Efficient Restoration by Compression

Nevo Agmon, Danny Priymak, Yuval Shildan
Under the supervision of Yehuda Dar

Geometric Image Processing Laboratory, Faculty of Computer Science
Technion - Israel Institute of Technology

Abstract

When dealing with signal compression, most compression algorithms optimize the reconstructed output with respect to the acquired input signal. A more general approach would be optimizing the compression algorithm with respect to the signal prior to the acquisition phase as the effective input, such that the final decompressed output is optimal. Ideally, this approach could utilize knowledge about specific acquisition devices to further optimize the compression, i.e. given a known degradation model, the presented approach could yield a highly optimized compressed result, which can be either used to transmit a signal of higher quality over the same infrastructure or alternatively, deliver the same quality using fewer resources.

Dar et al. [1] proposed an algorithm for joint restoration and compression of images, on which we rely in this work. This algorithm was implemented in MATLAB, which, due to MATLAB's overhead, opened up the possibility of significant efficiency improvements. For this reason, we chose to focus our efforts into optimizing the implementation runtime demands, while considering software engineering and object-oriented design as top priorities.

Contents

1 Proposed Method

Dar et al.'s mathematical formulation of this problem [1] has led to the conclusion that the presented technique for joint restoration and compression can be implemented as an iterative process of existing compression methods, with the addition of an optimization term.

What follows is a mathematical description of the problem settings, divided into phases, which will ultimately describe the thought process that has led to the translation of the problem to an iterative process of existing compression methods. A visual illustration of the entire imaging system can be seen in figure 1.

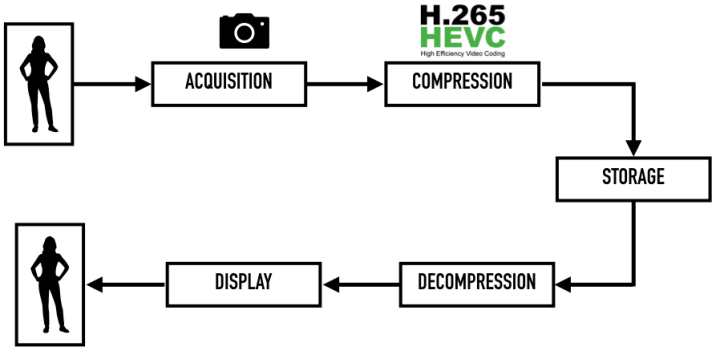

Figure 1: Imaging system visualization

Acquisition Phase. Consider a real and continuous source signal $x \in \mathbb{R}^M$ degraded via the acquisition degradation model

$$w = Ax + n \qquad w \in \mathbb{R}^M$$

where A is an $M \times N$ matrix, in our case here representing some blurring operator, and n is a white Gaussian noise vector of size M.

Compression Phase. Consider a compression operator $C : \mathbb{R}^M \longrightarrow \mathcal{B}$ where $\mathcal{B}$ is a discrete set of binary compressed representations. We denote the compressed form of the degraded signal w by $b \coloneqq C(w)$.

Decompression Phase. Consider a decompression operator $F : \mathcal{B} \longrightarrow \mathcal{S}$ where $\mathcal{S} \subset \mathbb{R}^M$ is a discrete set of decompressed signals. The set $\mathcal{S}$ is discrete since the domain of F domain $\mathcal{B}$ is discrete, and F is a real mapping. We denote the decompressed form of the signal b by $v \coloneqq F(b)$.

Lastly, we denote $R(v)$ as the binary length of $b = F^{-1}(v)$.

Using these three phases, we can mathematically describe our entire model as

$$y = F(C(Ax + n)) \qquad\qquad y \in \mathbb{R}^N$$

where y is the perceived output signal. The compression goal can be described as minimizing the bit-cost $R(v)$, while satisfying a demand on the maximal error allowed in the reconstructed signal with respect to the original one. A metric for evaluating the output signal y must consider the degraded signal w since x is unknown. A initial motivating distortion metric to consider could be

$$d_s(w, y) \coloneqq \frac{1}{M} \|w - Ay\|_2^2$$

which, given an ideal output $y_{\text{ideal}} = x$, yields

$$d_s(w, y_{\text{ideal}}) = \frac{1}{M} \|n\|_2^2 \approx \sigma_n^2$$

The following equations involve the image y using its decompressed form, which we've denoted by v. As stated earlier, the compressor's goal is to minimize the bit cost while retaining the signal quality. Minimizing the bit cost under the system distortion constraint can be described formally as

$$\begin{aligned} \widehat{v} &= \underset{v \in \mathcal{S}}{\operatorname{argmin}} R(v) \\ D_0 &\leq \frac{1}{M} \|w - Av\|_2^2 \leq D_0 + D \end{aligned}$$

for some minimal distortion level D_0 (defined in [1]) and a maximal level of allowed distortion D. Using the unconstrained Lagrangian optimization yields

$$\widehat{v} = \underset{v \in \mathcal{S}}{\operatorname{argmin}} R(v) + \lambda \frac{1}{M} \|w - Av\|_2^2$$

where λ is a Lagrange multiplier corresponding to some distortion level D_λ.

The last optimization is discrete, requiring the evaluation of the cost for all the candidates in $\mathcal{S}$, leading to impractical computational requirements for its direct treatment for high-dimensional signals (such as images) where $\mathcal{S}$ is typically a huge set. Therefore, a variable splitting technique can be used, which yields the optimization problem

$$(\widehat{v}, \widehat{z}) = \underset{v \in \mathcal{S}, z \in \mathbb{R}^M}{\operatorname{argmin}} R(v) + \frac{\lambda}{M} \|w - Az\|_2^2 \qquad \text{subject to } v = z$$

To solve this optimization problem, we can use the Augmented Lagrangian Method of Multipliers (ADMM) which, applied to our setting, yields

$$\begin{aligned} \left(\widehat{v}^{(t)}, \widehat{z}^{(t)}\right) &= \underset{v \in \mathcal{S}, z \in \mathbb{R}^M}{\operatorname{argmin}} R(v) + \frac{\lambda}{M} \|w - Az\|_2^2 + \frac{\beta}{2} \left\| v - z + u^{(t)} \right\|_2^2 \\ u^{(t+1)} &= u^{(t)} + \left(\widehat{v}^{(t)} - \widehat{z}^{(t)}\right) \end{aligned}$$

where $u^{(t)} \in \mathbb{R}^M$ is the scaled dual variable, and β is a parameter of the augmented Lagrangian. This is an iterative refinement method that optimizes the split variable.

We can then minimize each variable alternatively. This results in

$$\widehat{v}^{(t)} = \underset{v \in \mathcal{S}}{\operatorname{argmin}} R(v) + \frac{\beta}{2} \left\| v - \widetilde{z}^{(t)} \right\|_2^2 \tag{1}$$

$$\widehat{z}^{(t)} = \underset{z \in \mathbb{R}^M}{\operatorname{argmin}} \frac{\lambda}{M} \|w - Az\|_2^2 + \frac{\beta}{2} \left\| z - \widetilde{v}^{(t)} \right\|_2^2 \tag{2}$$

$$u^{(t+1)} = u^{(t)} + \left(\widehat{v}^{(t)} - \widehat{z}^{(t)}\right) \tag{3}$$

where

$$\widetilde{z}^{(t)} = \widehat{z}^{(t-1)} - u^{(t)}$$
$$\widetilde{v}^{(t)} = \widehat{v}^{(t)} + u^{(t)}$$

It can be observed that parts of the iterative model described above can be translated to the use of standard compression and decompression algorithms. Equation (1) is basically a compression optimization process that uses a standard squared error. Therefore, (1) can be replaced by an application of a standard compression algorithm of the form

$$b^{(t)} = \text{StandardCompress}\left(\widetilde{z}^{(t)}, \rho\right)$$
$$\widehat{v}^{(t)} = \text{StandardDecompress}\left(b^{(t)}\right)$$

In addition, equation (2) is an ℓ_2-constrained deconvolution optimization of simple quadratic terms.

To summarize, the described method is an iterative procedure comprised of three simpler procedures:compression, decompression (using standard compression and decompression methods), and deconvolution. Formally, the method is given by

$$b^{(t)} = \text{StandardCompress}\left(\widetilde{z}^{(t)}, \rho\right)$$
$$\widehat{v}^{(t)} = \text{StandardDecompress}\left(b^{(t)}\right)$$
$$\widehat{z}^{(t)} = \underset{z \in \mathbb{R}^M}{\text{argmin}} \frac{\lambda}{M} \|w - Az\|_2^2 + \frac{\beta}{2} \left\|z - \widetilde{v}^{(t)}\right\|_2^2$$
$$u^{(t+1)} = u^{(t)} + \left(\widehat{v}^{(t)} - \widehat{z}^{(t)}\right)$$

where

$$\widetilde{z}^{(t)} = \widehat{z}^{(t-1)} - u^{(t)}$$
$$\widetilde{v}^{(t)} = \widehat{v}^{(t)} + u^{(t)}$$

A visualization of this method can be seen in figure 2.

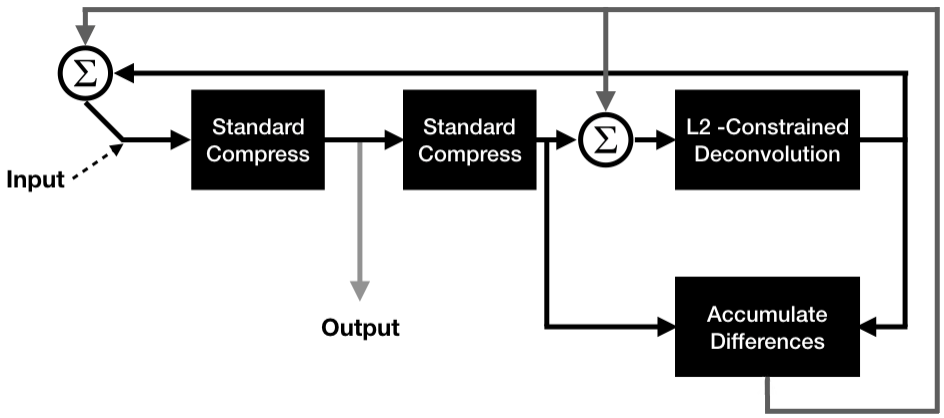

Figure 2: Algorithm visualization

2 Project Goals

Given the mathematical problem description above, we would like to have a working code implementation to test the theory in real life. Luckily, Dar et al. have provided a MATLAB implementation of the proposed algorithm as a proof of concept. As a result, there is room to provide a new implementation that greatly improves the current one, based on the following observations.

2.1 Observations

Runtime Performance. Seeing as the current implementation is written in MATLAB, execution time will inevitably be sub-optimal. This is mainly due to MATLAB's JRE (Java Runtime Environment) backend and the way it performs garbage collection. Due to this reason, common MATLAB implementations tend to disregard runtime efficiency optimization as a major concern.

Code Coupling. The code is implemented as a collection of MATLAB scripts, which in turn makes it very coupled. In addition, tweaking of inputs and parameters for testing and development purposes requires hard-coded changes to be made. Finally, as a proof of concept, the code is not intended to provide modularity, which poses difficulty when applying modifications during development and testing.

2.2 Suggested Improvements

Runtime Optimization

To facilitate runtime optimization, we first suggest using C++ as the implementation language, as C++ provides both low-level control and high-level architectural capabilities. A second suggestion is to provide a design that puts significant emphasis on runtime optimization.

Decoupling and Modularity

Production-ready software should ideally be distributed as a "black box", providing APIs for tweaking and adjustments. Such an application should ideally employ object oriented design, which decouples logical dependencies and allows for modularity.

3 Design Overview

The following sections describe our design and how it implements the suggestions presented above. The final algorithm we implemented is Algorithm 1, proposed by Dar et al [1], and presented below.

Algorithm 1 Restoration By Compression

Inputs: y, β, θ
Initialize $\widehat{z}^{(0)} = y$.
$t = 1$ and $u^{(1)} = 0$.
repeat
  $\widetilde{z}^{(t)} = \widehat{z}^{(t-1)} - u^{(t)}$
  Solve the ℓ_2-constrained deconvolution:
    $\widehat{x}^{(t)} = \text{argmin}_x \left\|Ax - y\right\|_2^2 + \frac{\beta}{2}\left\|x - \widetilde{z}^{(t)}\right\|_2^2$
  $\widetilde{x}^{(t)} = \widehat{x}^{(t)} + u^{(t)}$
  $\widehat{z}^{(t)} = \text{CompressDecompress}_\theta\left(\widetilde{x}^{(t)}\right)$
  $u^{(t+1)} = u^{(t)} + \left(\widehat{x}^{(t)} - \widehat{z}^{(t)}\right)$
  $t \longleftarrow t + 1$
until stopping criteria are satisfied

3.1 Software Design

In designing the implementation we chose to use C++, while utilizing object oriented design. To replace MATLAB's DSP functionality we used the OpenCV library.

The classes we chose to implement and their descriptions are as follows.

ImageWrapper. This class' main function is to hold an image as an OpenCV matrix and provide an API to interact with it. Its second role is to help with the analysis and evaluation of the program's behavior.

InputImageSimulator. This class generates a degraded image based on the previously described degradation model. The resulting degraded image is used as the algorithm's input, which it aspires to best restore. This is basically a simulation of the acquisition phase described earlier.

RestorationByCompression. This class provides the API that handles the algorithm implementation. Specifically, the `restoreImage` method that encapsulates the restoration.

3.2 Third-party Sources

DSP Library

MATLAB provides a set of built-in DSP algorithm implementations, which Dar et al.'s code utilized. Since we used C++, which does not include built-in DSP libraries, we had to find a 3rd-party solution. We had to make sure that the chosen library provides the same algorithms used in the MATLAB code. In addition, we would like the library's algorithms to be implemented as efficiently as possible. As any veteran C++ developer will agree, the go-to C++ library collection is Boost [2]. Boost includes the Generic Image Library (GIL) module [3], which provides an efficient image processing framework. After additional research, the Open Source Computer Vision Library (OpenCV) [4] has proven to be a better fit for our needs, due to its higher prevalence, better documentation, and superior abstractions. Finally, our application must be able to interact with an image both as an image object and a matrix, interchangeably; A task easily achieved using OpenCV's APIs.

Optimization Procedure

To perform the described ℓ_2-constrained deconvolution optimization, we need some numerical optimization procedure. In their MATLAB implementation, Dar et al. used the Bi-Conjugate Gradient Descent (BiCG) method, whose implementation is provided as a built-in MATLAB function. To address this numerical optimization need in our work, we've considered either a C++ BiCG implementation or some other numerical optimization algorithm. The benefit of using a C++ implementation is the ability to directly compare our implementation to MATLAB's implementation. Unfortunately, OpenCV does not provide a built-in BiCG implementation. Hence, we tried utilizing other numerical optimization methods that OpenCV does provide. After looking into the optimization algorithms provided by OpenCV, we came to the conclusion that they don't fit our needs. Once we came to this realization, we embarked on a journey towards a C++ BiCG implementation that utilizes OpenCV's image matrix representation. We finally concluded that implementing our own version of the BiCG algorithm is necessary, as described in more detail below.

4 Implementation Challenges

Efficient BiCG implementation. As described above, we first tried using OpenCV's optimization algorithms. We then came across the C++ Iterative Methods Library (IML++) [5], which provided a highly efficient C++ BiCG implementation. Hence, we've decided to use this implementation. Implementing the BiCG algorithm involved prior theoretical mathematical knowledge in numerical optimization. After initial mathematical study, we began to implement. The main problems with IML++'s implementation were its use of C++ templates and its assumptions on the templated matrix realizations. OpenCV's matrices do not behave as expected by IML++. As a result, we applied modifications that allowed support for OpenCV matrices. After removing the template and modifying the operations to OpenCV's matrix operations, we found the matrix operator's size was too large to fit in memory. To solve this problem, we used an approach presented in Dar et al.'s MATLAB implementation. Its main idea is: given a circulant matrix, use a function to represent its kernel. When applying matrix operations to the function, the function should behave as the matrix represented by this kernel would. This approach is easily implemented in MATLAB, but in C++, it requires delicate usage of lambda functions and operator overloading. This optimization has significantly improved performance since high-demanding operations on large matrices were avoided.

Image representation differences. When comparing our algorithm implementation with Dar et al.'s, we've noticed major differences in runtime. Our implementation, even though written in C++, was significantly slower. Upon further inspection, we've discovered that our implementation did not meet Dar et al.'s algorithm stopping criteria, which is the tolerance of difference between v and z. While investigating the cause, we noticed significant pixel value differences

between the same images in both implementations. After confirming that our implementation was indeed logically correct, we decided to investigate OpenCV as the cause for the differences. Examining raw pixel values during runtime has led us to realize that MATLAB's runtime image representation is different from OpenCV's representation. We tried solving the problem in multiple ways: MATLAB Coder [6], which we tried to use to convert MATLAB's image reading and writing functions to C++ code, or MATLAB Compiler [7], to compile these functions directly into C++ executables, that can be run from within our C++ code. Unfortunately, the built-in MATLAB functions we needed were not supported by both the MATLAB Coder nor MATLAB Compiler. A second solution we've tried was to use the MATLAB Engine API for C++ [8], which creates a MATLAB runtime daemon and enables communication via C++ code. Due to library dependency mismatch, we could not use this option either. Our final attempt was to export the raw runtime pixel values to text files from both implementations and compare the values manually, in order to find some correlation or pattern to help us programatically solve the problem. The attempt was unsuccessful as no conclusive mapping between the values was found. Needless to say the different runtime pixel values yielded different numerical calculation results during runtime, which in turn did not meet the stopping criteria that were set.

5 Results

5.1 Runtime comparison

As expected, our implementation performed better, time-wise, than Dar et al.'s MATLAB one. A runtime comparison in milliseconds is presented in figure 3. We measured the runtime in several resolutions in order to try and pinpoint the parts of the code with the greatest influence on performance.

Time improvements can be measured both relatively (e.g. using percentiles) and absolutely. While a relative comparison may first seem more important, the de facto, perceived improvement is the absolute difference in milliseconds. Hence, what follows is an in-depth comparison of both relative and absolute differences, in percentage and milliseconds, respectively. This comparison ultimately enabled us to better analyze the runtime improvements we've achieved. The time measurements, taken as the averages over 10 different runs, are presented in table 1, followed by our analysis.

	BiCG	**Compress-Decompress**	**Single iteration**
Dar et al.	2211	561	3231
Ours	1665	26	1704
Absolute improvement	546	535	1527
Relative improvement	24.7%	95.4%	47.3%

Table 1: 10-run runtime averages comparison [milliseconds]

As can be seen in table 1, the Compress-Decompress phase's improvement of 95.4% in milliseconds is quantitatively about the same as the BiCG's 24.7% improvement, both of which constitute the total iteration time improvement of 47.3%. This has led us to conclude that both of these improvements are equally important. Therefore, even though BiCG's improvement provided a mere 24.7% boost in speed, the great amount of work we've put into researching and implementing the C++ algorithm was worthwhile.

Originally, the algorithm's total runtime was $14,435$ ms. Our implementation took $25,672$ ms, which is $11,237$ ms slower. Relatively, this results in a 77.8% slowdown. This may seem like an invalidation of our work. Actually, this is due to the fact that our algorithm did not meet the stopping criteria which the original implementation did meet after 4 iterations. Ours was limited to a maximum of 15 iterations, which it reached. Upon further investigation, following the first 4 iteration of our implementation, the PSNR value plateaued. As a result, we could safely modify the stopping criteria to better fit our image runtime representation, which yielded a correct 4-iteration run, whose total time was $7,156$ ms, a 50.4% improvement in total to the original implementation.

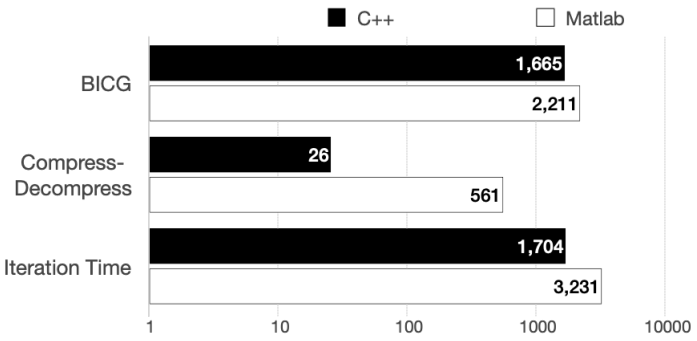

Figure 3: 10-run runtime averages visualization [milliseconds] (shown on a logarithmic scale)

5.2 Visual and quantitative results

Our implementation has reached lower PSNR values than the ones achieved in the original MATLAB implementation in some test images, and surpassed the original results in others. A numerical PSNR value comparison of six test images is presented in table 2. A visual representation of these test images with the corresponding PSNR values is presented in figure 4.

		Barbara	**Butterfly**	**Almonds**	**Cards**	**Cameraman**	**Table**
Dar et al.	**Input**	25.22	30.02	26.65	23.18	25.87	27.30
	Output	33.64	36.33	32.96	29.10	31.36	32.07
Ours	**Input**	25.32	30.09	26.94	23.47	26.00	27.65
	Output	34.21	37.75	33.12	30.51	34.48	32.64

Table 2: Input and output PSNR value comparison [dB]

(a) Input, Dar et al., 25.22 dB

(b) Output, Dar et al., 33.64 dB

(c) Input, ours, 25.32 dB

(d) Output, ours, 34.21 dB

(e) Input, Dar et al., 30.02 dB

(f) Output, Dar et al., 36.33 dB

(g) Input, ours, 30.09 dB

(h) Output, ours, 37.75 dB

(i) Input, Dar et al., 26.65 dB

(j) Output, Dar et al., 32.96 dB

(k) Input, ours, 26.94 dB

(l) Output, ours, 33.12 dB

(m) Input, Dar et al., 23.18 dB

(n) Output, Dar et al., 29.10 dB

(o) Input, ours, 23.47 dB

(p) Output, ours, 30.51 dB

(q) Input, Dar et al., 25.87 dB

(r) Output, Dar et al., 31.36 dB

(s) Input, ours, 26.00 dB

(t) Output, ours, 34.48 dB

(u) Input, Dar et al., 27.30 dB

(v) Output, Dar et al., 32.07 dB

(w) Input, ours, 27.65 dB

(x) Output, ours, 32.64 dB

Figure 4: Visual and PSNR comparison of Dar et al.'s and our results over several test image

6 Conclusion and Future Work

Considering the obstacles we've faced during our work and the results reached, we believe a substantial improvement has been made to Dar et al.'s original MATLAB implementation. As described above, the runtime improvement we've achieved is significant and the results closely match and sometimes even surpass Dar et al.'s results, while utilizing a modular, object oriented design. Having said that, there are still a few issues that our work has yet to resolve. The most major one is the image runtime representation difference described earlier, which makes comparing our quantitative results to Dar et al.'s ones a bit inconclusive, even though a visual qualitative restoration can be clearly seen. Seeing as image pixel values differ in both implementations, applying mathematical operations to these values will result in different output values and thus different restored output images. Once this problem is addressed and an identical image representation is used in both implementations, a more precise quantitative comparison can be performed. Alternatively, the current image representation may be used, as long as the mathematical operations have been tweaked to adhere to the new representation. Either way, once a solution is found, integrating it into our existing code should be a minor task, thanks to the modular design used in our implementation.

References

[1] Y. Dar, M. Elad, and A. M. Bruckstein, "Restoration by Compression," IEEE Transactions on Signal Processing, vol. 66, no. 22, pp. 5833–5847, 2018.

[2] Boost C++ Libraries. boost.org

[3] Boost Generic Image Library (Boost.GIL). boost.org/doc/libs/1_69_0/libs/gil/doc/html/index.html

[4] Open Source Computer Vision Library (OpenCV). opencv.org

[5] Iterative Methods Library (IML++). math.nist.gov/iml++

[6] MathWorks MATLAB Coder. mathworks.com/products/matlab-coder.html

[7] MathWorks MATLAB Compiler. mathworks.com/products/compiler.html

[8] MATLAB Engine API for C++. mathworks.com/help/matlab/calling-matlab-engine-from-cpp-programs.html