# amDCT

## adaptive multipurpose DCT Filter

## Version 1.0 January 2017

## by Jim Conklin

## Table of Contents



# Overview

**amDCT** provides **sharpening, expansion, smoothing**, and **bright noise removal** in a single filter that can be used on video of any level of quality. These four operations work synergistically to drastically reduce block artifacts while maintaining detail and increasing local contrast.

A key advantage of **amDCT**--and one way in which it is a major advance over previous approaches--is that it determines how much smoothing, local range expansion and sharpening should be done at three different levels: the frame as a whole, block-by-block, and pixel-by-pixel.

Adaptivity was built into the core algorithm in order to provide a filter that could automatically adjust its strength to varying conditions in badly-encoded material so that good frames or good parts of bad frames would not be oversmoothed and bad frames or bad parts of good frames would be smoothed and deblocked more.

This adaptivity is especially useful if the video has some frames that need much more smoothing than others because of deinterlacing errors or transmission bandwidth limitation errors which can cause the video to occasionally become blocky.

Another useful feature is that users can select arguments specifically geared to the following:

- the ability to do fast adaptive edge-of- block deblocking of a frame.
- the ability to remove just the bright noise in a frame.
- the ability to create a very smoothed frame from the original. MVTools can use the smoothed frame as a prefiltered clip for more reliable motion estimation.

# How amDCT Works

The core of **amDCT** takes as its starting point the idea for reducing block artifacts proposed by A. Nosratinia of doing a quant dequant operation multiple times with the frame shifted to a different position each time then taking the average value as the result.

The trick in deblocking is of course to remove only the transients that are introduced by the encoding process, and not a line in the original image that happens to have the misfortune of being on a block boundary. **amDCT** reduces the high-frequency transients at the block boundaries by doing the following:

> For each 8x8 block in an image **amDCT** samples various 8x8 blocks shifted but still overlapping the current block. This is done by shifting the whole image. Then for each shifted image every 8x8 block is run through a fdct-quantize-dequantize-idct process that removes high-frequency transients. Then the shifted image is shifted back to its original position and averaged with all of the other shifted, processed, shifted-back, images to produce the deblocked image. This process reduces the high-frequency transients at the block boundaries to stay within the maximum high frequency of the original image. Additional information on this technique is provided in the documentation directory.

**amDCT** uses the same basic approach for smoothing, local range expansion and sharpening.

# Getting Started

In order to retain the maximum amount of detail in the video, **amDCT** should be the first video-processing filter applied to the video clip (assuming that the video clip is already in YV12 format). Since the values in the **amDCT** argument string will need to be set to optimally process both the best and the worst frames in the video, you should begin by previewing the video in VirtualDub and look for a set of outlier frames that need minimal processing; those that need a lot of deblocking; those that need a lot of smoothing; those that need a lot of sharpening; and those that have a lot of very bright area noise. Write down the frame numbers for the outlier frames so you can quickly recheck them as you set each parameter. Then proceed through the following five steps:

1. **quant**=1 Set quant to the lowest value where frames that need minimal processing look good.
2. **adapt**=1 Set adapt to the lowest value where frames that need a lot of deblocking or a lot of smoothing look good. Check the outlier frames for step 1. If they have become oversmoothed go back to step 1 and reduce the **quant** value. When you come back to step 2 the **adapt** value may need to change. If it does, make the change and go back to step 1. If it doesn't, then continue to step 3.
3. **expand**=0 Set expand to the lowest value that provides the desired range expansion. **expand** interacts with sharpening making sharpening artifacts worse, so initially keep it low.
4. **sharpWAmt**=5 Set **sharpWAmt** to a value that brings out detail without adding too many artifacts. .
5. **brightStart** =220, **brightAmt**=1) Set **brightStart** to the largest value that will tame the bright noise. If **brightStart** is too low, the midrange luma values will be oversmoothed. **brightAmt** controls the strength of the extra smoothing.

Then go back to step 1 and recheck the outlier frames and iterate the above steps until you have found a reasonable compromise among sharpness, artifacts, noise, detail, and smoothness. Note that artifacts can be reduced by decreasing **sharpWAmt**, reducing **expand**, increasing **quality**, or increasing **adapt**. Increasing **quant** will also decrease sharpening-induced artifacts but should only be done if the artifacts show up on frames that need minimal processing.

# Parameters (default values in bold):

**amDCT**(clip, adapt=**0**, shift=**3**, quality=**3**, quant=**1**, matrix=**1**, expand=**0**, sharpWpos=**5**, sharpTpos=**7**, sharpWAmt=**0**, sharpTAmt=**0**, brightStart=**255**, brightAmt=**0**, qtype=**1**, ncpu=**1**)

## General Settings

**adapt** (**0**...31): This parameter specifies the maximum adaptation value used. The system is adaptive at the frame level, the block level, and the pixel level.

Adaptation works by determining how blocky the frame as a whole is. It does so by using the blindPP code to do a fast deblock on the frame and then uses the difference between the deblocked and the original frame to compute the Total Blockiness of the Frame (TBF). The possible high and low value of TBF has been determined experimentally. Other measures used are several statistical measures done for each block of the frame. Please see the top of the source file blindPPcode.c for credits for the blindPP code.

Next for each block in the frame the system computes separate blockiness values for smoothing, range expansion, and sharpening, using a different algorithm for each.

The system then separately computes 31 levels of sharpening, 31 levels of range expansion, and 31 levels of smoothing for each block.

This information is then used in the main DCT loop as an index to retrieve the appropriate strength of processing for that block.

Blocks with high blockiness reduce the strength of range expansion and sharpening for the block. Blocks with high blockiness increase the strength of smoothing for the block.

If **adapt** is less than **quant** or if **quant**=0, then adaptivity is turned off for smoothing.

The following example takes advantage of the special case of **quant=0** to do fast adaptive edge-of-block deblocking of a frame.

**amDCT**(**quant=**0, **adapt=**20) Large values of **adap**t often work well.

**shift** (0, 1, 2, **3**, 4, 5): Specifies the number of shifted versions used for filtering. The value chosen determines the speed vs. quality tradeoff. A larger value will tend to produce an image that appears smoother and more detailed and has fewer artifacts.

- **0.** Uses 0 shifted versions. Runs through the loop once without shifting. Used during development for debugging.
- **1.** Uses 4 shifted versions.
- **2.** Uses 8 shifted versions.
- **3.** Uses 16 shifted versions. Best speed vs. quality
- **4.** Uses 32 shifted versions.
- **5.** Uses 64 shifted versions.

The processing of each shifted version of the frame is where the filter spends most of its time. For each shifted version of the frame the frame is divided into 8x8 blocks. Each block is converted from the spatial domain to the frequency domain via DCT. The DCT block then has the values modified by doing a quant operation followed by a dequant operation. The resulting values are then converted back to the spatial domain by doing an inverse DCT. This loop is highly optimized. Even so, a 720x1280 pixel image contains 14,400 8x8 blocks. 16 shifted versions means 230,400 times thru the loop. 64 shifted versions means 921,600 times thru the loop. With **quality**=3 when doing smoothing, sharpening, and range expansion the total times thru the loop for 16 shifted versions grows to 633,600 and for 64 shifted versions 2,016,000.

**quality** (1, 2, **3**, 4): Provides a speed vs. quality tradeoff when doing sharpening, range expansion, or bright noise removal. A larger value is more accurate and time consuming.

**1.** Does no adaptivity.

- If **adapt** > 0 the blindPP code is run with strength equal to the **adapt** value.
- Does no presmoothing to reduce sharpening artifacts. Does sharpening, expansion, and smoothing in one pass.

**2.** Does fast lower-quality adaptivity.

- If **adapt** > 0 the blindPP code that works on horizontal lines `deblock_horiz()` is run and the amount of change is computed at both the frame and the block level. This information is used to set the internal variable frame_quant to the appropriate level between **quant** and **adapt**. The blindPP code is run with strength equal to frame_quant. If **quant** > 0 then **quant** is set to frame_quant.
- Does presmoothing biased toward speed to reduce sharpening artifacts.
- Does sharpening in 1 pass and combined expansion and smoothing in a second pass.

**3.** Does slower higher-quality adaptivity.

- Does presmoothing, biased toward quality, to reduce sharpening artifacts.
- Does adaptive processing at the block level.

**4.** Sharpening, Expansion, and smoothing are each done in their own pass.

- Does halo reduction.

**ncpu** (**1**…4): The number of cpus to use.

## Smoothing

**quant** (0...**1**...31): This parameter specifies the minimum value used for smoothing. Zero turns off smoothing. **quant** controls the overall strength of the high-frequency transients removal.

**matrix** (**1**-8): Selects a smoothing matrix to use. This is useful when a large **adapt** range is desired for video that has some frames that need a lot of smoothing and others that need almost no smoothing. Using a low-strength matrix allows **adapt** to be set at a higher level to adequately smooth the noisy frames while not oversmoothing the good frames. A stronger matrix is useful if the video has noisy frames that **adapt** does not compute as needing as much smoothing as they actually fffdo.

**matrix** provides some control over the thickness and direction (horizontal, vertical or diagonal lines) of the transient.

Note **matrix** is not used by **qtypes** 2 and 4 which use the built-in h263 matrices.

| Predefined Matrix Numbers and Relative Strengths | | | |
|---|---|---|---|
| Matrix Number | Strength | Additional Horizontal Line Strength | Additional Vertical Line Strength |
| 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | **3** |
| 3 | 1 | **3** | 0 |
| 4 | 1 | **3** | **3** |
| 5 | 6 | 0 | 0 |
| 6 | 6 | 0 | **6** |
| 7 | 6 | **6** | 0 |
| 8 | 6 | **6** | **6** |

For an excellent description of what a matrix does see http://forum.doom9.org/showthread.php?t=54147 and page 13 of Basis Images of 8x8 DCT.

**qtype** (**1**, 2, 3, 4): Selects the type of quant-dequant processing that is used in the filtering. They are ordered by their deblocking strength from least strong to strongest. h263 intra and h263 inter, **qtypes** 2 and 4, ignore the matrix argument and use their own built-in matrices.

1. mpeg intra
2. h263 intra
3. mpeg inter
4. h263 inter

The intra in **qtypes** 1 and 2 refers to the encoding done for I-Frames. The inter in **qtypes** 3 and 4 refers to the encoding done for P-frames and B-Frames.

For a good introduction to mpeg coding see:
http://www.cse.iitd.ac.in/~pkalra/siv864/pdf/MPEG_Video_Compression.pdf

Different matrices are usually used when encoding I-Frames then when encoding P-Frames and B-Frames. Since all the information about how a particular frame was originally encoded has been lost by the time **amDCT** is passed a frame to process, **amDCT** cannot do automatic intra-inter **qtype** or matrix switching.

To have **amDCT** produce a smoothed prefiltered clip for MVTools2():

**amDCT**(**qtype**=2, **quant**=2, **adapt**=6)

## Local Range Expansion

**expand** (**0**...31): The amount of local range expansion applied. Zero does no expansion. Range expansion will sometimes increase the sharpening effect. It can also make sharpening artifacts worse. Range expansion will not increase the brightness of a pixel beyond the value of **brightStart** .

## Sharpening Controls

Sharpening works by increasing small values in specific locations of the DCT-transformed 8x8 macroblock. The following four parameters specify the feature width and the strength of sharpening. In the parameter names **W** stands for Wide and **T** stands for Thin.

**sharpWPos** (**5**-7): The DCT block starting column. Five sharpens larger features; seven sharpens smaller features.

**sharpTPos** (5-**7**): The DCT block ending column. Five sharpens larger features; seven sharpens smaller features. If the parameter **sharpTPos** is omitted it defaults to 7.

**sharpWAmt** (**0**...31): The strength of the starting column.

**sharpTAmt** (0...31): The strength of the ending column. The value defaults to the value of **sharpWAmt**.

Sharpening will not increase the brightness of a pixel beyond the value of **brightStart** .

Sharpening was originally going to be called detail preservation. It was put in as an aid to preserving apparent detail when the frame was being deblocked and smoothed. It did this really well. During the procces of finetuning the internal parameters I discovered that if the strength of sharpening was turned up it would also sharpen fine low-contrast lines such as individual hairs. Unfortunately, when turned up that high, artifacts would start to become visible. After a lot of tuning the artifacts were minimized and the amount of useful sharpening increased.

Sharpening works by multiplying the DCT values that are associated with the **sharpWPos** and **sharpTPos** of the block being processed. It does so by an amount indirectly determined by the sharpening strength set by the values in **sharpWAmt** and **sharpTAmt**. As the sharpening strength becomes larger, DCT block values close to an experimentally-determined hard-coded upper limit at which the DCT value will not be changed. This can result in low-level noise being amplified more than the signal and artifacts becoming more prevalent. The low-level noise is automatically prefiltered in order to reduce artifacts while not removing the signal that is being sharpened. The strength of the prefiltering is dependent on the strength of the sharpening being applied. Increasing the value of **quality** will often allow increased values of sharpening before artifacts become apparent.

Both sharpening and range expansion will tend to increase haloes that are already in the image. If they are turned up high enough they will create haloes. These haloes are reduced by decreasing the amount of sharpening and range expansion at these boundaries.

Sharpening--and to a lesser extent range expansion--may create or increase preexisting frame-to-frame luma differences in the areas being sharpened. This will cause individual pixels to flicker. This is especially true with larger control values. This effect can usually be handled by running a temporal smoother such as mdegrain after running **amDCT.** Mdegrain will also reduce other sharpening artifacts.

## Bright Noise Removal

Bright noise removal applies extra smoothing to the brightest parts of a frame.

Videos with high-contrast high-luma scenes encoded with low bitrates will often produce quite visible bright artifacts that are difficult to remove. The arguments **brightStart** and **brightAmt** are used to modify the source-frame pixels which will undo some of the blocking artifact damage that was in the original frame, as well as mitigate some of the damage that would otherwise be introduced by any of the range expansion or sharpening that **amDCT** will be doing.

To have **amDCT** process only the Bright Noisy Parts of the Frame:

brightSmoothedClip **= amDCT(brightStart**=val**, brightAmt**=val**)**

**brightStart** (0...**255**): The smallest luma value at which extra smoothing will be done in bright areas of the frame. The default of 255 does no extra smoothing. Note that **brightStart** also sets the maximum value to which a pixel will be sharpened or expanded. Set **brightStart** to the largest value that will tame the bright noise. If **brightStart** is set too low, the midrange luma values will be oversmoothed.

**brightAmt** (**0**...31): The amount of extra smoothing that will be done on the bright areas of the frame. Zero does no extra smoothing. 31 does the most extra smoothing. The amount of extra smoothing applied will be **brightAmt** at luma 255 and will decrease until it reaches the luma value specified by **brightStart**.

## Dark Detail-Preserving Controls

These controls only have an effect if smoothing is being done and neither expanding nor sharpening is being done. Detail in dark areas of the frame can often be oversmoothed. **darkStart** and **darkAmt** can be used to maximize the amount of detail in the dark areas.

**darkStart (0**...255): The largest luma value to which extra detail preservation is applied to dark areas of the frame. The default of 0 does no extra detail preservation.

**darkAmt** (**0**...31): The amount of extra protection applied to the dark areas of the frame. The default of zero does no extra protection. 31 does the most extra protection. The amount of extra protection applied will be **darkAmt** at luma 0 and will decrease until it reaches the luma value specified by **darkStart**.

The program called Colors from [den4b.com](den4b.com) can help you determine the **brightStart** and **darkStart** values.

# Possible Future amDCT Enhancements

- Refine adaptivity to take into account the size of a frame. A 1080p frame can benefit from more smoothing without losing perceived detail than can a 352x240p frame.
- Implement a level between the frame and the individual blocks that will identify a contiguous area of blocks that should be processed in the same way.
- Write code to allow the core loop to run on a GPU so that a high number of shifted versions can be processed more quickly.
- The core loop would be an excellent place to convert an 8 bit to a higher bit-depth representation. For example, input an 8-bit per pixel frame and output a 10-bit or 12-bit per pixel frame. The core loop should be able to do this with very high quality using **quant**=1, **shift**=5, and a very weak **matrix**, since it already takes the sum of 64 samples and averages the sum down to 8 bits by doing a right shift by 6. A right shift by 4 would give 10 bits and a right shift by 2 would give 12 bits.
- Allow processing of U und V planes.
- Allow users to use their own smoothing matrix.

# History

The idea of reducing block artifacts by doing a quant dequant operation multiple times with the frame shifted to a different position each time then taking the average value as a result was put forth in A. Nosratinia's 2001 Master's Thesis at University of Dallas and was discussed in his paper: Nosratinia, Enhancement of JPEG-Compressed images by re-application of JPEG, Journal of VLSI Signal Processing, vol. 27, pp. 69-79, 2001. A copy of the paper is in the file "JPEG resampling paper.pdf" in the documentation directory. See also Homepage of Aria Nosratinia.

I first came across this idea in SmoothD, which is an AVS filter implementation of Nosratinia's JPEG resampling idea written by Tobias Bergmann**.** See the Doom9 SmoothD thread here.

The problem I had in using SmoothD was that it had a limited dynamic range, i.e., you could set the strength so it worked well for smoother areas but then it would not work well for noisy areas or else you could set the strength for noisy areas and it would oversmooth the smoother areas. I tried to build in some form of adaptability by using an AVS script, but it would crash if it was called more than once in a script.

Since it quickly became clear that a substantial rewrite of SmoothD would be needed to address that limitation, I went on to create SmoothD2, which is a complete rebuild of SmoothD. See the Doom9 SmoothD2 thread here. Smooth D2 is stable and provides the ability to adjust the strength of smoothing from the edge of the block to the center of the block.

Having released SmoothD2, I then went back to the drawing board to see whether I could create a program that would allow genuine adaptability. **amDCT** is the result.

# Code Used in amDCT

There is no way I could have written **amDCT** without borrowing both code and ideas from others.

To the best of my knowledge all of the code that is used in **amDCT** is open source. Code that I wrote is open source under GNU GENERAL PUBLIC LICENSE Version 2, June 1991. Additional code incorporated in **amDCT** is covered under some form of open source license such as GNU GENERAL PUBLIC LICENSE, MIT open source license, or Apache License:

**Asmlib:** A multi-platform library of highly-optimized functions for C and C++. Version 2.30. 2012-07-08 By Agner Fog. 2003-2012. GNU General Public License.

**blindPP:** Parts of the blindPP deblocking code were used to provide some of the information the adaptation calculator in **amDCT** uses. Authors identified in blindPP include “tritical", Donald A. Graft, MarcFD, Vlad59, and Tom Barry. The blindPP deblocking code came from PostProcess.cpp and PostProcess.h source files in the dgdecode158src file. GNU General Public License. Version 2, June 1991.

**Ctmf:** A Constant Time Median Filter. http://nomis80.org/ctmf.html Copyright (C) 2006 Simon Perreault. GNU General Public License

**Dgdecode158scr:** I used DrawYV12 routines to output debug text to the output frame. Also a source of the blindPP deblocking code and derring code. GNU General Public License Version 2, June 1991.

**Smooth, Max, Min, and Sad Filters:** are based on the JavaScript code found at Ivan Kuckir's blog that had the following on it. "All code that you see at this blog is free to use, under MIT license."

**UnDot:** A simple Dot Remover Copyright (C) 2002 Tom Barry - trbarry@trbarry.com This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License.

**Xvidcore1.3.2:** Used some of the matrix code and the quant dequant code and most of the fdct idct code and block copy code. GNU General Public License.