Sparse Random Approximation and Lossy Compression

M. Andrecut *

Abstract— **We discuss a method for sparse signal approximation, which is based on the correlation of the target signal with a pseudo-random signal, and uses a modification of the greedy matching pursuit algorithm. We show that this approach provides an efficient encoding-decoding method, which can be used also for lossy compression and encryption purposes.**

Keywords: sparse approximation; lossy compression; matching pursuit.

1 Introduction

Recently there has been an increased interest in alternatives to traditional signal approximation and compression techniques. Several new methods for approximating signals in dictionaries of waveforms (Fourier, Gabor, wavelets, cosine packets etc.) have been proposed [1]-[7]. Such a dictionary is a collection of waveforms represented by discrete time signals, also called atoms. Using this approach, a discrete-time signal of length N is decomposed in a sparse linear combination of dictionary atoms with corresponding coefficients. Dictionaries can be complete, if they contain N atoms, and respectively overcomplete, if they contain more than N atoms. Most of the dictionaries are obtained by merging complete dictionaries consisting of different types of waveforms. For typical applications the size of these dictionaries is quite big (tens-of-thousands of atoms), and raises high computational difficulties and memory requirements. The decomposition of a signal using overcomplete dictionaries is nonunique, since some elements in the dictionary have representations in terms of other elements. Therefore, the sparse approximation problem in an overcomplete dictionary is to find the minimal representation of a signal, in terms of dictionary atoms. Finding the sparsest approximation of a signal from an arbitrary dictionary is an NP-hard problem. Despite of this, several sub-optimal methods have been recently developed, such that a wide range of applications (coding, source separation, denoising etc.) have benefited from the progress made in this area.

Inspired by the sparse coding paradigm, here we discuss a method for signal approximation, which is based on the correlation of the target signal with a pseudo-random signal, and uses a modification of the greedy matching pursuit algorithm. We show that this approach provides an efficient encoding-decoding method, with good computational speed and low memory requirements, which also can be used for lossy compression and encryption purposes.

*Institute for Space Imaging Science, University of Calgary, 2500 University Drive NW, Calgary, Alberta, T2N 1N4, Canada, Email: mandrecu@ucalgary.ca. Manuscript submitted March 21, 2011.

2 Sparse approximation problem

The classical signal approximation problem seeks to represent an arbitrary signal by the best approximation using a restricted class of signals. The general formulation is as follows [8]. Consider $\mathcal{H}$ is an $N-$dimensional Hilbert space (or more generally a Banach space), with the norm defined in terms of the inner product as:

$$\|x\|_2 = \langle x, x\rangle^{1/2} = \left[\sum_{n=0}^{N-1} x_n^2\right]^{1/2}, \ \forall x \in \mathcal{H}, \tag{1}$$

and an M-dimensional subspace $U \subset \mathcal{H}$, $M \leq N$, spanned by the orthonormal basis:

$$\{u^{(0)}, u^{(1)}, ..., u^{(M-1)}\} \in U : \tag{2}$$

$$\left\langle u^{(i)}, u^{(j)} \right\rangle = \delta(i,j) = \left\{ \begin{array}{ll} 1 & if \quad i = j \\ 0 & if \quad i \neq j \end{array} \right. , \tag{3}$$

$$\forall y \in U, \ \exists c_m, \ m = 0, ..., M-1, \ y = \sum_{m=0}^{M-1} c_m u^{(m)}. \tag{4}$$

Given a vector $x \in \mathcal{H}$, the problem is to find the vector $y \in U$ such that:

$$y = \arg\min_{y \in U} \|x - y\|_2 . \tag{5}$$

According to the best approximation theorem, this problem has a unique solution:

$$y = \sum_{m=0}^{M-1} c_m u^{(m)}, \tag{6}$$

where

$$c_m = \left\langle x, u^{(m)} \right\rangle, \ m = 0, 1, ..., M-1, \tag{7}$$

are the Fourier coefficients.

The sparse approximation problem is different than the classical one, since we do not seek a representation in an orthonormal basis, but on a dictionary $\Phi \in \mathcal{H}$, such that $span(\Phi) = \mathcal{H}$ [1]-[5]. In general, we consider a countable and overcomplete dictionary of functions:

$$\Phi = \{\varphi^{(m)} | m = 0, 1, ..., M-1; M \geq N\} \tag{8}$$

in $\mathcal{H}$, which are normalized ($\left\|\varphi^{(m)}\right\|_2 = 1$), but not orthogonal, and possibly redundant. Given $x \in \mathcal{H}$, the sparse approximation problem consists in finding the sparse coefficients vector $c \in \mathbb{R}^M$, such that:

$$c = \arg\min_{c \in \mathbb{R}^M} \|c\|_0, \quad \left\| x - \sum_{m=0}^{M-1} c_m \varphi^{(m)} \right\|_2 \leq \varepsilon, \tag{9}$$

where $\varepsilon \geq 0$ is a small positive constant, and:

$$\|c\|_0 = \sum_{m=0}^{M-1} [1 - \delta(c_m, 0)], \tag{10}$$

is the ℓ_0 norm, measuring the number of nonzero coefficients. Obviously, when using an overcomplete dictionary we have more vectors, and thus a better probability of finding a small number of vectors that approximate well the given vector. However, since the dictionary may contain linearly dependent vectors, such an expansion is no longer unique. Also, in a lossy compression framework the goal is to use as few vectors as possible, in order to obtain a good approximation. Unfortunately, this is an NP-hard combinatorial optimization problem, since in order to find the optimal expansion it is necessary to try all possible combinations:

$$\left(\begin{array}{c} M \\ K \end{array} \right) = \frac{M!}{K!(M-K)!}, \tag{11}$$

searching for the smallest collection of K non-zero terms which best approximates the signal.

Several methods have been developed to solve the sparse approximation problem. The standard approach is based on the convexification of the objective function, obtained by replacing the ℓ_0 norm with the ℓ_1 norm [1]-[5]:

$$\|c\|_1 = \sum_{m=0}^{M-1} |c_m|. \tag{12}$$

The resulting optimization problem:

$$c = \arg\min_{c \in \mathbb{R}^M} \|c\|_1, \quad \left\| x - \sum_{m=0}^{M-1} c_m \varphi^{(m)} \right\|_2 \leq \varepsilon, \tag{13}$$

is known as Basis Pursuit (BP), and it can be solved using linear programming techniques whose computational complexities are polynomial [1]. However, in most real applications the BP approach requires the solution of a very large convex, non-quadratic optimization problem, and therefore suffers from high computational complexity. Another approach is based on greedy algorithms, which are suboptimal and require far less computations. Our goal is not only to obtain a good sparse expansion, but also to provide a fast computational method, therefore here we focus our attention on the greedy Matching Pursuit (MP) algorithm [6]-[7], which is the fastest known algorithm for the sparse approximation problem. Also, since we are interested in developing a compression scheme, where only maximum $K \leq N$ out of M dictionary elements can be used in the expansion, we reformulate the problem as following:

$$c = \arg\min_{c \in \mathbb{R}^M} \left\| x - \sum_{m=0}^{M-1} c_m \varphi^{(m)} \right\|_2, \quad \|c\|_0 \leq K. \tag{14}$$

3 Random dictionary

Previous studies have shown that the sparsity of the approximation depends on using an appropriate dictionary for the given class of signals [1]-[6]. For example, multiscale decompositions of natural images into discrete cosine or wavelet bases are quasi-sparse. Such decompositions have a few significant coefficients, which concentrate most of the energy and information. This energy compaction property is then exploited in compression and denoising applications, where the weak coefficients are usually discarded. Thus, a method to construct overcomplete dictionaries consists by concatenating orthonormal bases like: Fourier, Gabor functions, wavelets, cosine packets etc. These dictionaries can be improved by employing learning methods [9], which adapt an initial dictionary to a set of training samples. In this case the goal is to optimize a dictionary, such that a given class of signals, has a sparse approximation.

The sparse decomposition abilities of such dictionaries are characterized by the restricted isometry property (RIP) of the $N \times M$ matrix $\hat{\Phi}$, with the columns given by the dictionary atoms $\varphi^{(m)}$. The K-restricted isometry constant δ_K of $\hat{\Phi}$ is the smallest quantity such that for every K-sparse vector $c \in \mathbb{R}^M$ we have [2]-[4]:

$$(1 - \delta_K) \|c\|_2^2 \leq \left\| \hat{\Phi} c \right\|_2^2 \leq (1 + \delta_K) \|c\|_2^2. \tag{15}$$

This means that every set of less than K columns are approximately orthogonal. Smaller δ_K means better orthogonality, and therefore a better discrimination capability of atoms. Recently it has been shown that random matrices satisfy the RIP with high probability [2]-[4]. Therefore, some good examples of overcomplete dictionaries include:

- matrices of independent and identical distributed (i.i.d) Gaussian samples from $N(0, 1)$;

- matrices with Bernoulli entries, where $\varphi_{nm} = \pm 1$ with equal probability $p = 1/2$;

- matrices with randomly sampled Fourier elements etc.

Inspired by the above results obtained for random matrices, here we propose a simplified approach. Instead of using a large random dictionary, we simply use a random vector of length $N + M$:

$$f = [f_0, f_1, ..., f_{M-1+N}]^T \in \mathbb{R}^{N+M}, \tag{16}$$

generated by a reproducible random process, a pseudo-random number generator for example. An atom of this dictionary will simply be a normalized "window" vector:

$$\varphi^{(m)} = \frac{[f_m, f_{m+1}, ..., f_{m+N-1}]^T}{\sqrt{\sum_{n=0}^{N-1} f_{m+n}^2}} \in \mathbb{R}^N. \tag{17}$$

Obviously, there are M such window vectors in any realization of the pseudo-random process, and they are uncorrelated since f_m are i.i.d random variables (depending on the quality of the random number generator used). Such a vector can be easily generated by encoder and decoder using only a given seed value. This way we avoid the high memory and computational requirements, while still obtaining a good sparse decomposition, which we will show that it can be also efficiently used for lossy compression. Without loosing generalization, in order to obtain an easy normalization we consider that each f_m is a Bernoulli random variable. Thus, the normalization is easily achieved by simply dividing the vector f with $\sqrt{N}$. We should mention that the mean $\langle x \rangle$ of the signal can be captured correctly if we assume that the components of first atom of the dictionary are all set to 1:

$$\varphi^{(0)} = \frac{1}{\sqrt{N}}[1, 1, ..., 1]^T. \tag{18}$$

This dictionary choice also provides a simple encryption scheme, since for different seeds one obtains different dictionaries. Therefore, if the seed is user defined then the obtained expansion will also be encrypted, since the same secret seed is needed for decoding.

4 Matching pursuit

Matching Pursuit is a well known greedy algorithm widely used in approximation theory and statistics [6]-[7]. One of its main features is that it can be applied to arbitrary dictionaries. Starting from an initial approximation $c = 0$ and residual $r = x$, the algorithm uses an iterative greedy strategy to pick the dictionary atoms that are the most strongly correlated with the residual. Then, successively their contribution is subtracted from the residual, which this way can be made arbitrarily small. Using the simplified dictionary f, the pseudo-code of the MP algorithm takes the form listed in Algorithm 1.

Algorithm 1. Matching Pursuit (MP)

K; // number of atoms in the approximation

$c \leftarrow 0$; // coefficients of selected atoms

$p \leftarrow 0$; // positions of selected atoms

$r \leftarrow x$; // initial residual

for$(k = 0, 1, ..., K-1)\{$

  $s_{max} \leftarrow 0$;

  for$(m = 0, 1, ..., M-1)\{$

    $s \leftarrow \langle r, \varphi^{(m)} \rangle$;

    if$(|s| > |s_{max}|)\{$

      $s_{max} \leftarrow s$;

      $i \leftarrow m;\}\}$

  $p_k \leftarrow i$;

  $c_k \leftarrow s_{max}$;

  $r \leftarrow r - c_k \varphi^{(p_k)};\}$

return p, c;

Thus, at each iteration step $k = 0, 1, ..., K-1$ the algorithm selects the index p_k of the atom $\varphi^{(p_k)}$, which has the highest correlation with the current residual, and updates the estimate of the corresponding coefficient c_k, and the residual r. After K selection steps the algorithm returns the positions p of the selected atoms in the dictionary and their corresponding coefficients c in the expansion. A shortcoming of the MP algorithm is that although the asymptotic convergence is guaranteed and it can be easily proved, the resulting approximation after any finite number of steps $K \leq N$ will in general be suboptimal. Thus, one cannot expect an exact reconstruction of the target signal after decoding.

One can see that the decoding step requires both the positions p and the coefficients c of the selected atoms in order to compute the K-term approximation:

$$y \leftarrow \sum_{k=0}^{K-1} c_k \varphi^{(p_k)} \simeq x. \tag{19}$$

This is inconvenient from the point of view of compression. Assuming for example that the elements of the input vector $x \in \mathbb{R}^N$ are floating point numbers represented on Q bits, we need NQ bits to store the whole

vector. Thus, in order to compress x we need to reduce this number. For each position we need Q bits, and for each coefficient we also need Q bits, therefore the output of the MP algorithm requires $2KQ$ bits. Thus, in order to achieve compression we must have $K < N/2$. This condition can be relaxed by imposing that both p_k and c_k are stored together as a single number represented on Q bits. Of course, one may think that in this case the precision of c_k will be affected, and the approximation will deteriorate significantly. However, due to the random characteristic of the dictionary we may expect that this may actually work, and at each step the algorithm will pick the atom with the best pair (p_k, c_k) which can be "accommodated" on a number h_k represented on Q bits. Thus, instead of using $2Q$ bits to store a pair (p_k, c_k), we may actually use only Q bits to store their equivalent h_k. The question is how to do this efficiently?

We observe that we need to find the atom characterized by a pair (p_k, c_k), where there is a trade-off between the necessary precision of c_k and the length of p_k, such that they can be represented together on Q bits, and their inclusion in the approximation expansion decreases the residual r. Ideally, we would like to allocate $Q/2$ bits for the position p_k, and $Q/2$ bits for the corresponding coefficient c_k. That means to restrict the length of the dictionary to $M = 2^{Q/2}$. Thus, for a typical integer representation on $Q = 32$ bits, the dictionary will contain $M = 2^{16}$ elements.

Now, let us assume the signal is normalized $x \leftarrow x/\|x\|$ before the compression. Since the MP algorithm will always produce a residual r with $\|r\| < \|x\| = 1$, the result of the correlation term $s \leftarrow \langle r, \varphi^{(m)} \rangle$, will always be bounded: $s \in (-1, 1)$. Thus, the value of a resulted coefficient will also be bounded by the same interval: $c_k \in (-1, 1)$. Finally, the idea is to make the position p_k equal with the integer part of h_k, and the coefficient c_k equal with the fractional part of h_k:

$$int(h_k) \leftarrow p_k, \quad frac(h_k) \leftarrow c_k. \tag{20}$$

Obviously, by doing this some of the precision in the representation of c_k will be lost. The compressive MP (CMP) algorithm which takes into account these modifications is listed in Algorithm 2.

One can see that the computation of the cross-correlation term s requires two extra steps. In the first step, the index m of the currently tested atom $\varphi^{(m)}$, and its correlation $s \in (-1, 1)$ with the residual, are packet together in s, using:

$$s \leftarrow sign(s)m + s. \tag{21}$$

This will result in a loss of the precision of the fractional part of s, since it needs to accommodate also the integer part m, on the same number Q of bits. In the second step we extract the resulted fractional part using:

$$s \leftarrow sign(s)(|s| - |int(s)|), \tag{22}$$

in order to perform the comparison with the current maximum value $|s_{max}|$. If the test is true, then we store the obtained values in $s_{max} \leftarrow s$, and respectively $i \leftarrow m$. The values s_{max} and i, corresponding to the the best atom are then used to update the residual, and they are packed into h_k, this time without information loss. Thus, after the first packing step, when some precision is lost, the future packing-unpacking steps become reversible, and the precision is conserved.

Algorithm 2. Compressive Matching Pursuit (CMP)

K; // number of atoms in the approximation

$h \leftarrow 0$; // positions and coefficients of selected atoms

$h_K \leftarrow \|x\|_2$; // signal normalization

$r \leftarrow x/h_K$; // initial residual

for$(k = 0, 1, ..., K-1)\{$

  $s_{max} \leftarrow 0$;

  for$(m = 0, 1, ..., M-1)\{$

    $s \leftarrow \langle r, \varphi^{(m)} \rangle$;

    $s \leftarrow sign(s)m + s$;

    $s \leftarrow sign(s)(|s| - |int(s)|)$;

    if$(|s| > |s_{max}|)\{$

      $s_{max} \leftarrow s$;

      $i \leftarrow m;\}\}$

  $r \leftarrow r - s_{max}\varphi^{(i)}$;

  $h_k \leftarrow sign(s_{max})i + s_{max};\}$

return h, ε;

We should also save the norm of x, which is required for the decoding step: $h_K = \|x\|_2$. Therefore, the length of the vector h is $K+1$, where the first K values correspond to the positions (the positive integer part) and coefficients (the fractional part) of the atoms selected in the approximation expansion, while the last value contains the norm of the input signal. The decoding procedure is very fast and it is done as following:

$$y \leftarrow h_K \sum_{k=0}^{K-1} sign(h_k)(|h_k| - |int(h_k)|)\varphi^{(|int(h_k)|)} \simeq x. \tag{23}$$

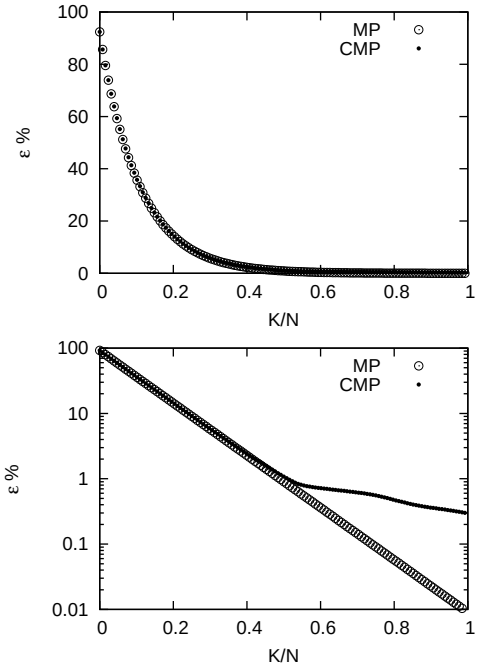

Figure 1: Relative approximation error ϵ of the MP and CMP algorithms.

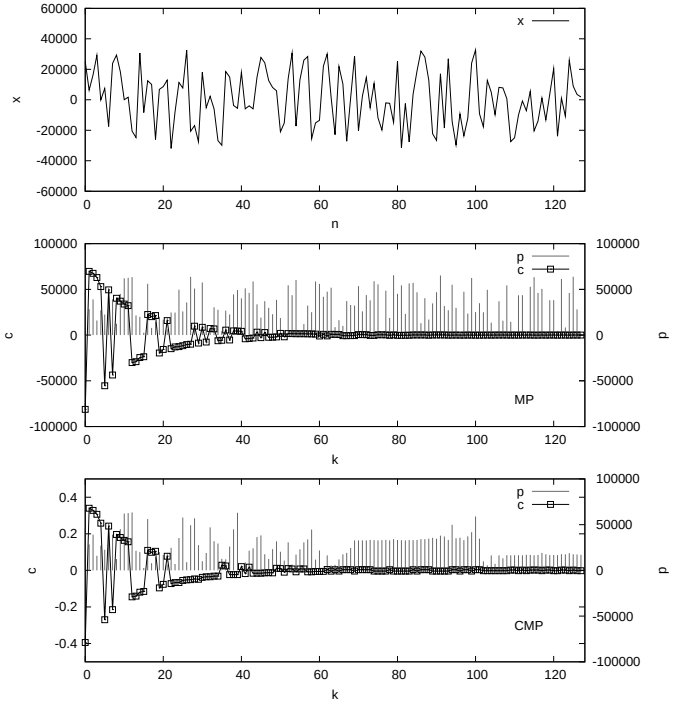

Figure 2: Decomposition of a random signal using MP and CMP algorithms.

5 Numerical results

We have implemented the MP and CMP algorithms in parallel using C, OpenMP and GCC on a Linux platform (see the last section "reproducible research"). The parallel implementation is almost n times faster than the serial version, where n is the number of available CPU cores.

In the following experiments we consider a random Bernoulli dictionary with $M = 2^{16}$ elements. Also, in order to speed up computation we set the length N of the input signal to $N = 128$. A longer input signal can be easily divided in chunks of length $N \leq 128$, which can be then independently processed. Ideally, we would like to approximate and compress any kind of signal, so the shape of the signal doesn't really matter. Therefore, for testing we choose random samples drawn from a uniform distribution on the interval (-2^{15},2^{15}), represented as floating point numbers on $Q = 32$ bits. This, means that we practically approximate and compress noise. This is usually a difficult task, since such signals will have the widest possible bandwidth, for the considered finite length N of the samples. The quantity of interest is the relative recovery error, which is defined as following:

$$\varepsilon = 100 \frac{\|x - y\|_2}{\|x - \langle x \rangle\|_2}, \tag{24}$$

where $\langle x \rangle$ is the signal mean value, and y is the K-term approximation expansion. In Figure 1 we give the value of ε as a function of the inverse of compression ratio $\rho^{-1} = K/N$ (linear scale - top, logarithmic scale - bottom). The results were averaged over 1000 samples for $K = 1, ..., N$. One can see that the effect of the CMP packing procedure $(p, c) \rightarrow h$ manifests only for $K/N > 0.5$, which is obviously not too bad for lossy compression purposes. Also, we have $\varepsilon(0.5) \simeq 1\%$, which means that a compression ratio of $\rho_{CMP} = 2 : 1$ produces a distortion of the data of only 1%.

In Figure 2 we give a typical sample, and its decomposition by the MP and CMP algorithms. Here we have represented the values of the coefficients c_k and the positions p_k of the corresponding atoms in the dictionary. In the case of CMP, the positions and the coefficients are extracted from h_k, using: $p_k = int(h_k)$, and $c_k = sign(h_k)(|h_k| - |p_k|)$. The positions and the coefficients for MP and CMP are different, due to the extra packing constraint in the CMP algorithm. Also, we should notice the exponential decrease of the magnitude of the coefficients c_k. Thus, a lossy compression of the signal can be achieved retaining only the first coefficients with large values.

In Figure 3 we have the same signal from Figure 2, and its recovery after lossy compression, for several different compression ratios: $\rho_{CMP} = 16 : 1$, $\varepsilon = 51.30\%$; $\rho_{CMP} = 8 : 1$, $\varepsilon = 28.25\%$; $\rho_{CMP} = 4 : 1$, $\varepsilon = 9.14\%$; and respectively $\rho_{CMP} = 2 : 1$, $\varepsilon = 1.01\%$. Similar results have been obtained for different types of signals. For example, in Figure 4 we consider a smooth signal (a superposition of sinusoids) for several compression ratios: $\rho_{CMP} = 8 : 1$, $\varepsilon = 27.83\%$; $\rho_{CMP} = 4 : 1$, $\varepsilon = 8.92\%$; and respectively $\rho_{CMP} = 2 : 1$, $\varepsilon = 0.98\%$.

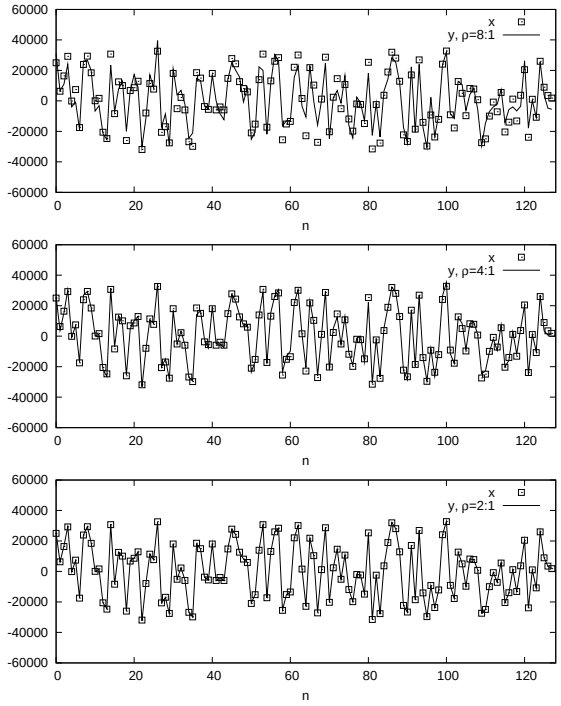

Figure 3: Lossy compression of a random signal.

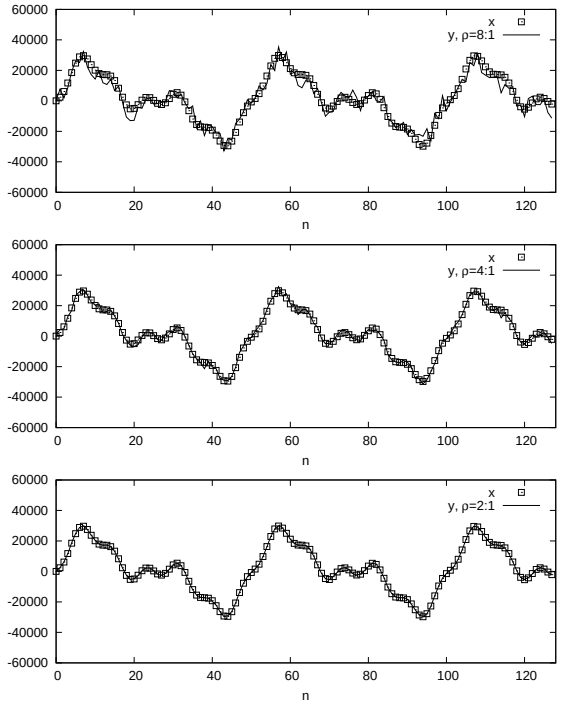

Figure 4: Lossy compression of a smooth signal.

6 Conclusion

We have discussed a sparse random approximation method, based on the correlation of the target signal with a pseudo-random signal, and a modification of the greedy matching pursuit algorithm. We have shown that this approach provides an efficient encoding-decoding method. Also, the presented method has the advantage of an easy implementation, with high computational speed and low memory requirements, which also can be used for lossy compression and encryption purposes.

7 Reproducible research

In support to the above analytical considerations, here we give the parallel C and OpenMP code of the Matching Pursuit and Compressive Matching Pursuit algorithms. The code was developed on an Ubuntu Linux 10.04 LTS platform, and requires GCC version 4.4.3, or higher.

```
/*
Matching Pursuit and Compressive Matching Pursuit
Copyright (C) 2011, M. Andrecut, mandrecu@ucalgary.ca
Institute for Space Imaging Science
University of Calgary, Alberta, Canada

This program is free software: you can redistribute it
and/or modify it under the terms of the GNU General
Public License as published by the Free Software
Foundation, either version 3 of the License, or any
later version (see: http://www.gnu.org/licenses/).
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>


float rnd(void)
{
/*
Computes a random number in (-1,1)
*/
return 2.0*rand()/(RAND_MAX + 1.0) - 1.0;
}


float relative_error(int P, int N, float *x, float *y)
{
/*
Computes relative error of two vectors x, y of length N:
rel_err = |x - y|/|x - <x>|
*/
int n;
float mean = 0, normx = 0, normxy = 0, tmp, re;

#pragma omp parallel for reduction(+:mean) if(P)
for(n=0; n<N; n++)
    {
    mean += x[n];
    }
mean = x[n]/N;
#pragma omp parallel for reduction(+:normx) if(P)
for(n=0; n<N; n++)
    {
    tmp = x[n] - mean;
```

```
    normx += tmp*tmp;
    }
#pragma omp parallel for reduction(+:normxy) if(P)
for(n=0; n<N; n++)
    {
    tmp = x[n] - y[n];
    normxy += tmp*tmp;
    }
return 100*sqrt(normxy)/sqrt(normx);
}


void init_signal(int P, int N, float *x)
{
/*
Computes the input signal x, a random vector of length N.
*/

int n;

#pragma omp parallel for if(P)
for(n=0; n<N; n++)
    {
    x[n] = 32768*rnd();
    }
}


void init_dictionary(int P, int M, int N, float *f)
{
/*
Computes the 'dictionary' f,
random Bernoulli vector of length M + N.
*/
int n, m;
float tmp, sN = 1.0/sqrt(N);

#pragma omp parallel \
for default(none) private(n) shared(N,f,sN) if(P)
for(n = 0; n < N; n++)
    {
    f[n] = sN;
    }
#pragma omp parallel \
for default(none) private(m,tmp) shared(M,N,f,sN) if(P)
for(m = N; m < M+N; m++)
    {
    tmp = rnd();
    f[m] = sN*((tmp>=0)-(tmp<0));
    }
}


void mp_encode(int P, int N, float *x, int M, \
float *f, int K, int *p, float *c)
{
/*
Encodes the input signal x of length N, using the
dictionary f of length M + N, using the Matching Pursuit
(MP) algorithm. Outputs the positions p and the
corresponding coefficients c (vectors of lenght K)
of the atoms selected from the dictionary.
*/
int i, n, m, k;
float smax, tmp;
float *r = (float*)malloc(N * sizeof(float));
float *s = (float*)malloc(M * sizeof(float));

#pragma omp parallel \
for default(none) private(n) shared(N,r,x) if(P)
for(n=0; n<N; n++)
    {
    r[n] = x[n];
    }
for(k=0; k<K; k++)
    {
    #pragma omp parallel \
    for default(none) private(m,n) \
    shared(M,N,s,f,r) if(P)
    for(m=0; m<M; m++)
        {
        s[m] = 0;
        for(n=0; n<N; n++)
            {
            s[m] += f[m+n]*r[n];
            }
        }
    smax = 0;
    for(m=0; m<M; m++)
        {
        if(fabs(s[m]) > fabs(smax))
            {
            smax = s[m];
            i = m;
            }
        }
    #pragma omp parallel \
    for default(none) private(n) \
    shared(N,f,r,smax,i) if(P)
    for(n=0; n<N; n++)
        {
        r[n] -= smax*f[i+n];
        }
    p[k] = i;
    c[k] = smax;
    }
free(r); free(s);
}


void cmp_encode(int P, int N, float *x, \
int M, float *f, int K, float *h)
{
/*
Encodes the input signal x of length N, using the
dictionary f of length M + N, using the Compressive
Matching Pursuit (CMP) algorithm. Outputs h, the
positions and the corresponding coefficients (vector of
lenght K) of the atoms selected from the dictionary.
*/
int i, n, m, k, t, p, j;
float smax, norm, c, q;
float *r = (float*)malloc(N * sizeof(float));
float *s = (float*)malloc(M * sizeof(float));

#pragma omp parallel \
for default(none) private(k) shared(K,h) if(P)
for(k=0; k<K; k++)
    {
    h[k] = 0;
    }
#pragma omp parallel \
for reduction(+:norm) if(P)
for(n=0; n<N; n++)
    {
    norm += x[n]*x[n];
    }
h[K] = sqrt(norm);
#pragma omp parallel \
for default(none) private(n) shared(N,r,x,h,K) if(P)
for(n=0; n<N; n++)
    {
    r[n] = x[n]/h[K];
    }
```

```
for(k=0; k<K; k++)
    {
    p = fabs(h[k]);
    c = ((h[k]>=0)-(h[k]<0))*(fabs(h[k]) - p);
    #pragma omp parallel \
    for default(none) private(n) shared(N,f,r,c,p) if(P)
    for(n=0; n<N; n++)
        {
        r[n] +=c*f[p+n];
        }
    #pragma omp parallel \
    for default(none) private(m,n) \
    shared(M,N,s,f,r,c,p,k) if(P)
    for(m=0; m<M; m++)
        {
        s[m] = 0;
        for(n=0; n<N; n++)
            {
            s[m] += r[n]*f[m+n];
            }
        s[m] = ((s[m]>=0)-(s[m]<0))*m + s[m];
        }
    smax = 0;
    for(m=0; m<M; m++)
        {
        j = fabs(s[m]);
        q = ((s[m]>=0)-(s[m]<0))*(fabs(s[m]) - j);
        if(fabs(q) > fabs(smax))
            {
            smax = q;
            i = j;
            }
        }
    #pragma omp parallel \
    for default(none) private(n) \
    shared(N,f,r,smax,i,c,p,k) if(P)
    for(n=0; n<N; n++)
        {
        r[n] -= smax*f[i+n];
        }
    h[k] = s[i];
    }
free(r); free(s);
}


void mp_decode(int P, int M, float *f, \
int K, int *p, float *c, int N, float *y)
{
/*
Decodes the previously MP encoded input signal x.
Outputs the approximation y of x, using p and c.
*/
int n, k;

#pragma omp parallel \
for default(none) private(n,k) shared(K,N,y,c,p,f) if(P)
for(n = 0; n<N; n++)
    {
    y[n] = 0;
    for(k=0; k<K; k++)
        {
        y[n] += c[k]*f[p[k]+n];
        }
    }
}


void cmp_decode(int P, int M, float *f, \
int K, int KK, float *h, int N, float *y)
{
/*
Decodes the previously CMP encoded input signal x.
Outputs the approximation y of x, using h.
*/
int n, k, p;
float c;

#pragma omp parallel \
for default(none) private(n,k) \
shared(K,KK,N,y,c,p,f,h) if(P)
for(n = 0; n<N; n++)
    {
    y[n] = 0;
    for(k=0; k<KK; k++)
        {
        p = fabs(h[k]);
        c = ((h[k]>=0)-(h[k]<0))*(fabs(h[k]) - p);
        y[n] += h[K]*c*f[p+n];
        }
    }
}


void plot_results(int N, int K, float *x, \
float rerr_mp, float rerr_cmp)
{
/*
Plot the results. Requires Gnuplot.
*/
int n;
float maxx = 0;

for(n=0; n<N; n++)
    {
    if(fabs(x[n]) > maxx)
        {
        maxx = fabs(x[n]);
        }
    }
FILE *pipe = popen("gnuplot -persist","w");
fprintf(pipe,"set multiplot layout 2,1\n");
fprintf(pipe,"set xrange[0:%d]\n", (int)(N*1.3));
fprintf(pipe,"set yrange[%f:%f]\n", -maxx, maxx);
fprintf(pipe,"set size 1.0, 0.5\n");
fprintf(pipe,"set title 'MP, error = %f %%, \
              compression = %f'\n", rerr_mp, N/(2.0*K));
fprintf(pipe,"set origin 0.0, 0.5\n");
fprintf(pipe, "plot 'mp_results.txt' \
              u 1:2 with lines lc rgb 'red' title 'x', \
              'mp_results.txt' u 1:3 with lines \
              lc rgb 'blue' title 'y', \
              'mp_results.txt' u 1:($2-$3) with lines \
              lc rgb 'green' title 'x-y'\n");

fprintf(pipe,"set title 'CMP, error = %f %%, \
             compression = %f'\n", rerr_cmp, N/(1.0*K));
fprintf(pipe,"set origin 0.0, 0.0\n");
fprintf(pipe, "plot 'cmp_results.txt' u 1:2 with lines \
              lc rgb 'red' title 'x', \
              'cmp_results.txt' u 1:3 with lines \
              lc rgb 'blue' title 'y', \
              'cmp_results.txt' u 1:($2-$3) with lines \
              lc rgb 'green' title 'x-y'\n");
fprintf(pipe,"unset multiplot\n");
fclose(pipe);
}


int main(int argc, char **argv)
{
/*
Example of encoding-decoding.
*/
```

```
int n;
double start_t, end_t;

srand(time(NULL));

printf("Running MP-CMP with the parameters:\n");

/* Initialization */
int P = atoi(argv[1]);
printf("P = %d\n", P);
int V = atoi(argv[2]);
printf("V = %d\n", V);
int M = atoi(argv[3]);
printf("M = %d\n", M);
int N = atoi(argv[4]);
printf("N = %d\n", N);
int K = atoi(argv[5]);
printf("K = %d\n", K);

float *f = (float*)malloc((M + N) * sizeof(float));
float *x = (float*)malloc(N * sizeof(float));
float *y = (float*)malloc(N * sizeof(float));
float *c = (float*)malloc(K * sizeof(float));
int *p = (int*)malloc(K * sizeof(int));
float *h = (float*)malloc((K + 1) * sizeof(float));

/* OpenMP test*/
if(P)
    {
    int nthreads;
    #pragma omp parallel
        {
        nthreads = omp_get_max_threads();
        }
    printf("Parallel computation, \
            number of threads = %d\n", nthreads);
    }
else
    {
    printf("Serial computation, number of threads=1\n");
    }

start_t = omp_get_wtime();
init_signal(P, N, x);
init_dictionary(P, M, N, f);
end_t = omp_get_wtime();
printf("Initialization time: %f s\n", end_t - start_t);
printf("MP-CMP results:\n");

/* MP encoding-decoding section */
start_t = omp_get_wtime();
mp_encode(P, N, x, M, f, K, p, c);
end_t = omp_get_wtime();
printf("MP encoding time: %f s\n", end_t - start_t);
start_t = omp_get_wtime();
mp_decode(P, M, f, K, p, c, N, y);
end_t = omp_get_wtime();
printf("MP decoding time: %f s\n", end_t - start_t);
float rerr_mp = relative_error(P, N, x, y);
printf("MP approximation error: %f %%\n", rerr_mp);

/* Save MP results */
FILE *file_mp = fopen("mp_results.txt", "w");
for(n = 0; n < N; n++)
    {
    fprintf(file_mp, "%d\t%f\t%f\n", n, x[n], y[n]);
    }
fclose(file_mp);

/* CMP encoding-decoding section */
start_t = omp_get_wtime();
cmp_encode(P, N, x, M, f, K, h);
end_t = omp_get_wtime();
printf("CMP encoding time: %f s\n", end_t - start_t);
start_t = omp_get_wtime();
cmp_decode(P, M, f, K, K, h, N, y);
end_t = omp_get_wtime();
printf("CMP decoding time: %f s\n", end_t - start_t);
float rerr_cmp = relative_error(P, N, x, y);
printf("CMP approximation error: %f %%\n", rerr_cmp);

/* Save CMP results */
FILE *file_cmp = fopen("cmp_results.txt", "w");
for(n = 0; n < N; n++)
    {
    fprintf(file_cmp, "%d\t%f\t%f\n", n, x[n], y[n]);
    }
fclose(file_cmp);

/* Plot results (requires Gnuplot)*/
if(V) plot_results(N, K, x, rerr_mp, rerr_cmp);

/* Clean memory and exit*/
free(f);
free(x);
free(y);
free(c);
free(p);
free(h);
return 0;
}
```

The visualization requires a proper installation of Gnuplot (http://www.gnuplot.info/). To compile and run the above program use the following script:

```
#!/bin/bash
#  Matching Pursuit and Compressive Matching Pursuit
#  Copyright (C) 2011, M. Andrecut, mandrecu@ucalgary.ca
#  Institute for Space Imaging Science
#  University of Calgary, Alberta, Canada

 clear

# Compile cmp

 gcc -g -O3 -ffast-math -fopenmp cmp.c -o cmp

# Parallel (P=1) or serial (P=0) computation

 P="1"

# Gnuplot visualization (V=1) or no visualization (V=0)

 V="1"

# Dictionary length

 M="65536"

# Signal length

 N="128"

# Approximation expansion lenght (uncomment K)

 K="64"   # 2:1 compression
#K="32"   # 4:1 compression
#K="16"   # 8:1 compression

# Run cmp with the above parameters
./cmp $P $V $M $N $K
```

References

[1] Chen, S.; Donoho, D.; Saunders, M. "Atomic Decomposition by Basis Pursuit." *SIAM Review*, 43, 129-159, 2001.

[2] Donoho, D.; Tanner, J. "Sparse nonnegative solutions of underdetermined linear equations by linear programming." *Proc. Nat. Acad. of Sci.*, 102(27), 9446-9451, 2005.

[3] Donoho, D. "Compressed Sensing." *IEEE Trans. Inf. Theory*, V52, 1289-1306, 2006.

[4] Candes, E.; Tao, T. "Near Optimal Signal Recovery from Random Projections: Universal Encoding Strategies?" *IEEE Trans. Inf. Theory*, V52, 5406-5425, 2006.

[5] Andrecut, M. "Stochastic Recovery Of Sparse Signals From Random Measurements." *Engineering Letters*, V19:1, EL-19-1-01, 2010.

[6] Mallat, S.; Zhang, Z. "Matching Pursuit in a Time-Frequency Dictionary." *IEEE Trans. Signal Process.*, V41, 3397-3415, 1993.

[7] Andrecut, M. "Fast GPU implementation of sparse signal recovery from random projections". *Engineering Letters*, V17:3, EL-17-3-01, 2009.

[8] Deutsch, F.R. *Best Approximation in Inner Product Spaces*. CMS Books in Mathematics, Springer, New York, 2001.

[9] Aharon, M.; Elad, M.; Bruckstein, A. "K-SVD: An Algorithm for Designing of Overcomplete Dictionaries for Sparse Representation." *IEEE Trans. Signal Process.*, 54(11), 4311-4322, 2006.