Efficient Implementations for Orthogonal Matching Pursuit

Hufei Zhu, Wen Chen and Yanpeng Wu

1 Department of Information Science and Engineering, Hunan First Normal University, Changsha 410205, China; hufezhu93@hotmail.com
2 College of Computer Science and Software, Shenzhen University, Shenzhen 518060, China
3 Department of Electronic Engineering, Shanghai Jiao Tong University, Shanghai 200240, China; wenchen@sjtu.edu.cn
* Correspondence: yanpengwu17@163.com; Tel.: +86-0731-8822-8167

Received: 26 July 2020; Accepted: 9 September 2020; Published: 14 September 2020

Abstract: Based on the efficient inverse Cholesky factorization, we propose an implementation of OMP (called as version 0, i.e., v0) and its four memory-saving versions (i.e., the proposed v1, v2, v3 and v4). In the simulations, the proposed five versions and the existing OMP implementations have nearly the same numerical errors. Among all the OMP implementations, the proposed v0 needs the least computational complexity, and is the fastest in the simulations for almost all problem sizes. As a tradeoff between computational complexities/time and memory requirements, the proposed v1 seems to be better than all the existing ones when only considering the efficient OMP implementations storing G (i.e., the Gram matrix of the dictionary), the proposed v2 and v3 seem to be better than the only existing one when only considering the efficient implementations not storing G, and the proposed v4 seems to be better than the naive implementation that has the (known) minimum memory requirements. Moreover, all the proposed five versions only include parallelizable matrix-vector products in each iteration, and do not need any back-substitutions that are necessary in some existing efficient implementations (e.g., those utilizing the Cholesky factorization).

Keywords: inverse cholesky factorization; orthogonal matching pursuit (OMP); efficient implementation

1. Introduction

A signal vector $x \in \mathbb{C}^N$ is said to be $K$-sparse if $x$ includes no more than $K$ none-zero entries. The measurements of the signal $x$ can be collected in a data vector $u \in \mathbb{C}^M$ satisfying

$$u = \Phi x + n,$$

where $\Phi$ is an $M \times N$ dictionary with $K < M \ll N$, and $n$ is noise. The problem of signal recovery, i.e., solving (1) to estimate $x$, is sparse approximation or representation [1–6].

Recently, the OMP algorithm has been applied in various scenarios [7–13]. The stagewise OMP algorithm was proposed in [7] to enhance the convergence speed for switching signal reconstruction in IGBT (insulated gate bipolar transistor) online condition monitoring, while the SiT based stagewise OMP was proposed in [8] for the estimation of massive-MIMO sparse uplink channels. A modified OMP algorithm was utilized in [9] to propose compressed sensing ISAR (inverse synthetic aperture radar) imaging with highly maneuvering motion. The OMP algorithm was used in [10] for AoD (angle of departure) estimation at the mobile station, applied in [11] for channel estimation and equalization on the basis of the Pseudo-Noise (PN) sequences, and was chosen in [12] as the imaging algorithm to reconstruct imaging scenes in the single-pixel camera, since it contains the stable and
efficient operation in reconstructing small data. Moreover, simultaneous OMP [14] was utilized in [13] to convert image patches into sparse coefficients for representing the source multi-focus images.

The computational complexity of OMP can be reduced by utilizing the Cholesky factorization [15–17], the QR factorization [18–20], or the Matrix Inversion Lemma (MIL) [20–23] that is also called as Matrix Inversion Bypass (MIB). Most of the efficient implementations of OMP in [15–23] have been compared in [24]. Hardware-efficient architectures of OMP based on Cholesky factorization, QR factorization and MIL have been reported in [25–34], respectively. Moreover, to implement simultaneous OMP [14] efficiently, a novel partitioned inversion method was proposed in [35].

Based on the efficient inverse Cholesky factorization proposed in [36], we propose an efficient OMP implementation and its four memory-saving versions (This work was presented partly in [37] at IEEE Vehicular Technology Conference (VTC) 2013 fall. The new contribution in this manuscript includes the memory-saving versions 1 and 4, the derivation of the equations in the proposed OMP implementations, the more detailed introduction of the existing OMP implementations, and the more detailed comparison among the presented OMP implementations), which are then compared with the existing ones. To focus on efficient implementations of OMP, we do not consider the extended, variant, or approximate implementations of OMP, such as the Stagewise OMP (StOMP) [38], the Backtracking-based Adaptive OMP (BAOMP) [39], the gradient pursuit [40], or the cyclic matching pursuit [41].

This paper is organized as follows. Section 2 describes the existing implementations for OMP. In Section 3, we propose an efficient OMP implementation and its four memory-saving versions. Then in Section 4, we evaluate the presented implementations for OMP. Finally, we make conclusion in Section 5.

In what follows, (·)T, (·)∗, and (·)H denote the matrix transposition, the matrix conjugate, and the Hermitian matrix, respectively. We use 0M, IM, and ∅ to denote the length-M zero column vector, the M × M identity matrix and the empty matrix (or vector), respectively, and we use ∥·∥ to denote the square norm (i.e., the L2 norm) of a vector.

2. The Existing Implementations for OMP

In this section, we summarize the existing OMP implementations.

2.1. The Naive Implementation of OMP

The dictionary Φ in (1) can be written as

\[ \Phi = \left( \varphi_1 \quad \varphi_2 \quad \cdots \quad \varphi_N \right), \]

where \( \varphi_j \in \mathbb{C}^M \) denotes the i-th column of \( \Phi \), and is called as an atom. The signal recovery problem is to find the sparse solution to the underdetermined linear system (1), i.e., to represent \( u \) as a weighted sum of the least number of atoms. OMP is a greedy algorithm that selects atoms from \( \Phi \) iteratively to approximate \( u \) gradually. In each iteration, OMP selects the atom best matching the current residual (i.e., the approximation error), renews the weights for all the already selected atoms to obtain the least squares approximation of \( u \), and then updates the residual accordingly.

Algorithm 1 summarizes the naive implementation of OMP [20,24], which is called Naive in this paper, as in [24]. As the OMP implementation described in ([20] (left lines 27–34 on page 2)), we also follow these conventions: \( \Lambda_k \) is a set containing the indices of the selected \( k \) atoms, and \( \Phi_{\Lambda_k} \) is a sub-matrix of \( \Phi \) containing only those columns of \( \Phi \) with indices in \( \Lambda_k \). The estimate \( \hat{x} \) for \( x \) has nonzero entries at the indices listed in \( \Lambda_k \), and the \( \lambda_i \)-th entry of \( \hat{x} \) equals the \( i \)-th entry of \( x_k \).
Algorithm 1 The Naive Implementation of OMP

Step 1: Initialize $\Lambda_0 = \emptyset$, the residual $r^0 = u$, and the iteration counter $k = 1$.

Step 2: Projection: compute the inner products

$$p^{k-1} = \Phi^H r^{k-1}. \quad (3)$$

Step 3: Decide $i^k = \arg\max_{i=1,2,\ldots,N} \left( \frac{|p^{k-1}_i|}{\|\varphi_i\|} \right)$, where $p^{k-1}_i$ is the $i$-th entry of $p^{k-1}$.

Let $\Lambda_k = \Lambda_{k-1} \cup \{i^k\}$, i.e., $\lambda_k \triangleq i^k$ is the $k$-th entry of the set $\Lambda_k$.

Step 4: Renew the weights for the selected $k$ atoms by

$$\hat{x}_k = \left( \Phi_{\Lambda_k}^H \Phi_{\Lambda_k} \right)^{-1} \Phi_{\Lambda_k}^H u, \quad (4)$$

and then utilize $\hat{x}_k$ to update the residual by

$$r^k = u - \Phi_{\Lambda_k} \hat{x}_k. \quad (5)$$

Step 5: $k := k + 1$. Then return to Step 2 if $k < K$ and the residual energy $\|r^k\|^2$.

$$\|r^k\|^2 > \xi \|u\|^2, \quad (6)$$

where $\xi$ is a small positive real number, e.g., $\xi = 10^{-10}$.

Step 6: Output: $r^K$, $\Lambda_K$ and $\hat{x}_K$, where $K' \leq K$.

2.2. The Existing Efficient Implementations of OMP

To implement OMP efficiently, the Cholesky factorization has been applied in [15–17,25–28], the QR factorization has been applied in [18–20,29–31], and the MIL (i.e., Matrix Inversion Lemma) has been applied in [20–23,32–34]. In [24], the authors summarized the existing various implementations of OMP in [15–23], studied their numerical and computational performances, and empirically compared their computational time. According to the conclusions in [24], the accumulation of error is insignificant in each existing OMP implementation. On the other hand, any of the existing implementations compared in [24] can be the fastest for some specific problem sizes, while the efficient OMP implementation by the QR factorization [19] appears to be the fastest for particularly large problem sizes.

In the rest of this subsection, we introduce the existing efficient implementations of OMP, including two implementations by the Cholesky factorization [15–17] and two implementations by the QR factorization [19], which are called as Chol-1, Chol-2, QR-1 and QR-2, respectively, as in [24]. Moreover, we also include two implementations by the MIL (i.e., Matrix Inversion Lemma) discussed in [21,24], i.e., MIL-1 and MIL-2. Since the literatures [25–34] mainly deal with the hardware architectures for the existing OMP implementations, we will not introduce the OMP implementations in [25–34] in the rest of this subsection.

For simplicity, we assume that the dictionary $\Phi$ has normalized columns, i.e., $\|\varphi_i\|^2 = 1$, as in [18] and the shared simulation code [42] of [24]. We omit step 6 to only focus on the $k$-th iteration, and we omit steps 1, 3 and 5, since they are the same as those in Algorithm 1 except that now it is required to initialize $\|r^0\|^2 = \|u\|^2$ in step 1. Thus we only introduce steps 2 and 4, which update the projection $p^{k-1}$ and the residual energy $\|r^k\|^2$, respectively.

Many existing efficient OMP implementations utilize the Gram matrix [20] of all the $N$ atoms $\varphi_1, \varphi_2, \ldots, \varphi_N$, i.e.,

$$G = \Phi^H \Phi. \quad (7)$$
The Gram matrix of the $k$ atoms selected in iteration $k$ is

$$G_{\Lambda_k} = \Phi_{\Lambda_k}^H \Phi_{\Lambda_k},$$  \hspace{1cm} \text{(8)}$$

where $\Phi_{\Lambda_k}$ can be written as

$$\Phi_{\Lambda_k} = \begin{bmatrix} \Phi_{\Lambda_{k-1}} & \varphi_{\lambda_k} \end{bmatrix}.$$

\hspace{1cm} \text{(9)}$$

In Algorithm 2 we introduce Chol-1 and Chol-2, which utilize the Cholesky factor [43] of $G_{\Lambda_k}$, i.e., $V_k$ satisfying

$$V_k V_k^H = G_{\Lambda_k}.$$  \hspace{1cm} \text{(10)}$$

We also need

$$s_{k-1} = \Phi_{\Lambda_{k-1}}^H \varphi_{\lambda_k},$$

\hspace{1cm} \text{(11)}$$

which is obtained in $G_{\Lambda_k}$ by

$$G_{\Lambda_k} = \begin{bmatrix} G_{\Lambda_{k-1}} & s_{k-1} \\ s_{k-1}^H & g_{\lambda_k} \lambda_k \end{bmatrix}. $$

\hspace{1cm} \text{(12)}$$

To verify (12) and (11), substitute (9) into (8) to deduce

$$G_{\Lambda_k} = \begin{bmatrix} \Phi_{\Lambda_{k-1}}^H \Phi_{\Lambda_{k-1}} & \Phi_{\Lambda_{k-1}}^H \varphi_{\lambda_k} \\ \varphi_{\lambda_k}^H \Phi_{\Lambda_{k-1}} & \varphi_{\lambda_k}^H \varphi_{\lambda_k} \end{bmatrix}. $$

\hspace{1cm} \text{(13)}$$

In Algorithm 3 we introduce QR-1 and QR-2, which utilize the QR factors [19] of $\Phi_{\Lambda_k}$, i.e., $Q_k$ and $R_k$ satisfying

$$\Phi_{\Lambda_k} = Q_k R_k.$$  \hspace{1cm} \text{(14)}$$

In Algorithm 4 we introduce MIL-1 and MIL-2. In each iteration, MIL-1 updates the biorthogonal basis of $\Phi_{\Lambda_k}$, i.e.,

$$\Psi_{\Lambda_k} = \Phi_{\Lambda_k} (\Phi_{\Lambda_k}^H \Phi_{\Lambda_k})^{-1}$$

\hspace{1cm} \text{(15)}$$

satisfying

$$\Psi_{\Lambda_k} H \Phi_{\Lambda_k} = \Phi_{\Lambda_k}^H \Psi_{\Lambda_k} = I,$$

\hspace{1cm} \text{(16)}$$

while MIL-2 updates $G_{\Lambda_k}^{-1}$. 
Algorithm 2 OMP by the Cholesky Decomposition

Step 2. Projection: if $k = 1$, compute $p^0$ by (3); else when $k \geq 2$, compute $\mathbf{x}_{k-1}$ by solving the triangular system

$$V^H_{k-1}\mathbf{x}_{k-1} = y_{k-1},$$

(17)

and then compute $p^{k-1}$ by one of two ways. In the first case (Chol-1), explicitly compute $r^{k-1}$ ($k \geq 2$) by (5) and then directly compute $p^{k-1}$ ($k \geq 2$) by (3); or in the second case (Chol-2), compute $p^{k-1}$ ($k \geq 2$) by

$$p^{k-1} = p^0 - G(:, \Lambda_{k-1}) \cdot \mathbf{x}_{k-1},$$

(18)

where $G(:, \Lambda_{k-1})$ includes $k - 1$ columns in $G$ and satisfies

$$G(:, \Lambda_{k-1}) = \Phi^H \Phi_{\Lambda_{k-1}}.$$

(19)

Step 4. If $k = 1$, obtain $V_1 = 1$ from (10); else when $k \geq 2$, obtain $V_k$ by

$$V_k = \begin{bmatrix} V_{k-1} & 0_{k-1} \end{bmatrix} \begin{bmatrix} \sqrt{S_{\lambda_k} - \|z_{k-1}\|^2} \end{bmatrix},$$

(20)

where $S_{\lambda_k} = \phi^H_{\lambda_k} \phi_{\lambda_k}$ is in $S$, and $z_{k-1}$ is computed by solving the triangular system

$$V_{k-1}z_{k-1} = s_{k-1} = \Phi^H_{\Lambda_{k-1}} \phi_{\lambda_k}.$$  

(21)

Then for any $k \geq 1$, compute $y_k$ by solving

$$V_k y_k = p^0_{\Lambda_k} = \Phi^H_{\Lambda_k} u,$$

(22)

where

$$p^0_{\Lambda_k} = \begin{bmatrix} p^0_{\lambda_1} & p^0_{\lambda_2} & \cdots & p^0_{\lambda_k} \end{bmatrix}^T$$

(23)

has the $i$-th entry $p^0_{\lambda_i}$ ($1 \leq i \leq k$) being the $\lambda_i$-th entry in $p^0$, and finally compute

$$\|r^k\|^2 = \|u\|^2 - \|y_k\|^2.$$
Algorithm 3 OMP by the QR Decomposition

Step 2. Projection: if $k = 1$, compute $p^0$ by (3); else compute $p^{k-1}$ ($k \geq 2$) by one of two ways. In the first case (QR-1), compute $p^{k-1}$ ($k \geq 2$) by

$$p^{k-1} = p^{k-2} - (q_{k-1}^H u)\Phi^H q_{k-1}$$  \hspace{1cm} (24)

where $q_{k-1}$ denotes the $(k-1)$-th column in $Q_{k-1}$; or in the second case (QR-2), compute $y_{k-1}$ ($k \geq 2$) by solving the triangular system

$$R_{k-1}^H y_{k-1} = p^0 = \Phi^H \Lambda_{k-1}$$  \hspace{1cm} (25)

compute $\hat{x}_{k-1}$ ($k \geq 2$) by solving

$$R_{k-1} \hat{x}_{k-1} = y_{k-1}$$  \hspace{1cm} (26)

and then compute $p^{k-1}$ ($k \geq 2$) by (18).

Step 4. If $k = 1$, obtain $Q_1$ and $R_1$ from (14); else when $k \geq 2$, compute

$$w_{k-1} = Q_{k-1}^H \varphi \lambda_k$$  \hspace{1cm} (27)

and

$$\beta_k = \sqrt{g_{\lambda_k} - ||w_{k-1}||^2}$$  \hspace{1cm} (28)

to obtain

$$Q_k = \begin{bmatrix} Q_{k-1} & (\varphi \lambda_k - Q_{k-1} w_{k-1})/\beta_k \end{bmatrix}$$  \hspace{1cm} (29)

and

$$R_k = \begin{bmatrix} R_{k-1} & w_{k-1} \\ 0 & \beta_k \end{bmatrix}$$  \hspace{1cm} (30)

and finally compute $||r_k||^2 = ||r^{k-1}||^2 - |q_k^H u|^2$. 

Algorithm 4 OMP by the Matrix Inversion Lemma

Step 2. Projection: if \( k = 1 \), compute \( p^0 \) by (3); else compute

\[
p^{k-1} = p^{k-2} + \delta_{k-1} G(:, \Lambda_{k-1}) \begin{bmatrix} h^T_{k-2} & -1 \end{bmatrix},
\]  

(31)

where \( k \geq 2 \), and \( h_0 = \emptyset \) when \( k = 2 \).

Step 4. If \( k = 1 \), compute \( \varepsilon_1 = 1 / g_{\lambda_1} \lambda_1 \) and \( \delta_1 = \varepsilon_1 p^0_{\lambda_1} \), and obtain \( \Psi_{\Lambda_1} \) by (15) for MIL-1 or \( G^{-1}_{\Lambda_1} = 1 / G_{\Lambda_1} \) for MIL-2; else compute the intermediate results by one of two ways. In the first case (MIL-1) when \( k \geq 2 \), compute

\[
\psi_{k-1} = \varepsilon_{k-1} (\varphi_{\lambda_{k-1}} - \psi^{k-1})
\]  

(32)

to obtain

\[
\Psi_{\Lambda_{k-1}} = \begin{bmatrix} \Psi_{\Lambda_{k-2}} - \psi_{k-1} h^H_{k-2} & \psi_{k-1} \end{bmatrix}
\]  

(33)

compute

\[
h_{k-1} = \Psi^H_{\Lambda_{k-1}} \varphi_{\lambda_k},
\]  

(34)

\[
\psi^k = \Phi_{\Lambda_{k-1}} h_{k-1},
\]  

(35)

\[
\varepsilon_k = 1 / \left( g_{\lambda_k} \lambda_k - \|\psi^k\|^2 \right)
\]  

(36)

and

\[
\delta_k = \varepsilon_k \left( p^0_{\lambda_k} - h^H_{k-1} p^0_{\lambda_{k-1}} \right);
\]  

(37)

or in the second case (MIL-2) when \( k \geq 2 \), compute

\[
G^{-1}_{\Lambda_{k-1}} = \begin{bmatrix}
G^{-1}_{\Lambda_{k-2}} + \varepsilon_{k-1} h_{k-2} h^H_{k-2} & -\varepsilon_{k-1} h_{k-2} \\
-\varepsilon_{k-1} h^H_{k-2} & \varepsilon_{k-1}
\end{bmatrix},
\]  

(38)

obtain \( s_{k-1} \) in \( G_{\Lambda_k} \) by (12) to compute

\[
h_{k-1} = G^{-1}_{\Lambda_{k-1}} s_{k-1}
\]  

(39)

and

\[
\varepsilon_k = 1 / \left( g_{\lambda_k} \lambda_k - s^H_{k-1} h_{k-1} \right),
\]  

(40)

and compute \( \delta_k \) by (37). Finally compute

\[
\| r^k \|^2 = \| r^{k-1} \|^2 + |\delta_k|^2 / \varepsilon_k - 2 \text{Re}\{\delta_k p^{k-1}_{\lambda_k}\}.
\]  

(41)

3. The Proposed Efficient OMP Implementation and Its Memory-Saving Versions

In this section, we propose an efficient OMP implementation, which is based on the efficient inverse Cholesky factorization proposed in [36]. We also introduce four memory-saving versions for the proposed OMP implementation.
3.1. The Proposed OMP Implementation by the Inverse Cholesky Factorization

The inverse Cholesky factor \[36\] of \( G_{\Lambda_k} \) can be denoted as \( F_k \) satisfying
\[
F_k F_k^H = G_{\Lambda_k}^{-1} = \left( \Phi^H_{\Lambda_k} \Phi_{\Lambda_k} \right)^{-1},
\]
while from (10) we can deduce
\[
V_k^{-H} V_k^{-1} = G_{\Lambda_k}^{-1}.
\]
Then we can compare (43) with (42) to deduce
\[
F_k = V_k^{-H},
\]
since \( V_k \) is the unique lower-triangular Cholesky factor \[43\] of \( G_{\Lambda_k} \), and \( F_k \), the upper-triangular equivalent Cholesky factor of \( G_{\Lambda_k}^{-1} \) \[36\], is also unique \[43\].

In \[36\], \( F_k \) is computed from \( F_{k-1} \) iteratively by
\[
F_k = \begin{bmatrix} F_{k-1} & t_{k-1} \\ 0_{k-1} & \gamma_k \end{bmatrix},
\]
where \( \gamma_k \) and \( t_{k-1} \) is computed by (36) (Equation (24))
\[
\gamma_k = 1/\sqrt{g_{\lambda_k^2} - c_{k-1}^H c_{k-1}},
\]
\[
t_{k-1} = -\gamma_k F_{k-1} c_{k-1},
\]
and
\[
c_{k-1} = F_{k-1}^H s_{k-1}.
\]
Notice that \( s_{k-1} \) in (48) and \( g_{\lambda_k^2} \) in (46) are the vector and scalar in \( G_{\Lambda_k} \), respectively, as shown in (12).

To utilize \( F_k \) in OMP, we substitute (42) into (4) to deduce
\[
\hat{x}_k = F_k F_k^H \Phi^H_{\Lambda_k} u.
\]
Then we substitute (49) into (5) to obtain \( r^k = u - \Phi_{\Lambda_k} F_k F_k^H \Phi^H_{\Lambda_k} u \), which is substituted into (3) to obtain
\[
p^{k-1} = \Phi^H u - \Phi^H \Phi_{\Lambda_k} F_k F_k^H \Phi^H_{\Lambda_k} u.
\]
Equation (50) can be written as
\[
p^{k-1} = \Phi^H u - B_{k-1} a_{k-1},
\]
where
\[
a_k = F_k^H \Phi^H_{\Lambda_k} u = E_k^H u,
\]
\[
B_k = \Phi^H \Phi_{\Lambda_k} F_k = \Phi^H E_k,
\]
and
\[
E_k = \Phi_{\Lambda_k} F_k.
\]
Let us deduce
\[
F_1 = \sqrt{G_{\Lambda_1}^{-1}} = \sqrt{g_{\lambda_1^2}} = \sqrt{ \left( \Phi^H_{\Lambda_1} \Phi_{\Lambda_1} \right)^{-1}}
\]
(55)
from (42), and substitute (47) into (45) to obtain

$$F_k = \begin{bmatrix} F_{k-1} & -\gamma_k F_{k-1} c_{k-1} \\ 0 & \gamma_k \end{bmatrix}.$$  

(56)

Then we utilize the above-described intermediate results $a_k$ and $B_k$, to propose an efficient OMP implementation (called as version 0, i.e., v0) in the following Algorithm 5, where steps 1, 3 and 5 are omitted as in Algorithms 2–4. In Algorithm 5, we will use (3), (46), (55), (56) and (5) successively.

**Algorithm 5** The Proposed OMP Implementation (v0) by the Inverse Cholesky Factorization

**Step 2.** Projection: if $k = 1$, compute $p_0 = \Phi^H r_0$; else when $k \geq 2$, compute $p_{k-1} (k \geq 2)$ by

$$p_{k-1} = p_{k-2} - b_{(k-1)} a_{k-1},$$  

(57)

where $b_{(k-1)}$ is the $(k-1)$-th column of $B_{k-1}$, and $a_{k-1}$ is the $(k-1)$-th entry of $a_{k-1}$.

**Step 4.** Obtain

$$c_{k-1} = \left( b_{\lambda k,1:k-1}^H \right)^H,$$  

(58)

where $b_{\lambda k,1:k-1}^H$ is the $\lambda_k$-th row of $B_{k-1}$. Then compute $\gamma_k = 1 / \sqrt{g_{\lambda_k,\lambda_k} - c_{k-1}^H c_{k-1}},$

$$a_k = \gamma_k p_{k-1},$$  

(59)

$$a_k = \begin{bmatrix} a_{k-1}^T \\ a_k \end{bmatrix},$$  

(60)

$$b_k = \gamma_k (g_{\lambda_k} - B_{k-1} c_{k-1}),$$  

(61)

$$B_k = \begin{bmatrix} B_{k-1} & b_k \end{bmatrix},$$  

(62)

where $p_{\lambda k}^{k-1}$ is the $\lambda_k$-th entry of $p^{k-1}$, $g_{\lambda k}$ is the $\lambda_k$-th column of $G$, and $c_0 = B_0 = a_0 = \emptyset$ is assumed for $k = 1$. Finally update the residual energy by

$$\| r^k \|^2 = \| r^{k-1} \|^2 - \| a_k \|^2.$$  

(63)

If $F_k$ is required to compute $\hat{x}_k$ and $\hat{r}^k$, compute $F_k$ by $F_1 = \sqrt{g_{\lambda_1,\lambda_1}}$ when $k = 1$, or by

$$F_k = \begin{bmatrix} F_{k-1} & -\gamma_k F_{k-1} c_{k-1} \\ 0 & \gamma_k \end{bmatrix}$$  

when $k \geq 2$.

**Step 6.** Output: $\| r^{K'} \|^2$ and $A_{K'}$. When $\hat{x}_{K'}$ and $r^{K'}$ are required, compute

$$\hat{x}_{K'} = F_{K'} a_{K'},$$  

(64)

and $r^{K'} = u - \Phi a_{K'},$ where $K' \leq K$.

In the next subsection, we will verify Equations (57)–(64) in Algorithm 5. Moreover, we will also verify

$$e_k = \gamma_k (\varphi_{\lambda_k} - E_{k-1} e_{k-1}),$$  

(65)

and

$$E_k = \begin{bmatrix} E_{k-1} & e_k \end{bmatrix},$$  

(66)
which can be utilized to update the intermediate result $E_k$.

3.2. To Verify Equations (57)–(66)

Here we verify the above-mentioned Equations (57)–(66), in the order of (64), (58), (65), (66), (61), (62), (59), (60), (57) and (63).

To verify (64), we simply substitute (52) into (49). Then we deduce

$$B_{h,i,k}^H = \varphi_{h,i,k}^H \Phi_{A_{h-1}} F_{k-1} = \varphi_{h,i,k}^H \Phi_{A_{h-1}} F_{k-1}$$  (67)

from (53), and substitute (11) into (48) to obtain

$$c_{k-1} = F_{k-1}^H \Phi_{A_{k-1}} \varphi_{i,k}.$$  (68)

Finally, we compare (67) with (68) to verify (58).

To verify (65) and (66), let us substitute (9) and (45) into (54) to obtain

$$E_k = \left[ \Phi_{A_{k-1}} F_{k-1} \right]_0 = \left[ \Phi_{A_{k-1}} F_{k-1} \right]_0 + \gamma_k \varphi_{i,k},$$  (69)

and substitute (47) into (69) to obtain

$$E_k = \left[ \Phi_{A_{k-1}} F_{k-1} \right]_0 + \gamma_k \varphi_{i,k} - \gamma_k \varphi_{i,k} = \left[ \Phi_{A_{k-1}} F_{k-1} \right]_0,$$  (70)

Finally, let us substitute (54) into (70) to obtain

$$E_k = \left[ E_{k-1} \right]_0 + \gamma_k \varphi_{i,k} - \gamma_k \varphi_{i,k} = \left[ E_{k-1} \right]_0,$$  (71)

from which we can deduce (65) and (66).

To verify (62), we substitute (66) into (53) to obtain

$$B_k = \left[ \Phi_{A_{k-1}} F_{k-1} \right]_0.$$  (72)

Then we substitute (53) into (72) to verify (62) and obtain

$$b_{k} = \Phi_{A_{k-1}} \Phi_{k}.$$  (73)

Then let us verify (61). We can substitute (65) into (73) to obtain

$$b_{k} = \gamma_k \left( \varphi_{i,k} - \varphi_{i,k} \right),$$  (74)

and utilize (7) and (53) to write (74) as (61).

To verify (60), we substitute (66) into (52) to obtain $a_k = \left[ \left( E_{k-1}^H u \right) \right]^T$, into which we substitute (52) to verify (60) and obtain

$$a_k = e_{A_{k-1}}^H u.$$.  (75)

Now let us verify (59). We can substitute (65) into (75) to obtain $a_k = \gamma_k \left( \varphi_{i,k} - \varphi_{i,k} \right)$, into which we substitute (52) to deduce

$$a_k = \gamma_k \left( \varphi_{i,k} - \varphi_{i,k} \right).$$  (76)

Finally we substitute (58) into (76) to obtain

$$a_k = \gamma_k \left( \varphi_{i,k} - \varphi_{i,k} \right).$$  (77)
It can be seen from (51) that $q^H_{i,j} u - b_{i,j,k-1} a_{k-1}$ in (77) is the $i^k$-th entry of the vector $p^{k-1} = \Phi^H u - B_{k-1} a_{k-1}$, i.e., we have verified (59).

To verify (57), substitute (60) and (62) into (51) to deduce

$$p^{k-1} = \Phi^H u - B_k a_{k-2} - b_{i,(k-1)} a_{k-1}, \quad (78)$$

and then substitute (51) (with $k = k - 1$) into (78).

Finally let us verify (63). Substitute (64) into (5) to obtain

$$r^k = u - \Phi_{A_k} F_k a_k. \quad (79)$$

Let $f_k = \begin{bmatrix} t_{k-1}^T & \gamma_k \end{bmatrix}^T$ denote the last $k$-th column of $F_k$. Then substitute (9), (45), (59) and (60) into (79) to obtain $r^k = u - \Phi_{A_{k-1}} F_{k-1} a_{k-1} - a_k \Phi_{A_k} f_k$, i.e., $r^k = r^{k-1} - a_k \Phi_{A_k} f_k$, from which we obtain

$$(r^{k})^H r^k = (r^{k-1})^H r^{k-1} - a_k (r^{k-1})^H \Phi_{A_k} f_k - a_k^H \Phi_{A_k}^H r^{k-1} + a_k^H \Phi_{A_k}^H \Phi_{A_k} f_k. \quad (80)$$

In this paragraph, we consider the second, third and fourth entries in the right side of (80). Substitute (79) into the second entry to obtain

$$a_k (r^{k-1})^H \Phi_{A_k} f_k = a_k u^H \Phi_{A_k} f_k - a_k^H F_{k-1}^H \Phi_{A_k}^H \Phi_{A_k} f_k. \quad (81)$$

It can be seen from (52) that the last $k$-th entry of $a_k$ is

$$a_k = f_k^H \Phi_{A_k}^H u = (u^H \Phi_{A_k} f_k)^H. \quad (82)$$

On the other hand, from (8), (10) and (44) we have $F_k^H \Phi_{A_k}^H F_k = V_k^{-1} G_k V_k^{-1} = V_k^{-1} V_k V_k^H V_k^{-1}$, i.e., $F_k^H \Phi_{A_k}^H F_k = I_k$, from which it is easy to deduce

$$F_{k-1}^H \Phi_{A_k}^H F_k = 0_{k-1} \quad (83)$$

and

$$f_k^H \Phi_{A_k}^H F_k = 1, \quad (84)$$

since $F_{k-1}^H \Phi_{A_k}^H F_k$ and $f_k^H \Phi_{A_k}^H F_k$ constitute the last column (i.e., the $k$-th column) of $F_k^H \Phi_{A_k}^H F_k = I_k$. Then we substitute (82) and (83) into (81), to deduce that the second entry

$$a_k (r^{k-1})^H \Phi_{A_k} f_k = a_k a_k^* - a_k a_{k-1}^H 0_{k-1} = a_k a_k^*. \quad (85)$$

The third entry in the right side of (80) is equal to (85), since

$$a_k^* f_k^H \Phi_{A_k}^H = \left( a_k (r^{k-1})^H \Phi_{A_k} f_k \right)^H = a_k a_k^*. \quad (86)$$

The fourth entry in the right side of (80) is

$$a_k^* a_k^H f_k^H \Phi_{A_k}^H \Phi_{A_k} f_k = a_k a_k^*, \quad (87)$$

where (84) is utilized. Lastly, we can substitute (85)–(87) into (80) to obtain

$$(r^{k})^H r^k = (r^{k-1})^H r^{k-1} - a_k a_k^*, \quad (i.e., (63)).$$

3.3. Several Memory-Saving Versions of the Proposed OMP Implementation

The proposed OMP implementation (v0) requires $N^2 + Nk$ memories (when considering the required memories, we neglect the difference between $k - 1$ and $k$ for simplicity) for $G$ and $B_k$ in the
we substitute (19) into (53) to deduce

we will show that the proposed OMP implementation (v0) can be modified to obtain the proposed memory-saving v1, v2, v3 and v4 (i.e., versions 1, 2, 3 and 4), which all can save some memories at the expense of higher computational complexity.

With respect to the proposed v0, the proposed memory-saving v1 saves the Nk memories for Bk, and requires the extra k^2 memories for Fk. To obtain the proposed memory-saving v1, we need to modify steps 2 and 4 of the proposed OMP implementation (v0) (i.e., Algorithm 5) into steps 2^a and 4^a, respectively, which are described in the following Algorithm 6. To deduce (89) in Algorithm 6, we substitute (19) into (33) to deduce Bk = G(:,Ak)fk, i.e.,

\[ b_k = G(:,A_k)f_k \]  

(88)

and then substitute (88) into (57).

**Algorithm 6** The Proposed Memory-Saving version 1

**Step 2^a.** Projection: if \( k = 1 \), compute \( p^0 = \Phi^H r^0 \); else when \( k \geq 2 \), compute

\[ p^{k-1} = p^{k-2} - G(:,A_{k-1}) \cdot (f_{k-1}a_{k-1}) \]  

(89)

where \( f_{k-1} \) is the last (i.e., the \((k-1)\)-th) column in \( F_{k-1} \).

**Step 4^a.** If \( k = 1 \), compute \( F_1 = \sqrt{g_{1,1}a_1} \), \( \gamma_1 = 1/\sqrt{g_{1,1}a_1} \) and \( a_1 = a_1 = \gamma_1p_{1,1}^0 \); else when \( k \geq 2 \), compute \( c_{k-1} = F_{k-1}\hat{s}_{k-1}, \gamma_k = 1/\sqrt{g_{k,1}a_k - c_{k-1}^Hc_{k-1}}, a_k = \gamma_kp_{k,1}^{k-1} \), \( a_k = \left[ a_{k-1}^T a_k \right]^T \), and \( F_k = \left[ F_{k-1} - \gamma_kF_{k-1}c_{k-1} \right]^T \). For any \( k \geq 1 \), update \( \|r^1\|^2 \) by \( \|r^k\|^2 = \|r^{k-1}\|^2 - \|a_k\|^2 \).

Notice that the above-mentioned \( \hat{s}_{k-1} \) and \( g_{k,1}a_k \) are the vector and scalar in \( G_{A_k} \), respectively.

With respect to the proposed v0, the proposed memory-saving v1 saves the \( N^2 \) memories for \( G \), and requires the extra \( Mk \) memories for \( E_k \). The proposed memory-saving v2 only needs to modify step 4 of the proposed OMP implementation (v0) (i.e., Algorithm 5) into step 4^b, which is described in the following Algorithm 7. It can easily be seen that (90) in Algorithm 7 can be deduced from (72).

**Algorithm 7** The Proposed Memory-Saving version 2

**Step 4^b.** Obtain \( c_{k-1} = \left( b_H_{\lambda_k,1:k-1} \right)^H \) where \( b_H_{\lambda_k,1:k-1} \) is the \( \lambda_k \)-th row of \( B_{k-1} \). Compute

\[ g_{k,1}a_k = r_{\lambda_k}^H \Phi; \lambda_k, \gamma_k = 1/\sqrt{g_{k,1}a_k - c_{k-1}^Hc_{k-1}}, a_k = \gamma_kp_{k,1}^{k-1} \],

\[ e_{k} = \gamma_k \left( \Phi;_{\lambda_k} - E_{k-1}c_{k-1} \right) \], and \( E_k = \left[ E_{k-1} e_k \right] \). Then compute

\[ b_k = \Phi^He_k \]  

(90)

to obtain \( B_k = \left[ B_{k-1} b_k \right] \). Notice that when \( k = 1 \), \( c_0 = E_0 = B_0 = a_0 = 0 \) is assumed.

Finally update the residual error by \( \|r^k\|^2 = \|r^{k-1}\|^2 - \|a_k\|^2 \). If \( F_k \) is required to compute \( \hat{s}_k \) and \( r^k \), compute \( F_k \) by \( F_1 = \sqrt{g_{1,1}a_1} \) when \( k = 1 \), or by \( F_k = \left[ F_{k-1} - \gamma_kF_{k-1}c_{k-1} \right]^T \gamma_k \) when \( k \geq 2 \).

With respect to the proposed v0, the proposed memory-saving v3 saves the \( N^2 + Nk \) memories for \( G \) and \( B_k \), and requires the extra \( Mk \) memories for \( E_k \). It modifies steps 2 and 4 of the proposed
OMP implementation (v0) (i.e., Algorithm 5) into steps 2\(^c\) and 4\(^d\), respectively, which are described in the following Algorithm 8. We only need to substitute (73) into (57) to obtain (91) in Algorithm 8. Moreover, we can substitute (54) into (67) to obtain \(b_{r',1:k-1}^H = \varphi_{r',k}^H F_{k-1}\), which is then substituted into (58) to obtain (92) in Algorithm 8.

**Algorithm 8 The Proposed Memory-Saving version 3**

**Step 2\(^c\).** Projection: if \(k = 1\), compute \(p^0 = \Phi^H r^0\); else when \(k \geq 2\), compute \(p^{k-1}(k \geq 2)\) by

\[
p^{k-1} = p^{k-2} - \Phi^H \left( e_{(k-1)a_{k-1}} \right).
\]

**Step 4\(^c\).** Compute \(c_{k-1}\) from \(E_{k-1}\) by

\[
c_{k-1} = E_{k-1}^H \varphi_{r,\beta}.
\]

Then compute \(s_{\lambda_k,\lambda_k} = \varphi_{\lambda_k,\lambda_k}^H \varphi_{\lambda_k,\lambda_k}, \gamma_k = 1/\sqrt{s_{\lambda_k,\lambda_k} - e_{k-1}^H c_{k-1}}, a_k = \gamma_k p_{\lambda_k}^{k-1}, a_k = \left[ a_{k-1}^T \ a_k \right]^T, e_k = e_{k-1} \varphi_{\lambda_k} - E_{k-1} c_{k-1},\) and \(E_k = \left[ E_{k-1} \ e_k \right]\). Finally update the residual energy by

\[
\|r^k\|^2 = \|r^{k-1}\|^2 - ||a_k||^2.
\]

If \(c_k\) is required to compute \(s_k\) and \(r^k\), compute \(F_k\) by \(F_1 = \sqrt{s_{\lambda_1,\lambda_1}}\) when \(k = 1\), or by \(F_k = \left[ F_{k-1} - \gamma_k F_{k-1} c_{k-1} \ \ 0_{d_{k-1}}^T \ g_k \right]\) when \(k \geq 2\).

With respect to the proposed memory-saving v3, the proposed memory-saving v4 saves the \(Mk\) memories for \(E_k\) and requires the extra \(k^2\) memories for \(F_k\). The proposed memory-saving v4 needs to modify steps 2 and 4 of the proposed OMP implementation (v0) (i.e., Algorithm 5) into steps 2\(^d\) and 4\(^d\), respectively, which are described in the following Algorithm 9. From (54), we can deduce \(e_{\lambda_k} = \Phi_{\lambda_k} f_k\), which is then substituted into (91) to obtain (93) in Algorithm 9.

**Algorithm 9 The Proposed Memory-Saving version 4**

**Step 2\(^d\).** Projection: if \(k = 1\), compute \(p^0 = \Phi^H r^0\); else when \(k \geq 2\), compute \(p^{k-1}(k \geq 2)\) by

\[
p^{k-1} = p^{k-2} - \Phi^H \left( \Phi_{\lambda_k-1}^H (f_{k-1} a_{k-1}) \right).
\]

**Step 4\(^d\).** If \(k = 1\), compute \(s_{\lambda_1,\lambda_1} = \varphi_{\lambda_1,\lambda_1}^H \varphi_{\lambda_1,\lambda_1}, F_1 = \sqrt{s_{\lambda_1,\lambda_1}}, \gamma_1 = 1/\sqrt{s_{\lambda_1,\lambda_1}}\) and \(a_1 = a_1 = \gamma_1 p_{\lambda_1}^0\); else when \(k \geq 2\), compute \(s_{\lambda_k,\lambda_k} = \varphi_{\lambda_k,\lambda_k}^H \varphi_{\lambda_k,\lambda_k}, s_{k-1} = \Phi_{\lambda_k-1}^H \varphi_{\lambda_k,\lambda_k}, c_{k-1} = E_{k-1}^H s_{k-1},\)

\[
\gamma_k = 1/\sqrt{s_{\lambda_k,\lambda_k} - e_{k-1}^H c_{k-1}}, a_k = \gamma_k p_{\lambda_k}^{k-1}, a_k = \left[ a_{k-1}^T \ a_k \right]^T, \text{and } F_k = \left[ F_{k-1} - \gamma_k F_{k-1} c_{k-1} \ \ 0_{d_{k-1}}^T \ g_k \right].
\]

For any \(k \geq 1\), update \(\|r^k\|^2\) by \(\|r^k\|^2 = \|r^{k-1}\|^2 - ||a_k||^2\).

4. Analysis of Presented OMP Implementations

In this section, we compare the presented implementations for OMP, by theoretical and empirical analysis.

4.1. Theoretical Analysis of Computational Complexities and Memory Requirements

Table 1 compares the expected computational complexities (in the implementations with \(N^2\) memories (to store \(G\), the Gram matrix of the dictionary), we assume that \(G\) has been computed offline and stored) and memory requirements of the existing and proposed OMP implementations in the \(k\)-th iteration \((k \geq 2)\), where the complexities are the numbers of multiplications and additions, which include both dominating terms and coefficients. It can be seen that each implementation needs
nearly the same number of multiplications and additions. “Dominating terms” here means that we omit the terms that are $O(N)$, $O(M)$ or $O(k)$. For example, “Dominating terms” for Chol-2 only include the terms for (18) and to solve the triangular systems (17), (21) and (22). When evaluating the memory requirements for Table 1, we neglect the fact that we may only store parts of some matrices, such as the Hermitian Gram matrix $G$ and the triangular Cholesky factor. The entries in Table 1 are counted from the above-described relevant equations. The complexities of the existing OMP implementations listed in Table 1 are consistent with those in Table 1 of [24], while the complexity of each proposed OMP implementation is broken down for each equation in Table 2.

In Table 1, Naive, Chol-1, Chol-2, QR-1, QR-2, MIL-1 and MIL-2 denote the naive OMP implementation (i.e., Algorithm 1), two implementations by the Cholesky factorization (i.e., Algorithm 2), two implementations by the QR factorization (i.e., Algorithm 3), and two implementations by the Matrix Inversion Lemma (i.e., Algorithm 4), respectively. On the other hand, Proposed-v0, Proposed-v1, Proposed-v2, Proposed-v3 and Proposed-v4 in Tables 1 and 2 denote the proposed implementation of OMP (v0) and the proposed 4 memory-saving versions, which have been described in Algorithms 5–9, respectively.

### Table 1. Comparison of complexities and memory requirements in the $k$-th iteration ($k \geq 2$) among OMP implementations, where the dictionary $\Phi$ is $M \times N$.

| Algorithm | Complexity | Memory |
|-----------|------------|--------|
| Naive     | $NM + 2Mk + k^2$ $+ \frac{1}{2}(Mk^2 + k^3)$ | $NM$ |
| Proposed-v4 | $NM + 2Mk + k^2$ | $NM + k^2$ |
| Proposed-v3 | $NM + 2Mk$ | $NM + Mk$ |
| QR-1     | $NM + 2Mk$ | $NM + Mk + k^2$ |
| Proposed-v2 | $NM + Mk$ | $N(M + k) + Mk$ |
| Proposed-v1 | $Nk + k^2$ | $N^2 + NM + k^2$ |
| Chol-2   | $Nk + \frac{3}{2}k^2$ | $N^2 + NM + k^2$ |
| MIL-2    | $Nk + \frac{3}{2}k^2$ | $N^2 + NM + k^2$ |
| Chol-1   | $NM + Mk + \frac{3}{2}k^2$ | $N^2 + NM + k^2$ |
| MIL-1    | $Nk + 3Mk$ | $N^2 + NM + Mk$ |
| QR-2     | $Nk + 2Mk + k^2$ | $N^2 + NM + Mk + k^2$ |
| Proposed-v0 | $Nk$ | $N^2 + N(M + k)$ |

From Table 1, it can be seen that the proposed OMP implementation (i.e., Proposed-v0) needs the least computational complexities. With respect to any of the existing efficient implementations (i.e., Chol-2, MIL-2, Chol-1, MIL-1 and QR-2) spending $N^2$ memories for $G$, Proposed-v1 requires less computational complexities and the same or less memories. On the other hand, with respect to the only existing efficient implementation not storing $G$, i.e., QR-1, Proposed-v2 needs less computational complexities and a little more memories, while Proposed-v3 needs the same complexities and less memories. Lastly, with respect to Naive, Proposed-v4 needs much less computational complexities and only a little more memories.
Table 2. Complexities of the equations in the k-th iteration (k ≥ 2) for the proposed OMP implementations.

| Algorithm      | Complexity | Equation                                                                 |
|----------------|------------|---------------------------------------------------------------------------|
| Proposed-v4    | NM + Mk    | \( \mathbf{p}^{k-1} = \mathbf{p}^{k-2} - \mathbf{\Phi}^H \cdot (\mathbf{f}_{k-1} a_{k-1}) \) |
|                | Mk         | \( \mathbf{s}_{k-1} = \mathbf{\Phi}^H_{k-1} \mathbf{q}_{\lambda_{k-1}} \)   |
|                | \( \frac{1}{2}k^2 \) | \( \mathbf{c}_{k-1} = \mathbf{F}_{k-1}^H \mathbf{s}_{k-1} \)               |
|                | \( \frac{1}{2}k^2 \) | \(-\gamma_k \mathbf{F}_{k-1} \mathbf{c}_{k-1} \)                          |
| Proposed-v3    | NM         | \( \mathbf{p}^{k-1} = \mathbf{p}^{k-2} - \mathbf{\Phi}^H \cdot (\mathbf{e}_{(k-1)} a_{k-1}) \) |
|                | Mk         | \( \mathbf{c}_{k-1} = \mathbf{E}_{k-1}^H \mathbf{q}_{\lambda_{k-1}} \)       |
|                | Mk         | \( \mathbf{e}_{k} = \gamma_k (\mathbf{q}_{\lambda_{k-1}} - \mathbf{E}_{k-1} \mathbf{c}_{k-1}) \) |
| Proposed-v2    | NM         | \( \mathbf{b}_{k} = \mathbf{\Phi}^H \mathbf{e}_{k} \)                      |
|                | Mk         | \( \mathbf{e}_{k} = \gamma_k (\mathbf{q}_{\lambda_{k-1}} - \mathbf{E}_{k-1} \mathbf{c}_{k-1}) \) |
| Proposed-v1    | \( \frac{1}{2}Nk \) | \( \mathbf{p}^{k-1} = \mathbf{p}^{k-2} - \mathbf{G} (\mathbf{\lambda}_{k-1}) \cdot (\mathbf{f}_{k-1} a_{k-1}) \) |
|                | \( \frac{1}{2}k^2 \) | \( \mathbf{c}_{k-1} = \mathbf{F}_{k-1}^H \mathbf{s}_{k-1} \)               |
|                | \( \frac{1}{2}k^2 \) | \(-\gamma_k \mathbf{F}_{k-1} \mathbf{c}_{k-1} \)                          |
| Proposed-v0    | \( \frac{1}{2}Nk \) | \( \mathbf{b}_{k} = \gamma_k (\mathbf{g}_{\lambda_{k-1}} - \mathbf{B}_{k-1} \mathbf{c}_{k-1}) \) |

4.2. Empirical Comparison for OMP Implementations

We perform MATLAB simulations to compare all the proposed 5 versions of OMP implementations with the existing ones, on a 64-bit 2.4-GHz Microsoft-Windows Xeon workstation with 15.9 GB of RAM. We give the simulation results for numerical errors and computational time (the MATLAB code to generate Figures 1–6 of this paper is shared in https://github.com/zhuhufei/OMP.git in Figures 1–6. Moreover, we show the floating-point operations (flops) required by different OMP implementations in Figures 7–9, where the complexities listed in Table 1 are utilized to count the flops. To obtain Figures 1–9, the shared simulation code [42] is utilized.

Figure 1. Numerical errors of Chol-2 and the proposed 5 versions (of OMP implementations) with respect to Naive, as a function of the algorithm iteration \( k/K \), where the sparsity \( \rho = K/M = 0.5 \) and \( N = M = 400 \).

As Figure 1 in [24], Figure 1 here shows the accumulation of numerical errors in the recursive computation of the inner products \( \mathbf{p}^k \)'s and the solutions \( \mathbf{s}_k \)'s, where the errors mean the differences...
in the computed $p^k$s and $\hat{x}_k$s between the naive implementation and any considered efficient implementation. For example, the numerical errors of Chol-2 are defined by

$$ERROR_{p^k} = \frac{\|p^k(\text{Chol-2}) - p^k(\text{Naive})\|^2}{\|p^k(\text{Naive})\|^2},$$  \hspace{1cm} (94)$$

and

$$ERROR_{\hat{x}_k} = \frac{\|\hat{x}_k(\text{Chol-2}) - \hat{x}_k(\text{Naive})\|^2}{\|\hat{x}_k(\text{Naive})\|^2}. \hspace{1cm} (95)$$

We perform 100 independent trials, and the other simulation parameters are usually identical to those for Figure 1 in [24]. For example, we set $N = M = 400$, $K = 200$ and the sparsity $\rho = K/M = 0.5$, and sample sparse vectors from a Rademacher or Normal distribution. For any $k$-th ($k = 1, 2, \ldots, K$) iteration, we compute the mean relative errors of the inner products and those of the solutions, which are plotted in Figure 1a and Figure 1b, respectively. Since Figure 1 in [24] shows that QR-1, Chol-2 and MIL-1 have nearly the same numerical errors, here we only compare the numerical errors of the proposed five versions with those of Chol-2. Moreover, we do the same simulations in Figure 2 as in Figure 1, except that we set the sparsity $\rho = 0.1$ instead of $\rho = 0.5$.

![Figure 2. Numerical errors of Chol-2 and the proposed 5 versions (of OMP implementations) with respect to Naive, as a function of the algorithm iteration $k/K$, where the sparsity $\rho = K/M = 0.1$ and $N = M = 400$.](image)

(a) Mean error in inner products.  \hspace{2cm} (b) Mean error in solutions.

In Figure 3, we perform Monte Carlo simulations to show the Signal-to-Reconstruction-Error Ratio (SRER) of different OMP implementations versus Indeterminacy $M/N$ (i.e., Fraction of Measurements), where we follow the simulations for Figures 1 and 2 in [44], and SRER is defined by

$$SRER (dB) \triangleq 10 \log_{10} \frac{E \|x\|^2}{E \|x - \hat{x}\|^2}. \hspace{1cm} (96)$$

As Figure 1 in [44], Figure 3a gives the SRER performance for Normal (i.e., Gaussian) sparse signals in the clean measurement case and the noisy measurement case with $SMNR = 15$ dB, where $SMNR$ denotes Signal to Measurement-Noise Ratio and satisfies

$$SMNR (dB) \triangleq 10 \log_{10} \frac{E \|x\|^2}{E \|n\|^2}. \hspace{1cm} (97)$$

Similarly, Figure 3b gives the SRER performance for Rademacher sparse signals, as Figure 2 in [44]. In Figure 3, we set $N = 500$ and $K = 20$. Moreover, we generate the dictionary $\Phi$ 100 times and for each realization of $\Phi$, we generate a sparse signal $x$ 100 times.
Figure 3. Signal-to-reconstruction-error ratio (SRER) of different OMP implementations versus indeterminacy $M/N$ (i.e., fraction of measurements).

From Figures 1–3, it can be seen that the differences in numerical errors between the implementations are inconsequential. Then let us show the computational time of 100 independent runs of $K$ iterations for $\Phi$ of size $M \times N$ in Figure 4, where we follow (Figure 2 in [24]) gives the computational time of four existing OMP implementations, i.e., Naive, QR-1, Chol-2 and MIL-1. Instead of MIL-1 in Figure 2 of [24], we simulate MIL-2 in our Figure 4, since MIL-1 is slower than MIL-2 in our simulations. Actually Chol-1 and QR-2 omitted in both Figure 2 of [24] and our Figure 4 are also slower than Chol-2 and QR-1, respectively) the simulations for Figure 2 in [24]. We measure time by tic/toc in MATLAB, and show $\log_{10}(T_{\text{naive}}(K, M, N)/T_{\text{imp}}(K, M, N))$ for two different $N$ in Figure 4. To do fair comparisons, the MATLAB code for the proposed five versions is similar to the shared simulation code [42] (of [24]). Moreover, we show several sections of the 3D Figure 4b in Figures 5 and 6, where we show $\text{Time Ratio} = T_{\text{naive}}(K, M, N)/T_{\text{imp}}(K, M, N)$ instead of $\log_{10} \text{Time Ratio}$ in Figure 4. Figure 5a,b set the indeterminacies $M/N$ to be 0.1155 and 0.5086, respectively, to show time ratios as a function of the sparsity $K/M$. Then Figure 6a,b set the sparsities $K/M$ to be 0.1155 and 0.5086, respectively, to show time ratios as a function of the indeterminacies $M/N$.

Figure 4. Computational time of the OMP implementations as a function of problem sizes for two different dimensions $N$. 

(a) $N = 200$. (b) $N = 800$. 

(a) SRER for Normal Signals. (b) SRER for Rademacher Signals.
From Figures 4–6, it can be seen that Proposed-v0 is the fastest for almost all $N$, $M$ and $K$, while Proposed-v2 and Proposed-v3 are faster than the existing implementations for most problem sizes. Figures 5 and 6 also show that in most cases, the speedups in computational time of Proposed-v0, Proposed-v2, Proposed-v3 and QR-1 over Naive grow almost linearly with the increase of the sparsity $K/M$ for the two fixed indeterminacies $M/N$, and grow almost linearly with the increase of the indeterminacy $M/N$ for the two fixed sparsities $K/M$. The computational time of Proposed-v4 is near or less than that of an existing efficient implementation (e.g., MIL-2 or QR-1) for most problem sizes, while the memory requirements of Proposed-v4 are much less than those of any existing efficient implementation, and are only a little more than the (known) minimum requirements (i.e., the requirements of Naive). Thus Proposed-v4 seems to be a good choice when we need an efficient implementation with the minimum memory requirements. Moreover, from Table 1, Figures 4–6, it can be seen that with respect to Chol-2 and MIL-2, Proposed-v1 needs the same size of memories and $\frac{1}{2}k^2$ less of complexities, and is usually a little faster.

In Figure 7, we show $\log_{10}\left(\frac{\text{Flops}_{\text{naive}}(K, M, N)}{\text{Flops}_{\text{imp}}(K, M, N)}\right)$ for two different $N$. Then in Figures 8 and 9, we show the flops ratios of Chol-2 over Proposed-v0, Proposed-v1 and MIL-2, respectively, to compare flops among the efficient OMP implementations storing $G$, and we also show the flops ratios of of QR-1 over Proposed-v2, Proposed-v3 and Proposed-v4, respectively, to compare...
flops among the efficient implementations that do not store $G$. In Figure 8, we give the flops ratios as a function of the sparsity $K/M$ for the indeterminacies $M/N = 0.1155$ and $M/N = 0.5086$, respectively, while in Figure 9, we give the flops ratios as a function of the indeterminacy $M/N$ for the sparsities $K/M = 0.1155$ and $K/M = 0.5086$, respectively.

From Figure 7, it can be seen that the speedups in flop numbers of Proposed-v0, Proposed-v1, Chol-2, QR-2 and MIL-1 over Naive are obvious for almost all problem sizes. From Figures 8 and 9, it can be seen that the speedups in flop numbers of Proposed-v0 and Proposed-v1 over Chol-2 and those of Proposed-v2 over QR-1 grow linearly with the increase of the sparsity $K/M$ for the two fixed indeterminacies $M/N$, and grow linearly with the increase of the indeterminacy $M/N$ for the two fixed sparsities $K/M$. On the other hand, with respect to QR-1, Proposed-v3 requires the similar number of flops and Proposed-v4 requires a little more flops, while as shown in Table 1, both Proposed-v3 and Proposed-v4 require less memories.

**Figure 7.** Flops of the OMP implementations as a function of problem sizes for two different dimensions $N$.

(a) $N = 200$.  
(b) $N = 800$.  

**Figure 8.** Flops of the OMP implementations as a function of the sparsity $K/M$ for two different indeterminacies $M/N$.

(a) Indeterminacy $M/N = 0.1155$, $N = 800$.  
(b) Indeterminacy $M/N = 0.5086$, $N = 800$.  

| Sparsity $K/M$ | Flops Ratio |
|----------------|-------------|
| Proposed-v0    | Chol-2/Proposed-v0 |
| Proposed-v1    | Chol-2/Proposed-v1 |
| Proposed-v2    | Chol-2/Proposed-v2 |
| Proposed-v3    | Chol-2/Proposed-v3 |
| Proposed-v4    | Chol-2/Proposed-v4 |
| QR-1/Proposed-v2 | QR-1/Proposed-v2 |
| QR-1/Proposed-v3 | QR-1/Proposed-v3 |
| QR-1/Proposed-v4 | QR-1/Proposed-v4 |
| QR-1/Chol-2    | QR-1/Chol-2    |
| QR-1/QR-2      | QR-1/QR-2      |
| QR-1/MIL-1     | QR-1/MIL-1     |
| QR-1/Proposed-v0 | QR-1/Proposed-v0 |
| QR-1/Proposed-v1 | QR-1/Proposed-v1 |
| QR-1/Proposed-v2 | QR-1/Proposed-v2 |
| QR-1/Proposed-v3 | QR-1/Proposed-v3 |
| QR-1/Proposed-v4 | QR-1/Proposed-v4 |
Figure 9. Flops of the OMP implementations as a function of the indeterminacy $M/N$ for two different sparsities $K/M$.

Figures 4 and 7 indicate that Proposed-v1, Chol-2 and MIL-2 need less flops and more computational time, with respect to Proposed-v2, Proposed-v3 and QR-1, while Table 1 and Figure 2 of [24] also indicate that with respect to QR-1, Chol-2 and MIL-1 need less flops and more computational time. One possible reason for this phenomenon could be that the computational time is decided not only by the required flops, but also by the required operations for memory access. Specifically, this reason can partly explain why MIL-2 is obviously slower than Chol-2, since in the $k$-th iteration, MIL-2 needs to access $k^2$ memories in (38) to obtain the inverse of $G_{A_k}^{-1}$, while Chol-2 only needs to access $k$ memories in (20) to obtain the Cholesky factor of $G_{A_k}^{-1}$.

In addition to the flops and the operations for memory access, the parallelizability of the OMP implementation can also affect the computational time. When considering this factor, notice that the proposed five versions do not need any back-substitution in each iteration, and only include parallelizable matrix-vector products [45]. Contrarily, the existing Chol-1, Chol-2 and QR-2 usually solve triangular systems by back-substitutions [43], which are inherent serial processes unsuitable for parallel implementations [45]. It can be easily seen that Chol-1 and Chol-2 both need $\frac{3}{2}k^2$ multiplications and additions for the back-substitutions to solve $3k \times k$ triangular systems (17), (21) and (22), while QR-2 needs $k^2$ multiplications and additions for the back-substitutions to solve $2k \times k$ triangular systems (25) and (26).

5. Conclusions

Based on the efficient inverse Cholesky factorization proposed in [36], we propose an implementation of OMP (i.e., the proposed v0) and its four memory-saving versions (i.e., the proposed v1, v2, v3 and v4), and compare them with the existing ones. Among all the OMP implementations, the proposed v0 needs the least computational complexity, and is the fastest in the simulations for almost all problem sizes. The proposed v1, v2, v3 and v4 are all good tradeoffs between computational complexities/time and memory requirements. When only considering the efficient OMP implementations that spend $N^2$ memories to store $G$ (i.e., the Gram matrix of the dictionary), the proposed v1 seems to be better than all the existing ones (i.e., QR-2, MIL-1, MIL-2, Chol-1 and Chol-2), since it needs less computational complexity, is usually faster in the simulations, and does not require more memories. When only considering the efficient OMP implementations that do not store $G$, with respect to the only existing one (i.e., QR-1), the proposed v2 needs less computational complexity and a little more memories, while the proposed v3 needs the same complexity and less memories. Moreover, our simulations show that both the proposed v2 and v3 are faster than all the existing implementations for most problem sizes. Lastly, with respect to the naive OMP implementation
that has the (known) minimum memory requirements, the proposed v4 only requires a little more memories, and needs much less computational time and complexities that are similar to those required by some existing efficient OMP implementations (e.g., the computational time of MIL-2 and Chol-2, and the complexities of QR-1). Thus the proposed v4 seems to be the best choice when we need an efficient implementation with the minimum memory requirements.

Our simulations show that the proposed five versions and the existing ones have nearly the same numerical errors. On the other hand, our simulations also show that some speedups in computational time or complexities of the proposed OMP implementations over the existing ones grow almost linearly with the increase of the sparsity $K/M$ or the indeterminacy $M/N$, e.g., the speedups in computational time of the proposed v0, v2 and v3 over Naive, those in complexities of the proposed v0 and v1 over Chol-2, and those in complexities of the proposed v2 over QR-1. Moreover, in each iteration, some existing efficient OMP implementations (e.g., Chol-1, Chol-2 and QR-2) solve triangular systems by back-substitutions (that are inherent serial processes unsuitable for parallel implementations), while the proposed five versions of OMP implementations do not need any back-substitution, and only include parallelizable matrix-vector products.

**Author Contributions:** Conceptualization, H.Z. and W.C.; methodology, H.Z., Y.W. and W.C.; software, W.C. and Y.W.; validation, Y.W.; formal analysis, H.Z. and W.C.; investigation, W.C.; resources, Y.W.; data curation, Y.W.; writing—original draft preparation, H.Z.; writing—review and editing, W.C.; visualization, H.Z.; supervision, W.C.; project administration, W.C.; funding acquisition, W.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported in part by the National Natural Science Foundation of China with No. 61621062, 61773407 and 61872408, in part by the Natural Science Foundation of Hunan Province of China with No. 2019JJ40050, in part by the Key Scientific Research Project of Education Department of Hunan Province of China with No. 19A099.

**Conflicts of Interest:** The authors declare no conflict of interest.

**References and Note**

1. Mallat, S.; Zhang, Z. Matching pursuits with time-frequency dictionaries. *IEEE Trans. Signal Process.* **1993**, *41*, 3397–3415. [CrossRef]
2. Rao, B.D.; Kreutz-Delgado, K. An affine scaling methodology for best basis selection. *IEEE Trans. Signal Process.* **1999**, *47*, 187–200. [CrossRef]
3. Chen, S.B.; Donoho, D.L.; Saunders, M.A. Atomic decomposition by basis pursuit. *SIAM J. Sci. Comput.* **1998**, *20*, 33–61. [CrossRef]
4. Miller, A.J. *Subset Selection in Regression*, 2nd ed.; Chapman and Hall: London, UK, 2002.
5. Temlyakov, V. Nonlinear methods of approximation. *Found. Comput. Math.* **2002**, *3*, 33–107. [CrossRef]
6. DeVore, R.A. Nonlinear Approximation and its Applications. In *Multiscale, Nonlinear, and Adaptive Approximation*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 169–201.
7. Li, H.; Zhao, M.; Yan, H.; Yang, X. Nanoseconds Switching Time Monitoring of Insulated Gate Bipolar Transistor Module by Under-Sampling Reconstruction of High-Speed Switching Transitions Signal. *Electronics* **2019**, *8*, 1203. [CrossRef]
8. Mansoor, B.; Nawaz, S.J.; Gulfam, S.M. Massive-MIMO Sparse Uplink Channel Estimation Using Implicit Training and Compressed Sensing. *Appl. Sci.* **2017**, *7*, 63. [CrossRef]
9. Khwaja, A.S.; Cetin, M. Compressed Sensing ISAR Reconstruction Considering Highly Maneuvering Motion. *Electronics* **2017**, *6*, 21. [CrossRef]
10. Kim, Y.J.; Cho, Y.S. Cell ID and Angle of Departure Estimation for Millimeter-wave Cellular Systems in Line-of-Sight Dominant Conditions Using Zadoff-Chu Sequence Based Beam Weight. *Electronics* **2020**, *9*, 335. [CrossRef]
11. Liu, L.; Zhao, H.; Li, M.; Zhou, L.; Jin, J.; Zhang, J.; Lv, Z.; Ren, H.; Mao, J. Modelling and Simulation of Pseudo-Noise Sequence-Based Underwater Acoustic OSDK Communication System. *Appl. Sci.* **2019**, *9*, 2063. [CrossRef]
12. Wei, Z.; Zhang, J.; Xu, Z.; Liu, Y. Optimization Methods of Compressively Sensed Image Reconstruction Based on Single-Pixel Imaging. *Appl. Sci.* **2020**, *10*, 3288. [CrossRef]
13. Zhu, Z.; Qi, G.; Chai, Y.; Li, P. A Geometric Dictionary Learning Based Approach for Fluorescence Spectroscopy Image Fusion. *Appl. Sci.* 2017, 7, 161. [CrossRef]

14. Tropp, J.; Gilbert, A.; Strauss, M. Algorithms for simultaneous sparse approximation. Part I: Greedy pursuit. *Signal Process.* 2006, 86, 572–588. [CrossRef]

15. Donoho, D.; Stodden, V.; Tsag, Y. SparSelab. 2007. Available online: http://sparselab.stanford.edu/ (accessed on 25 July 2020).

16. Damnjanovic, I.; Davies, M.E.P.; Plumbley, M.D. Smallbox—An evaluation framework for sparse representations and dictionary learning algorithms. In Proceedings of the LVA/ICA, St. Malo, France, 27–30 September 2010.

17. Rubinstein, R.; Zibulevsky, M.; Elad, M. Efficient Implementation of the K-SVD Algorithm Using Batch Orthogonal Matching Pursuit; CS Technion: Haifa, Israel, 2008. Available online: http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf (accessed on 25 July 2020).

18. Tropp, J.A.; Gilbert, A.C. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Trans. Inf. Theory* 2007, 53, 4655–4666. [CrossRef]

19. Cotter, S.F.; Adler, J.; Rao, B.D.; Kreutz-Delgado, K. Forward sequential algorithms for best basis selection. *IEEE Proc. Vision Image Signal Process.* 1999, 146, 235–244. [CrossRef]

20. Blumensath, T.; Davies, M.E. In Greedy Pursuit of New Directions: (Nearly) Orthogonal Matching Pursuit by Directional Optimisation. In Proceedings of the EUSIPCO, Poznan, Poland, 3–7 September 2007.

21. Huang, G.; Wang, L. High-speed signal reconstruction with orthogonal matching pursuit via matrix inversion bypass. In Proceedings of the 2012 IEEE Workshop on Signal Processing Systems (SiPS), Quebec City, QC, Canada, 17–19 October 2012; pp. 191–196.

22. Pati, Y.; Rezaifar, R.; Krishnaprasad, P. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In Proceedings of the ACSSC, Pacific Grove, CA, USA, 1–3 November 1993.

23. Fang, Y.; Chen, I.; Wu, J.; Huang, B. GPU Implementation of Orthogonal Matching Pursuit for Compressive Sensing. In Proceedings of the ICPADS 2011, Tainan, Taiwan, 7–9 December 2011.

24. Sturm, B.L.; Christensen, M.G. Comparison of orthogonal matching pursuit implementations. In Proceedings of the EUSIPCO 2012, Bucharest, Romania, 27–31 August 2012.

25. Yang, D.; Peterson, G.D. Compressed sensing and Cholesky decomposition on FPGAs and GPUs. *Parallel Comput.* 2012, 38, 421–437. [CrossRef]

26. Blache, P.; Rabah, H.; Amira, A. High level prototyping and FPGA implementation of the orthogonal matching pursuit algorithm. In Proceedings of the 2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA), Montreal, QC, Canada, 2–5 July 2012; pp. 1336–1340.

27. Rabah, H.; Amira, A.; Mohanty, B.K.; Almaadeed, S.; Meher, P.K. FPGA implementation of orthogonal matching pursuit for compressive sensing reconstruction. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 2015, 23, 2209–2220. [CrossRef]

28. Liu, S.; Lyu, N.; Wang, H. The implementation of the improved OMP for AIC reconstruction based on parallel index selection. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 2018, 26, 319–328. [CrossRef]

29. Bai, L.; Maechler, P.; Muehleberguber, M.; Kaeslin, H. High-speed compressed sensing reconstruction on FPGA using OMP and AMP. In Proceedings of the 2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012), Seville, Spain, 9–12 December 2012; pp. 53–56.

30. Stanislaus, J.L.V.M.; Mohsenin, T. Low-complexity FPGA implementation of compressive sensing reconstruction. In Proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC), San Diego, CA, USA, 28–31 January 2013; pp. 671–675.

31. Yu, Z.; Su, J.; Yang, F.; Su, Y.; Zeng, X.; Zhou, D.; Shi, W. Fast compressive sensing reconstruction algorithm on FPGA using orthogonal matching pursuit. In Proceedings of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, QC, Canada, 22–25 May 2016; pp. 249–252.

32. Jhang, J.-W.; Huang, Y.-H. A high-SNR projection-based atom selection OMP processor for compressive sensing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 2016, 24, 3477–3488. [CrossRef]

33. Huang, G.; Wang, L. An FPGA-based architecture for high-speed compressed signal reconstruction. *ACM Trans. Embed Comput. Syst.* 2017, 16, 1–23. [CrossRef]

34. Kulkarni, A.; Mohsenin, T. Low overhead architectures for OMP compressive sensing reconstruction algorithm. *IEEE Trans. Circuits Syst. I Reg. Pap.* 2017, 64, 1468–1480. [CrossRef]
35. Kim, S.; Yun, U.; Jang, J.; Seo, G.; Kang, J.; Lee, H.-N.; Lee, M. Reduced Computational Complexity Orthogonal Matching Pursuit Using a Novel Partitioned Inversion Technique for Compressive Sensing. *Electronics* 2018, 7, 206. [CrossRef]

36. Zhu, H.; Chen, W.; Li, B.; Gao, F. An Improved Square-root Algorithm for V-BLAST Based on Efficient Inverse Cholesky Factorization. *IEEE Trans. Wirel. Commun.* 2011, 10, 43–48. [CrossRef]

37. Zhu, H.; Yang, G.; Chen, W. Efficient Implementations of Orthogonal Matching Pursuit Based on Inverse Cholesky Factorization. In Proceedings of the IEEE Vehicular Technology Conference (VTC) 2013, Las Vegas, NV, USA, 2–5 September 2013.

38. Donoho, D.; Tsaig, Y.; Drori, I.; Starck, J.L. Sparse Solution of Underdetermined Systems of Linear Equations by Stagewise Orthogonal Matching Pursuit. *IEEE Trans. Inf. Theory* 2012, 58, 1094–1121. [CrossRef]

39. Huang, H.; Makur, A. Backtracking-Based Matching Pursuit Method for Sparse Signal Reconstruction. *IEEE Signal Process. Lett.* 2011, 18, 391–394. [CrossRef]

40. Blumensath, T.; Davies, M.E. Gradient pursuits. *IEEE Trans. Signal Process.* 2008, 56, 2370–2382. [CrossRef]

41. Sturm, B.L.; Christensen, M.G.; Gribonval, R. Cyclic pure greedy algorithms for recovering compressively sampled sparse signals. In Proceedings of the 2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 6–9 November 2011.

42. MATLAB code accompanying reference [24]. Available online: http://www.eecs.qmul.ac.uk/~sturm/software/OMPefficiency.zip (accessed on 25 July 2020).

43. Golub, G.H.; van Loan, C.F. *Matrix Computations*, 3rd ed.; Johns Hopkins University Press: Baltimore, MD, USA, 1996.

44. Ambat, S.K.; Chatterjee, S.; Hari, K.V.S. Fusion of Greedy Pursuits for compressed sensing signal reconstruction. In Proceedings of the 20th European Signal Processing Conference (EUSIPCO), Bucharest, Romania, 27–31 August 2012; pp. 1434–1438.

45. Baranoski, E.J. Triangular factorization of inverse data covariance matrices. In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing ICASSP 1991, Toronto, ON, Canada, 14–17 April 1991; pp. 2245–2247.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).