternations to hit a global maximum before decreasing again. The STFT loss (Figure 42) exhibits a similar pattern.

For the PSNR (see Figure 43), we observe a monotonically decreasing trend as the number of alternations increases. In contrast, the SI-SDR (Figure 44) shows a peak at a global maximum for 3 alternations and a second peak for 8 alternations.

Since our goal is to achieve high reconstruction quality with a short computation time, we opt for 3 alternations as the best compromise.

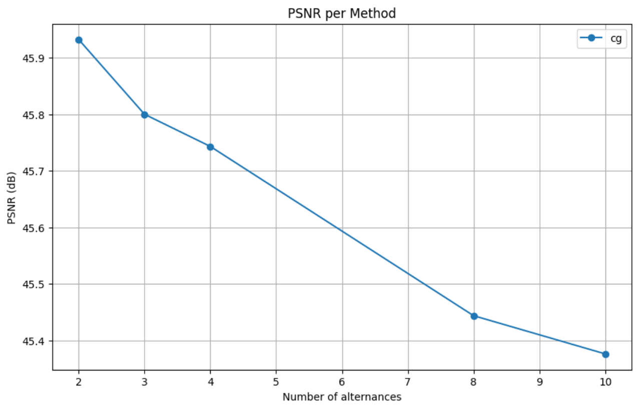


Figure 43: Variation of the PSNR metric as the number of alternations increases.

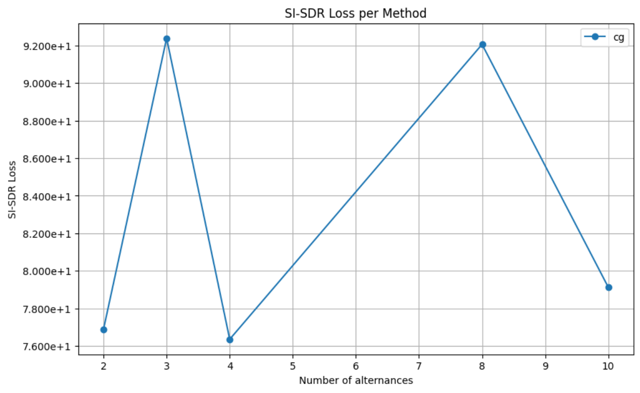


Figure 44: Variation of the SI-SDR metric as the number of alternations increases.

5.3.6 Comparison with the Baseline

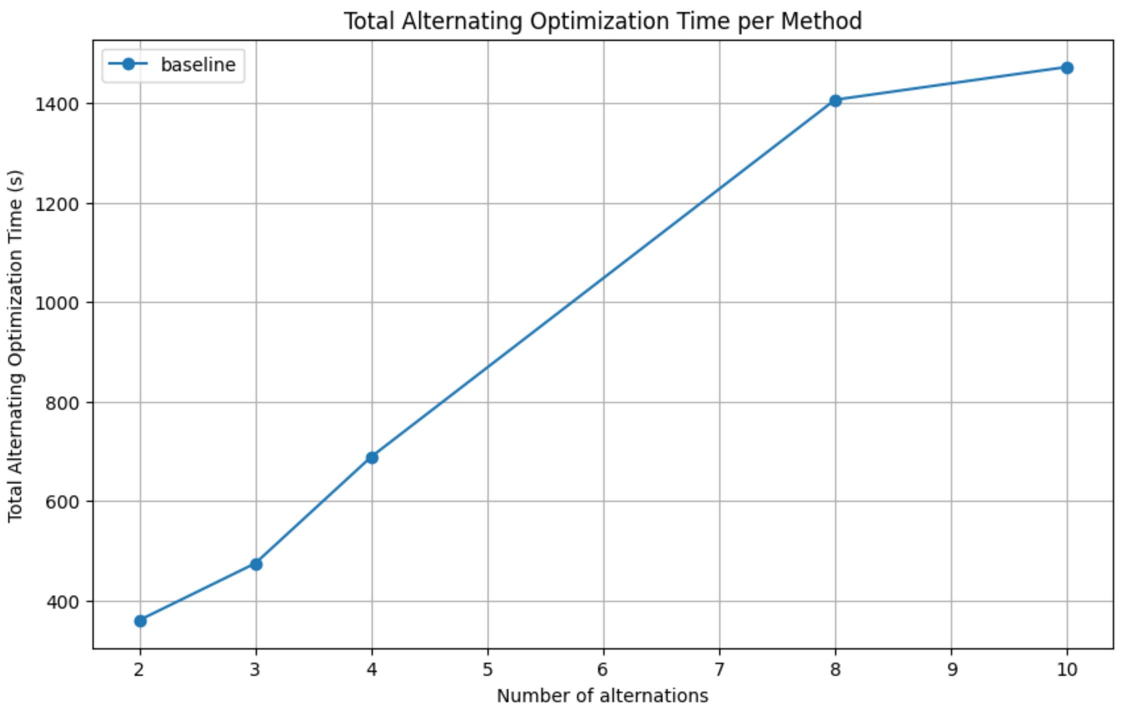


Figure 45: Computation time using the baseline.

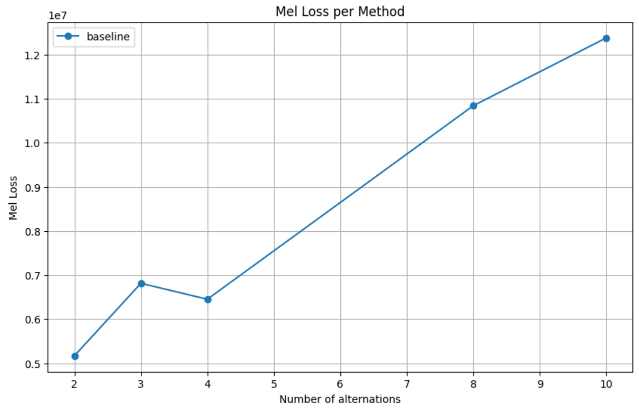


Figure 46: Variation of the MEL loss metric as the number of alternations increases.

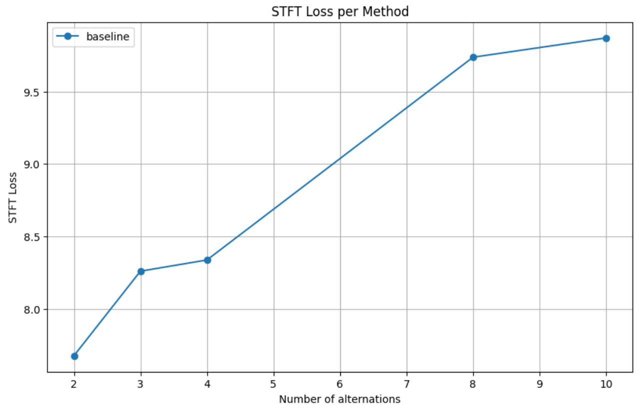


Figure 47: Variation of the STFT loss metric as the number of alternations increases.

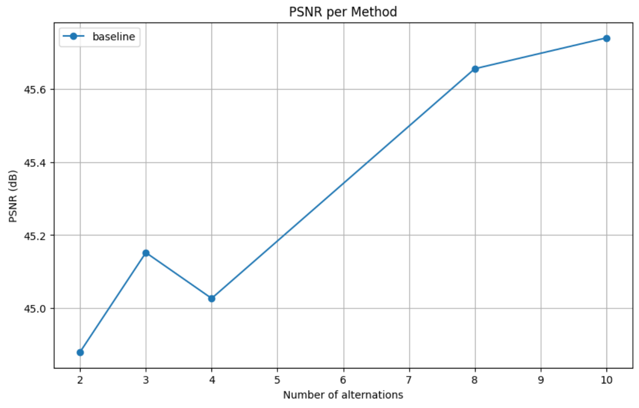


Figure 48: Variation of the PSNR metric as the number of alternations increases.

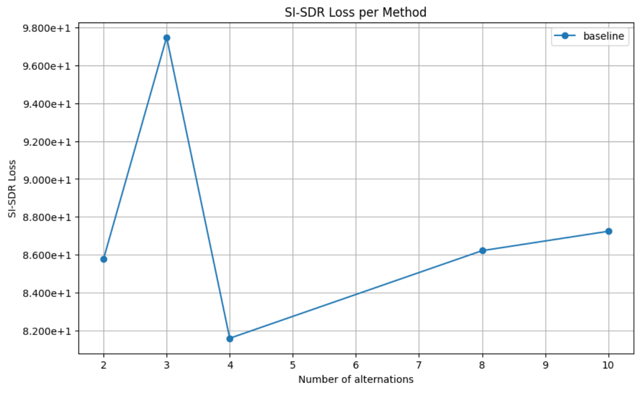


Figure 49: Variation of the SI-SDR metric as the number of alternations increases.

Similarly to our optimized methods, the baseline also shows an optimal number of alternations at 3. The baseline achieves a MEL loss of approximately 6.8×10^6, compared to 14.5×10^6 for our method. The STFT loss is about 8.25 for the baseline versus roughly 8.7 for our methods. The PSNR is around 45.15 for the baseline, compared to about 45.8 for our method, while the SI-SDR is approximately 97 for the baseline and 92 for our method. In conclusion, the results of these benchmarks are relatively similar, though the baseline offers slightly better overall performance.

5.4 Denoising: CDL in Real-World Applications

In this subsection, we present a practical application of our convolutional dictionary learning (CDL) framework for denoising. We consider three audio signals, each 5 seconds long and sampled at $f_s = 44.1$ kHz. Gaussian noise with zero mean and a variance equal to 10% of the maximum amplitude was added to each signal. The goal is to denoise these signals using CDL, as our framework is designed to model the underlying clean signal without overfitting when the hyperparameters are properly tuned.
The following results were obtained from the reconstruction process:

- Total alternating optimization time for 3 iterations: 344.5993 s
- Activation matrix sparsity: 97%

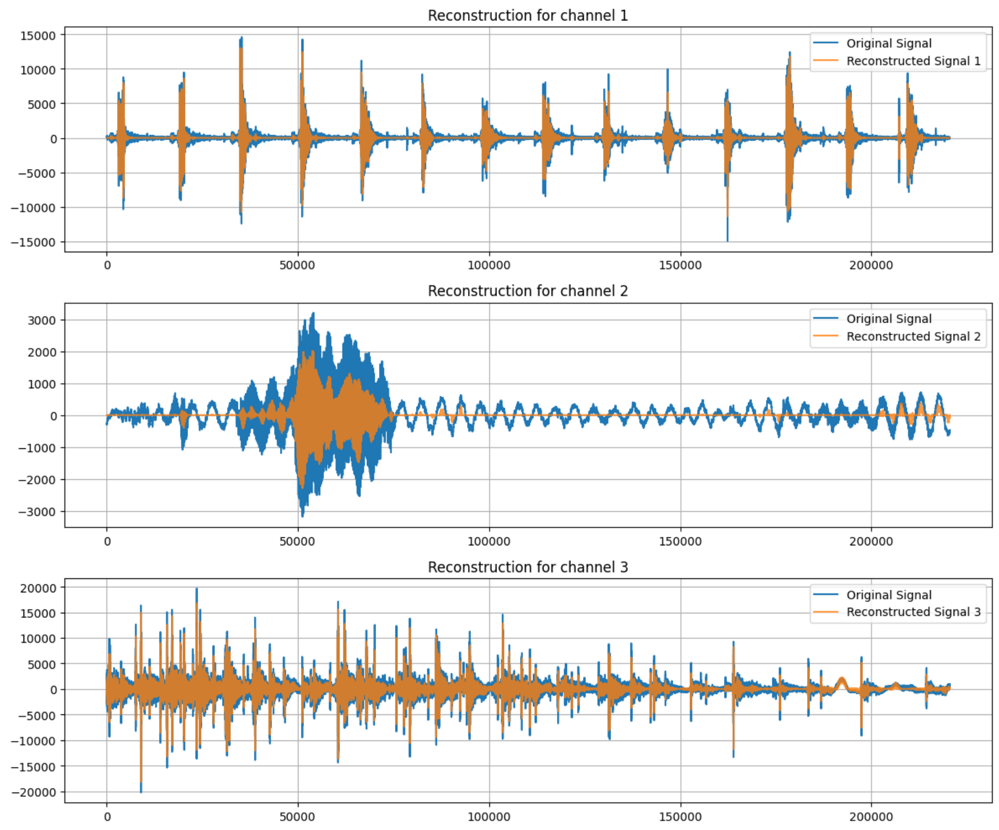


Figure 50: Reconstruction, Denoising and compression of 3 audio signals of 5 seconds each

The error metrics computed between the original signal and both the noisy and reconstructed signals are summarized in Table 1.

Table 1: Error Metrics: Original vs Noisy and Original vs Reconstructed Signals

Metric	**Original vs Noisy**	**Original vs Reconstructed**
RMSE	56.7016	53.2532
PSNR (dB)	51.06	51.61
STFT Loss	5.9805	7.2525
Mel Loss	255323.2656	3.15×10^6
SI-SDR Loss	96.9992	105.3843

Download the following audio files to compare the original, noisy, and reconstructed signals:

Original Audio Files

- Original Audio 1: Applause
- Original Audio 2: Goat Sounds
- Original Audio 3: Water Droplets

Noisy Audio Files

- Noisy Audio 1: Applause
- Noisy Audio 2: Goat Sounds

- Noisy Audio 3: Water Droplets

Denoised Audio Files (Reconstructed using CDL)

- Reconstructed Audio 1: Applause
- Reconstructed Audio 2: Goat Sounds
- Reconstructed Audio 3: Water Droplets

The global distortion metrics (RMSE, PSNR, SI-SDR) show an improvement in the reconstructed signal compared to the noisy signal, indicating effective noise reduction and a closer match to the original signal's overall shape. However, the frequency-based losses (STFT and Mel) are higher for the reconstructed signal, suggesting a loss of spectral richness and finer time-frequency details. This indicates that while the CDL framework successfully reduces noise and preserves the general structure of the signal, it may sacrifice some frequency detail during the reconstruction process.

6 Conclusion

In this work, we presented an accelerated convolutional dictionary learning framework that integrates iterative solvers with memory-efficient techniques and compiler optimizations. Our experiments demonstrated that our optimized methods can achieve competitive reconstruction quality and high compression ratios, particularly in the context of audio denoising on the ESC-50 dataset. Although our methods often rival the baseline in terms of reconstruction accuracy, the baseline still exhibits some advantages in computational efficiency.

Nonetheless, there remains significant room for improvement. Future work should aim to further enhance the performance of our methods by exploring state-of-the-art optimization techniques reported in recent literature. Moreover, an innovative idea worth investigating is the reformulation of convolutions in alternative domains. For example, by moving to the frequency domain, where convolution translates into a simple multiplication, it may be possible to significantly reduce computational time while preserving or even enhancing reconstruction fidelity.

In summary, while our study establishes a robust and scalable CDL framework, it also highlights directions for future research. By combining experimental validation, integrating advanced algorithms and alternative strategies, we lay a foundation for further advancements in efficient, real-time convolutional dictionary learning.

7 Code availability

The code is available in this GitHub repository : Click here to access the repository.

Glossary

Activation Matrix (Z) The matrix containing coefficients that indicate the contribution of each atom in reconstructing the signal.. 36

ADMM (Alternating Direction Method of Multipliers) An optimization algorithm for convex problems, decomposing objectives into simpler subproblems.. 5, 36

Atom A basic element or feature function in the dictionary that captures recurring patterns in the data.. 36

BiCGStab (Biconjugate Gradient Stabilized) An iterative solver for non-symmetric linear systems that improves convergence stability.. 1, 5, 36

CDL (Convolutional Dictionary Learning) A method for decomposing signals into sparse combinations of temporal patterns (atoms) using convolutions.. 1, 36

CG (Conjugate Gradient) An iterative algorithm for solving symmetric positive-definite linear systems by minimizing residuals in conjugate directions.. 1, 36

Correlation A measure of similarity between two signals as a function of time lag, often used as the transpose of the convolution operation.. 36

Dictionary (D) The set of atoms used to reconstruct a signal.. 36

FISTA (Fast Iterative Shrinkage-Thresholding Algorithm) An accelerated variant of ISTA for solving sparse optimization problems.. 36

GMRES (Generalized Minimal Residual) An iterative solver for non-symmetric linear systems that minimizes the residual over a Krylov subspace.. 1, 36

ISTA (Iterative Shrinkage Thresholding Algorithm) An algorithm for solving ℓ_1-regularized optimization problems using soft thresholding.. 11, 36

Iterative Solvers (e.g., CG, GMRES, BiCGStab) Algorithms that solve linear systems iteratively, avoiding explicit matrix inversion.. 36

JIT (Just-In-Time Compilation) A technique for dynamic code compilation during runtime to accelerate computations (e.g., using Numba).. 1, 17, 36

LinearOperator A class (e.g., in SciPy) that represents a matrix by its action on a vector, without requiring the full matrix to be stored in memory.. 36

MEL Loss A metric comparing Mel-scaled spectrograms that reflects human auditory perception.. 36

PSNR (Peak Signal-to-Noise Ratio) A measure of the fidelity of a reconstructed signal relative to the original, expressed in decibels.. 36

SI-SDR (Scale-Invariant Signal-to-Distortion Ratio) A metric assessing the quality of signal reconstruction by normalizing the amplitude of the signal.. 36

STFT (Short-Time Fourier Transform) A localized Fourier transform used to analyze the frequency content of non-stationary signals.. 36

References

[1] Cristina Garcia-Cardona and Brendt Wohlberg. *Convolutional Dictionary Learning: A Comparative Review and New Algorithms.*

[2] L. Oudre (2021). *Machine Learning and Artificial Intelligence for signals and time series.*, ENS Paris Saclay, University Paris Saclay. `https://github.com/oudre/parcoursIA`

[3] Dataset des sources audio - ESC50. `https://github.com/karolpiczak/ESC-50/tree/master/audio`

[4] Andrew H. Song, Student member, IEEE, Bahareh Tolooshams, Student member, IEEE, and Demba Ba, Member, IEEE. *Gaussian Process Convolutional Dictionary Learning.*

[5] Gustavo Silva et al. *Efficient Algorithm for Convolutional Dictionary Learning via Accelerated Proximal Gradient Consensus.*

[6] Yanis Gomes, Charles Truong et al. *Convolutional Sparse Coding with Multipath Orthogonal Matching Pursuit.*

[7] Jonathan Le Roux et al. *SDR – HALF-BAKED OR WELL DONE?*

[8] Benoît Giniès et al. *Using Random Codebooks for Audio Neural AutoEncoders*