Modern Image Denoising Techniques
Bryan Poling - Spring 2013

Background - What is Denoising?

There is a signal of interest that we wish to measure

In general, a signal is represented as a function of one or more variables. The domain and range of the function can be either discrete or continuous spaces.

Examples:

- In electronics, a signal may be a real-valued function of time (representing a voltage, for instance)
- An audio signal is a real-valued function of time, representing the time-varying pressure in the air at the microphone.
- An image is a function of 2 spatial coordinates. The range of the function can live in $\mathbb{R}^1$ (grayscale image), $\mathbb{R}^3$ (RGB image), or higher.

Background - What is Denoising?

The signal is corrupted by noise

We cannot directly observe the signal of interest. We measure the signal through some form of instrumentation. Our imperfect instruments corrupt the signal.

This is usually modeled as follows:

$$y = x + n \tag{1}$$

x represents the true signal. n represents noise on our signal. y is the observed signal.

The goal of denoising is to recover the true signal (x) from our observed signal (y).

Background - What is Denoising?

In order for denoising to be possible, we must make assumptions on the true signal (and/or the noise).

We need assumptions because without them...

How do we know this isn't what the true signal is *supposed* to look like?

What types of assumptions are used for denoising?

Assumptions on the true signal:

- Smoothness (equivalent to frequency roll-off)
- Sparsity in the appropriate domain
- Local self-similarity

Assumptions on the corrupting noise:

- Noise at different times/locations are independent
- Noise is 0-mean
- Noise is *stationary*
  (Noise properties do not change wildly with time/location).
- Explicit noise model (Gaussian, Uniform, etc.)

A Classical Example

Consider a voice audio signal

This is a real-valued function of time. Human voice only occupies a small piece of the band of audible frequencies. We can hear up to approximately 20 KHz, but voice is typically between 300 Hz and 3.5 KHz.

It is reasonable to make the following assumption for denoising:

The true signal does not have frequency components below 300 Hz or above 3.5 KHz.

A Classical Example

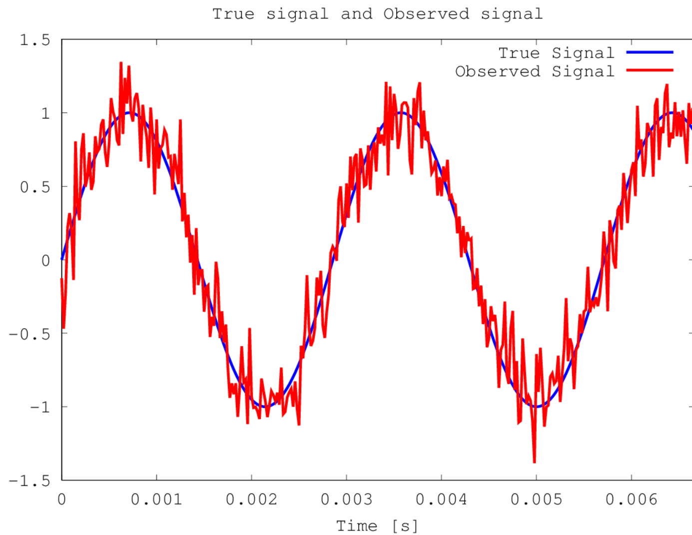
Exploit our assumption

Given our assumption, we know that any frequency components of our observed signal that lie outside of 300 Hz - 3.5 KHz must be due to noise. Also, anything inside that band could potentially be part of our signal.

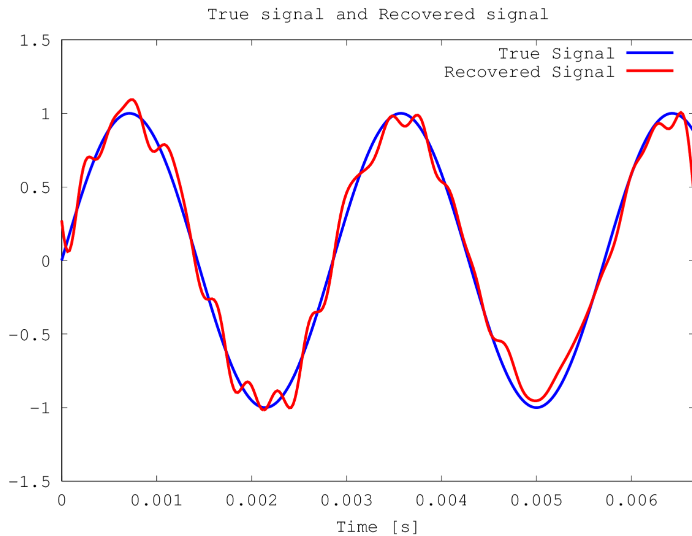
The Solution

We convert our observed signal to the frequency domain and remove all components that lie outside of 300 Hz - 3.5 KHz. Convert the result back to the time domain. This is unlikely to equal the true signal, but it is the best we can do. If there is noise that falls in the 300 Hz - 3.5 KHz band, it is indistinguishable from the true signal without additional assumptions.

A Classical Example



A Classical Example



What about images?

Image smoothness/Frequency roll-off assumption

- Natural images tend to exhibit a pretty high degree of smoothness. Thus, most of the energy in an image signal ends up at lower frequencies. But...
- Images also have sharp edges and lines (not due to noise).
- If we filter out high frequencies we will kill off more noise than signal. Unfortunately, we are very intolerant to any loss of signal.
- If we filter out high frequencies, sharp edges get blurred and the results look very bad.

Use other assumptions:

Sparsity in the appropriate domain

This is based on the observation that if you represent an image in certain ways, the representation can have very few non-zero entries.

Local self-similarity ($$$ Cha-Ching $$$)

A small sub-image extracted from a larger image at a certain location is called a "patch". If you grab a patch from a natural image and compare it to patches from nearby, you will usually find several other patches that are *very* similar to the one you started with.

How do we exploit sparsity?

A method which exploits sparsity has this flavor:

- Come up with a transformation such that in the transformed domain, you expect the image to be sparse. This may or may not be done "on-the-fly".
- Transform a given image into this new domain.
- Zero out small non-zero values in your new representation of the image (remember... most entries are supposed to be 0).
- Undo your transformation to re-build your denoised image.

Examples

- Wavelet denoising
- Texture dictionary methods

How about local self-similarity?

A method which exploits local self-similarity has this flavor:

- For each pixel, build an image patch centered at that pixel.
- For each pixel compare its corresponding patch with nearby patches. Find some patches that are similar. This process is called "Block Matching".
- Modify this set of patches to make them "more similar".
- Re-assemble the image from all of your modified patches.

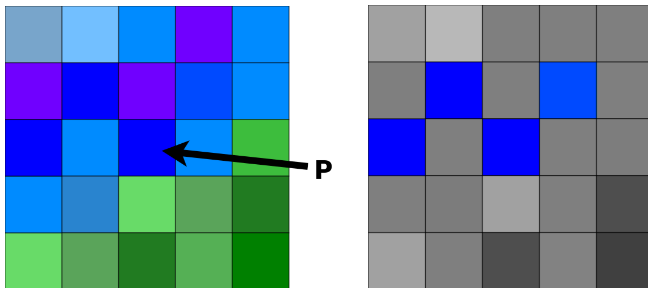
Examples

- The Method of Non-Local Means (NLM)
- Block-Matching 3D (BM3D)

Non-Local Means

The Idea:

Consider a clean image (no noise). For any given pixel, p, there are likely to be other pixels nearby that are nearly identical in color/intensity. Imagine that an oracle will tell us where to find these similar pixels. If we replace the value of p with the average value of this set of pixels, we do not damage the signal (this process effectively does nothing).



Non-Local Means

The Idea:

Now imagine that each pixel has some random noise on it.

- Assume that the noise values at different pixel locations are independent.
- Assume also that the noise is 0-mean.
- Imagine that for a given pixel, p, the oracle can still tell us which pixels in the true signal are most similar to p.

If we replace the value at p with the mean of this set of pixels, we do not affect the true signal, but the new noise value will be the mean of a set of 0-mean, i.i.d random noise variables. The mean is very likely to be closer to 0 than before. Thus, this process reduces noise without damaging the signal (law of large numbers).

Non-Local Means

The Idea:

Here is what we want to do. Iterate over all pixels in an image. For each pixel, p:

- Ask the oracle for the set of pixels in the image that are supposed to be most similar to p (in the true signal).
- Compute the average value of all pixels in this set.
- Replace the vale of pixel p with this average.

Non-Local Means

The problem: getPixelsFromOracle(int x, int y)

We don't have access to the true signal. So how are we supposed to know which pixels to average? We must try to identify the correct pixels by only looking at the noisy signal.

Block Similarity

We use another observation. Namely, at a given location in an image image, we won't just find similar pixels nearby, but we will find similar patches nearby.

Non-Local Means

Why is this more useful?

The similarity between two pixels in a signal can be made unobservable by adding noise. The similarity between two patches is more resilient because the noise values at each pixel are independent.

The per-pixel noise causes comparable corruption when comparing two pixels as when comparing patches, but with patches the variance of the corruption is much lower. This makes the selection of good patches more stable.

Insert Math

Non-Local Means

How do we use this to help select pixels for averaging?

For a given pixel p, we build a patch centered at p. For each nearby pixel (in some neighborhood) we build a patch centered there. We then compute the "distance" between the patch at p with all of these other patches. NLM uses mean-square error for this distance. The distance between that patch centered at p and the patch centered at any other pixel, p', is used as a measure of did-similarity between p and p'. We could just select the K least dis-similar nearby pixels for averaging.

Non-Local Means

Use all pixels and weighted averaging

In Non-Local Means, we don't just select the K best pixels for averaging. Instead, we use all nearby pixels and we use a weighted average based on each pixels dis-similarity with p. The weight for pixel j is computed as follows:

$$w(p,j) = \frac{1}{Z(p)} e^{-\frac{\|\text{Patch } P - \text{Patch } j\|_2^2}{h^2}} \tag{2}$$

h is a parameter that must be matched to the noise level of the image. $Z(p)$ is just to normalize the weights so they all sum to 1.

Non-Local Means

Qualitative Analysis - The Good

- Using all nearby pixels and applying weighted averages is supposed to make us more robust against mistakes in identifying the best pixels to average with.
- It also has the effect that if the patch at pixel p does not look like any other nearby patches, the total amount of averaging done at pixel p is very small. The intuition here is that we want to preserve unique areas without blurring them. We can call this adaptivity.

Qualitative Analysis - The Bad

- Using all nearby pixels and applying weighted averages has the downside that we are sure to include energy from nearby pixels that are not similar to the pixel in question (although the weight is small).
- The adaptivity of the method does not tend to cause problem on images with synthetic noise. However, real noise often has different characteristics in different regions of an image. The result of this adaptivity is that the denoising can look "patchy" on some images.

Non-Local Means

Non-Local Means

How can this be improved?

Non-Local Means is a very solid idea, but the implementation can be improved.

- Don't use all nearby pixels. Just select K good nearby pixels to average with. This works well in practice.
- Either don't use weighted averaging, or choose weights in such a way that the amount of energy that comes from neighboring pixels is fixed. This provides spatial consistency to the denoising (we don't want "patchy" results)
- Use Color! NLM is presented as a single-channel algorithm (perhaps just to simplify the formulation). If you run this on each color channel in an image separately, you are wasting wonderful color information.

Non-Local Means

Improvements:

- When processing pixel p, we will define the weight for pixel j to be $1/rank(j)$. $rank(j)$ is the number of pixels more similar to p than j is. We cap the number of pixels at K. All pixels with rank K or higher get weight 0.
- Process all color channels together. Use distance between color patches to build dis-similarities. Then use the same weight for all color channels.
- Iterate. If this algorithm actually reduces noise without significantly damaging the signal, then after running it once, if we try again, we will be better able to identify the correct pixels to average with. This suggests a scheme where we run several times, only mildly denoising the image in each iteration.

Noisy Image

Recovered Image - Non-Local Means

Recovered Image - Modified Non-Local Means

BM3D (Block Matching 3D)

The Idea:

Block Matching 3D is similar to Non-Local Means. It exploits block matching to collect sets of similar image patches. It then uses ideas from sparsity-based denoising to denoise patches. The image is re-assembled from denoised patches in a complicated way.

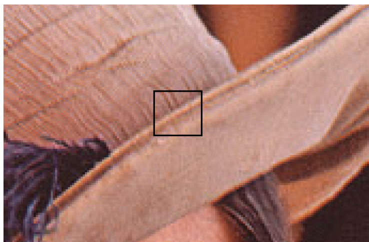
Modern Technique

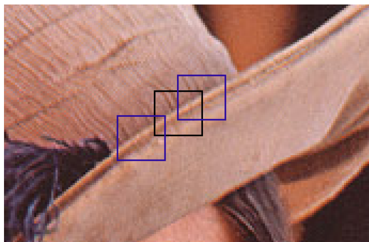
BM3D is currently considered state-of-the-art. Other methods which compete with it also employ block matching to exploit local self-similarity.

BM3D (Block Matching 3D)

Algorithm Step 1 - Perform for each pixel p in the image

build a patch centered at p. Find a collection of nearby patches that are most similar to this patch.

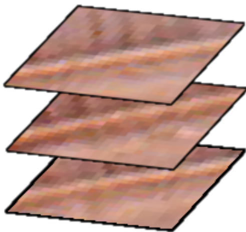




BM3D (Block Matching 3D)

Algorithm Step 1 - Perform for each pixel p in the image

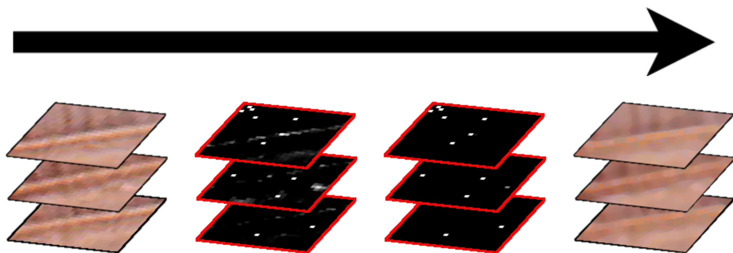
Stack these patches into a 3D array. Each slice of this array will be a single patch. Remember the center pixel location of each slice.



BM3D (Block Matching 3D)

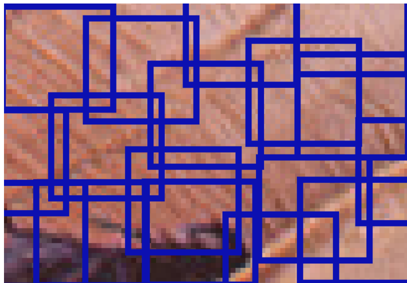
Algorithm Step 2 - Perform for each 3D array of patches, S

Let T be a 3D transform such that image stacks have sparse representation in the transformed domain. An example could be a wavelet transform. Compute $T(S)$. Threshold the coefficients of the result (set all small values to 0). Then compute the inverse transform.



Algorithm Step 3

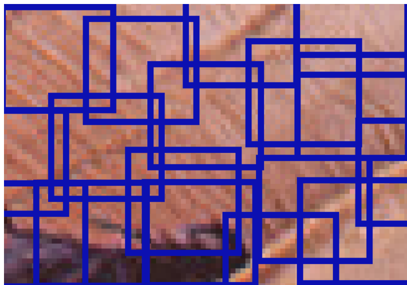
We now have a large collection of denoised patches. Each pixel is in at least one patch. However, the patches will overlap heavily. For each pixel p, we must fuse all of the value estimates for p provided by the patches that contain p.



BM3D (Block Matching 3D)

Algorithm Step 3

We could just average all estimated values for p. BM3D instead uses a weighted averaging scheme to combine these estimates.



BM3D (Block Matching 3D)

Details

We just went over the idea behind BM3D. It is actually a bit more complicated. BM3D actually executes 2 passes on the image (recall that this was a suggested improvement for NLM). In the first pass, the weights used in the final re-combination step are a function of patch distances. The distance metric used is complicated. Patches are pre-blurred before distances are computed. In the second pass, weights are generated by an empirical Wiener filter.

Noisy Image

Recovered Image - Modified Non-Local Means

Recovered Image - BM3D

BM3D - Qualitative Analysis

Why the added complexity?

BM3D tries to exploit both local self-similarity and the sparsity of natural images in an appropriate domain. The transform and threshold step is lifted out of classical denoising techniques.

How does it compare?

BM3D provides slightly better results than the modified NLM method discussed earlier (on average). Comparison reveals that BM3D recoveries are no sharper than NLM recoveries. The improvement comes from added spatial consistency. This is probably due to the way filtered blocks are re-combined to form the final image.

Conclusions

- Use block matching to exploit self-similarity!
- There are several competing methods for filtering blocks to make the slices more similar. Many of these methods offer similar results.
- The way in which filtered blocks are re-combined to form the final image is important. Doing this carefully can give better-looking results.

Thank You!

Optional - Computational Aspects

What is the best way to implement one of the algorithms mentioned here?

We state algorithms as sequential programs. However, most of the recent advances in processors have related to parallelism. A modern CPU has several processing cores. Denoising algorithms such as these are "local" algorithms. That is, you don't need to process the pixels in the image in any special order. This is great for parallelism (such a problem is often called "embarrassingly parallel").

Thread Parallelism

Threading APIs like PThreads, OpenMP, C++ 11 Threading, MPI allow you to write code that exploits multiple processing cores at the same time.

Optional - Computational Aspects

SIMD Parallelism

In addition to having multiple execution cores on modern CPUs, each separate core is effectively a "vector computer", capable of processing many pieces of data simultaneously through the use of SIMD instructions.

SIMD Parallelism

The datapath in a vector computer is very wide, while the control path is still very narrow. You can pack many pieces of data into the datapath and operate on all of them at the same time, so long as you are performing the same instructions on each item.

SIMD on todays hardware

Modern x86 processors from Intel and AMD implement "Advanced Vector Extensions". A single core in one of these processors has a 128-bit datapath. You can operate on 4 single-precision floating point values at once, or 2 double-precision floating point values at once. You can also pack integers into the datapath of width 8, 16, 32, or 64 bits. This means you can perform 16 operations at once if you are working with 8-bit integer data types (on one core)!

Optional - Computational Aspects

SIMD on todays hardware

The next iteration of the SIMD extensions is doubling the datapath (these are the Haswell extensions). You will be able to operate on 32 single-byte values at once, per execution core.

SIMD and multi-threading together

Each CPU core supports SIMD instructions separately. This means that the speedups from threading and SIMD compound. If you have a 4-core CPU with the Haswell SIMD extensions (I don't think these are actually on the market yet). you can process $32 * 4 = 128$ 8-bit int objects at once.

Optional - Computational Aspects

But Wait... Its Even Better

In addition to standard SIMD, new processors have support for so-called "reduction operators". For instance, There is a single instruction in AVX that takes 2 vectors of 16 uints. It computes their element-wise absolute differences ($|a - b|$) and then adds up all of the results. This is 1 instruction!

Optional - Computational Aspects

The Good News

Images are generally represented using 8-bit unsigned characters for each color value at each pixel. This means we can get the maximum benefit from all of these recent advances in CPU technology.

Optional - Computational Aspects

The Bad News

It is harder than ever to actually exploit the new capabilities of these advanced CPUs. Matlab will convert everything to double-precision float. It won't use SIMD at all, and it will make limited use of multi-threading. Result: We get very little benefit from our cutting edge processors.

In Compiled Code

If we write code in C/C++ ourselves to do image processing we can exploit all of these new capabilities, but don't expect your compiler to do it for you.

Optional - Computational Aspects

In Compiled Code

People have been working on compiler auto-threading for decades. It is now at a point where you can get the maximum benefit of multi-threading (in embarrassingly parallel problems) with almost no effort. Use OpenMP to get a fast, threaded implementation of your algorithm.

In Compiled Code

Compiler support for automatically exploiting SIMD is very immature (don't count on it). You have to do this by hand. Most popular compilers expose SIMD instructions to the programmer through primitive functions (basically in-lined assembly, but a bit more portable).

Optional - Computational Aspects

Is it worth it?

- Go from Matlab/Octave to C++ with aggressive compiler optimizations: 10x-100x speedup.
- Eliminate all of the casting overhead by working with native int data types: 2x-10x speedup.
- Use OpenMP to thread your code: 2x-4x speedup (= # of cores)
- Use SIMD in each thread: $\approx$ 10x-20x speedup (depends on tricks)

Of course, high-level algorithmic optimizations are extremely important as well.

Example

In my modified NLM code, going from Octave to threaded, optimized, C++ with SIMD, my code went from a 5 hour run-time on the "Bears" example to 1.5 seconds.