Advanced HD Video Encoding Guide

Not everything is going to be 100% correct and certain parts are missing.
`https://git.concertos.live/Encode_Guide/Encode_Guide/`

Contents

1 Tools

1.1 eac3to

While we don't like to recommend closed source software, there's no denying that `eac3to` [1] is a very useful tool for the demuxing process with no really good alternatives. If you don't want to download closed source software (let alone support it), you can also simply use MKVToolNix [2] without a separate demuxer.

For Windows users, simply download this from Doom9. Unix users will have to use Wine. When specifying the path, either use Winepath or swap out the slashes accordingly yourself.

1.2 VapourSynth

VapourSynth's installation is simple. There's an installer for Windows, Homebrew for Mac, and every distro's repository should have VapourSynth.

The VapourSynth documentation lists paths for where to place your plugins. The recommended path for Windows is `<AppData>\VapourSynth\plugins32` or `<AppData>\VapourSynth\plugins64`. Unix users can create a configuration file to specify paths.

Python scripts for VapourSynth should be placed in your Python `site-packages` folder. On Arch Linux, this is `/usr/lib64/Python3.*/sitepackages/`. Windows users will find this in their local AppData folder.

1.3 Editors

There are four popular editors for VapourSynth:

- VapourSynth Editor[3]
- Yuuno[4]
- VapourSynth Preview[5]
- AvsPmod[6]

Every Linux repository should have VSEdit. In Arch, for example, it's in `aur/vapoursynth-editor`. There's an installer for Windows, and Mac users should be able to install it via Homebrew.

Yuuno is an extension for Jupyter notebooks that allows you to edit and export VapourSynth scripts. You can install it via `$ pip install yuuno` and then `$ yuuno jupyter install`. Note that you'll need to have Jupyter or Jupyter Lab installed.

VapourSynth Preview requires a separate text editor or IDE to write the scripts, which makes it very useful for those who don't like the editor included in VSEdit.

AvsPmod is the editor used for AviSynth, but it also supports VapourSynth.

They all have their advantages and disadvantages, but for new users, I'd recommend VSEdit for local editing and yuuno for users looking to also write their scripts on servers. This is because

[1] https://forum.doom9.org/showthread.php?t=125966
[2] https://mkvtoolnix.download/
[3] https://bitbucket.org/mystery_keeper/vapoursynth-editor
[4] https://yuuno.encode.moe
[5] https://github.com/Endilll/vapoursynth-preview/
[6] https://avspmod.github.io/

Jupyter requires very little tinkering to work remotely. One tip for yuuno users would be to try out Jupyter Lab as a replacement for Jupyter.

Here are the most important differences between these two:

VSEdit

- Doesn't require having your browser open.
- Built-in benchmarking and encoding tools.
- Easy skimming through video via CTRL + SHIFT + ARROW KEYS.
- More stable.
- Barely under development anymore.
- Most private tracker users use this, hence it might be easier to get support.
- VapourSynth-specific syntax highlighting and suggestions.
- Allows you to store snippets and templates.

yuuno

- Very easy to use remotely.
- Easy exporting via iPython magic.
- Way better comparison tools via `%%vspreview clipa --diff clipb` ⇒ hovering over preview changes clip.
- Less mature and hence more prone to breaking.
- Allows you to work on and export multiple scripts from within one Jupyter Notebook.

1.4 x264 and x265

When installing x264 and x265, which are even easier to find than VapourSynth, you can also go with some mods. The most popular mod for x264 is tmod. Note worthy changes include `fade-compensate`, which can help encode fades with `mbtree` on, and more aq modes. Unless you're using one of these options, which almost nobody does, you can just use vanilla x264 instead. If you want to be even more precise when encoding, you can certainly give this a try and chain aq modes, since you might be able to squeeze a bit higher aq out of one of them to preserve dither while not ruining lines.

The most popular x265 mod, yukki, is even less impressive. All this includes is some cosmetic changes like a nicer ETA.

1.5 Opus, qAAC, and fdkaac

For encoding Opus, you need the `opus-tools` package.

qAAC requires iTunes libraries. On a Mac, this is no issue. Windows users can install iTunes or QuickTime. An alternative to this is the portable installer, which includes the necessary libraries with no iTunes requirements. Arch users can use the `qaac-wine` package from the AUR.

If you want to use fdkaac instead, this is a simple install everywhere.

1.6 MKVToolNix

Install the latest mkvtoolnix. This will install multiple tools, with the most important ones being `mkvtoolnix-gui` and `mkvextract`.

2 Demuxing

2.1 ffmpeg and bdinfo

For demuxing, there's a fantastic wrapper around `ffmpeg` called `bdinfo`[7]. Despite sharing its name with the other BDInfo program, it can't produce the same scans which many sites require. The README on the GitHub repository should suffice for use. You'll mainly need the `-x` and `-L` options.

2.2 50 fps, HEVC Seamless Branching, and Other eac3to Problems

If you encounter a Blu-ray with seamless branching (i.e. your playlists consist of hundreds of `m2ts` files played after each other) with HEVC streams, you'll have to use `makemkv` instead.

Other issues with `ffmpeg` and `bdinfo` (the errors are actually with `libbluray`) may also be solved by using `makemkv` or directly plugging the source into `mkvtoolnix`.

3 Video Encoding

3.1 Your First Script

A standard VapourSynth super basic script with no extra filtering done will look something like this:

```
# Uncomment the following line if you're using yuuno as apposed to VSEdit:
#%load_ext yuuno # This tells Jupyter to load yuuno.
#%%vspreview # This will allow you to preview the output.

# This section is for VSEdit #
import vapoursynth as vs
core = vs.core
##############################

src = core.ffms2.Source("/path/to/source.mkv") # For Windows paths, use
    r"C:\path\to\source.mkv".
out = core.std.Crop(src, top=138, bottom=138) # Crop off black bars as
    necessary.

out.set_output() # flags the variable as the output for vspipe
```

3.1.1 Resizing

Firstly, note that there'll be a separate section later on for descaling. Here, I'll explain resizing and what resizers are good for what.

If you want to resize, it's important to not alter the aspect ratio more than necessary. If you're downscaling, first figure out what the width and height should be. If you want to downscale to 720p, first crop, then figure out whether you're scaling to 720 height or 1280 width. If the former is the case, your width should be:

```
width = round(720 * src.height / (2 * src.width)) * 2
```

[7] https://github.com/schnusch/bdinfo

For the latter, the code to find your height is quite similar:

```
height = round(1280 * src.width / (2 * src.height)) * 2
```

You can also use the `cropresize` wrapper in `awsmfunc`[8] to do these calculations and resize.

There are many resizers available. The most important ones are:

- **Point** also known as nearest neighbor resizing, is the simplest resizer, as it doesn't really do anything other than enlargen each pixel or create the average of surrounding each pixel when downscaling. It produces awful results, but doesn't do any blurring when upscaling, hence it's very good for zooming in to check what each pixel's value is. It is also self-inverse, so you can scale up and then back down with it and get the same result you started with.
- **Bilinear** resizing is very fast, but leads to very blurry results with noticeable aliasing.
- **Bicubic** resizing is similarly fast, but also leads to quite blurry results with noticeable aliasing. You can modify parameters here for sharper results, but this will lead to even more aliasing.
- **Lanczos** resizing is slower and gets you very sharp results. However, it creates very noticeable ringing artifacts.
- **Blackmanminlobe** resizing, which you need to use `fmtconv`[9] for, is a modified lanczos resizer with less ringing artifacts. This resizer is definitely worth considering for upscaling chroma for YUV444 encodes (more on this later).
- **Spline** resizing is quite slow, but gets you very nice results. There are multiple spline resizers available, with `Spline16` being faster than `Spline36` with slightly worse results, and `Spline36` being similar enough to `Spline64` that there's no reason to use the latter. `Spline36` is the recommended resizer for downscaling content.
- **nnedi3**[10] resizing is quite slow and can only upscale by powers of 2. It can then be combined with `Spline36` to scale down to the desired resolution. Results are significantly better than the aforementioned kernels.
- **FSRCNNX**[11] is a shader for mpv, which can be used via the `vs-placebo`[12] plugin. It provides far sharper results than `nnedi3`, but requires a GPU. It's recommended to use this for upscaling if you can.

A comparison of these resizers can be found in the appendix under figure 16 for downscaling and figure 17 for upscaling. Additionally, as changing its parameters will have a very significant result on the output, a comparison of differently configured bicubic upscales is included after these in figure 18. For the extra curious, I've included a comparison of downscales scaled back to the original resolution in figure 19, as well as one showcasing the same resizer for downscaling then upscaling in figure 20 in the appendix.

While these screenshots should help you get a decent idea of the differences between resizers, they're only small parts of single images. If you want to get a better idea of how these resizers look, I recommend doing these upscales yourself, watching them in motion, and interleaving them (`std.Interleave`).

The difference between resizers when downscaling is a lot less noticeable than when upscaling. However, it's not recommended to use this as an excuse to be lazy with your choice of a resize kernel when downscaling.

TL;DR: Use `core.resize.Spline36` for downscaling.

[8] https://git.concertos.live/AHD/awsmfunc/
[9] https://github.com/EleonoreMizo/fmtconv/
[10] https://gist.github.com/4re/342624c9e1a144a696c6
[11] https://github.com/igv/FSRCNN-TensorFlow/releases
[12] https://github.com/Lypheo/vs-placebo

3.2 Filtering

There are a few things worth mentioning here. First of all, most Blu-rays come in YUV420P8 with limited range. The first group of information here is YUV. This stands for the planes our video has, with Y being luma and U and V being our chroma planes.

The color after that, 4:2:0 in this case, specifies the size of our planes. The three most common variations of this are 4:2:0, which means that chroma planes are half the size of the luma plane (e.g. a 1920×1080 video will have chroma planes in 960×540), 4:2:2, which means that the chroma planes are half the size on their horizontal axis and full size on the vertical axis, and 4:4:4, which means all planes are the same size. During playback, video players scale the chroma planes up to the same size as the luma. Smaller chroma planes aren't as noticeable, but one can certainly tell the difference when upscaling. To illustrate this, here's an example from AnoHana with one image being 4:2:0 720p upscaled to 1080p and the other being 4:4:4 720p upscaled to 1080p:

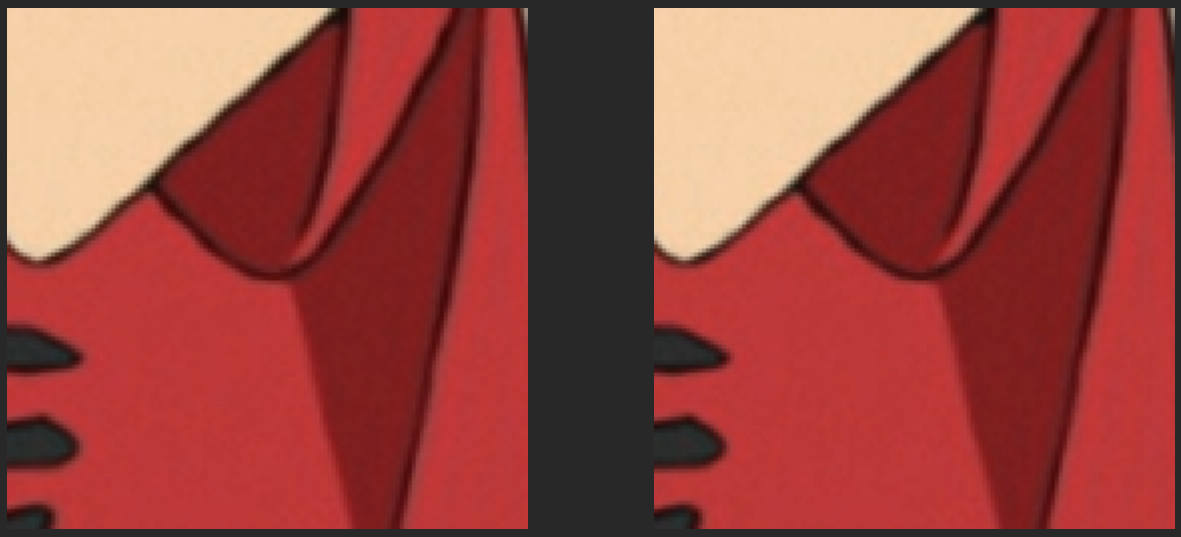

Figure 1: YUV420 on left, YUV444 on right. Descaled with `fvf.Debicubic`, then upscaled with `nnedi3_rpow2.nnedi3_rpow2` and `resize.Bilinear`. Zoomed in with a factor of 2.

In this case, the worst resizer (bilinear) was used for the chroma planes. If you use something like mpv with KrigBilateral, this should look far better.

Another example: the Parasite SDR UHD in 4:4:4 1080p compared to 4:2:0 1080p:

Figure 2: Parasite SDR UHD scaled to 1080p 4:2:0 compared to 4:4:4. Thanks go out to HyerrDoktyer for this example.

The next piece of information to explain is P8. This refers to the bit depth, which is 8-bit in this case. Most video is stored in 8-bit nowadays, as 8-bit AVC has the best hardware compatibility. However, as 8-bit leads to not having enough values (0-255) available, it easily introduces errors such as banding. Higher bit depths don't have such big issues with this, and due to their better accuracy, some (i.e. 10-bit) are actually better at storing video at lower file sizes. However, 10-bit AVC has almost no hardware compatibility and takes a lot longer to encode, hence people on private trackers usually dislike it. Most content is actually produced in either 10-bit or 12-bit. The most popular bit depth for filtering is 16-bit due to increased accuracy. It's worth noting that UHD Blu-rays will be in YUV420P10, meaning 10-bit.

The final piece to the puzzle here is limited range. In full range 8-bit, we have every value between 0 and 255 available. However, TVs usually can't display all these values and are limited to a range of 16 to 235 for luma and 16 to 240 for chroma. Lots of consumer content, e.g. Blu-rays are also in limited range. If you give a TV a full range video, it will simply make everything 16 and under or 235/240 and over the same value (e.g. black or white).

3.2.1 Checking Your Source

This is what will probably take up the most time: checking your source for issues. This requires going through your entire source file and checking with your own eyes for banding, aliasing, dirty lines, and whatever other issues might be present. The good news is that VSEdit allows you to simply go through your source by a self defined step via the CTRL + SHIFT + LEFT/RIGHT ARROW keys. This step can be defined in the bottom right hand corner of the preview window. I'd recommend a step between 1 s and 3 s. Obviously, the slower the better, since you'll be covering more frames.

In order to apply a filter to a given range, use something like the following:

```
filtered = my_filter(src)
out = awf.rfs(src, filtered, mappings="[FIRST_FRAME LAST_FRAME]")
```

3.2.2 Dithering

Although not necessary to work in if you're exporting in the same bit depth as your source, working in high bit depth and dithering down at the end of your filter chain is recommended in order to avoid rounding errors, which can lead to artifacts such as banding (an example is in figure 23). Luckily, if you choose not to write your script in high bit depth, most plugins will work in high bit depth internally. As dithering is quite fast and higher depths do lead to better precision, there's usually no reason not to work in higher bit depths other than some functions written for 8-bit being slightly slower.

If you'd like to learn more about dithering, the Wikipedia page[13] is quite informative. There are also a lot of research publications worth reading. What you need to understand here is that your dither method only matters if there's an actual difference between your source and the filtering you perform. As dither is an alternative of sorts to rounding to different bit depths, only offsets from actual integers will have differences. Some algorithms might be better at different things from others, hence it can be worth it to go with non-standard algorithms. For example, if you want to deband something and export it in 8-bit but are having issues with compressing it properly, you might want to consider ordered dithering, as it's known to perform slightly better in this case (although it doesn't look as nice). To do this, use the following code:

```
source_16 = fvf.Depth(src, 16)
```

[13] https://en.wikipedia.org/wiki/Dither

```
deband = core.f3kdb.Deband(source_16, output_depth=16)

out = fvf.Depth(deband, 8, dither='ordered')
```

Again, this will only affect the actual debanded area. This isn't really recommended most of the time, as ordered dither is rather unsightly, but it's certainly worth considering if you're having trouble compressing a debanded area. You should obviously be masking and adjusting your debander's parameters, but more on that later.

In order to dither up or down, you can use the `Depth` function within either `fvsfunc`[14] (fvf) or `mvsfunc`[15] (mvf). The difference between these two is that fvf uses internal resizers, while mvf uses internal whenever possible, but also supports `fmtconv`, which is slower but has more dither (and resize) options. Both feature the standard Filter Lite error_diffusion dither type, however, so if you just roll with the defaults, I'd recommend fvf. To illustrate the difference between good and bad dither, some examples are included in the appendix under figure 21. Do note you may have to zoom in quite far to spot the difference. Some PDF viewers may also incorrectly output the image.

I'd recommend going with Filter Lite (fvf's default or `mvf.Depth(dither=3)`, also the default) most of the time. Others like Ostromoukhov (`mvf.Depth(dither=7)`), void and cluster (`fmtc.bitdepth(dither=8)`), standard Bayer ordered (`fvf.Depth(dither='ordered')` or `mvf.Depth(dither=0)`) can also be useful sometimes. Filter Lite will usually be fine, though.

3.2.3 Debanding

This is the most common issue one will encounter. Banding happens when bitstarving and poor settings lead to smoother gradients becoming abrupt color changes, which obviously ends up looking bad. The good news is higher bit depths can help with this issue, since more values are available to create the gradients. Because of this, lots of debanding is done in 16 bit, then dithered down to 10 or 8 bit again after the filter process is done.
One important thing to note about debanding is that you should try to always use a mask with it, preferably an edge mask or similar. See 3.2.14 for details!
There are two great tools for VapourSynth that are used to fix banding: `f3kdb`[16] and `fvsfunc`'s `gradfun3`. The latter one is less commonly used, but does have a built in mask.
Let's take a look at `f3kdb` first: The default relevant code for VapourSynth looks as follows:

```
deband = core.f3kdb.deband(src = clip, range = 15, y = 64, cb = 64, cr =
   64, grainy = 64, grainc = 64, dynamic_grain = False, output_depth = 8)
```

These settings may come off as self-explanatory for some, but here's what they do:

- `src` This is obviously your source clip.
- `range` This specifies the range of pixels that are used to calculate whether something is banded. A higher range means more pixels are used for calculation, meaning it requires more processing power. The default of 15 should usually be fine.
- `y` The most important setting, since most (noticeable) banding takes place on the luma plane. It specifies how big the difference has to be for something on the luma plane to be considered as banded. You should start low and slowly but surely build this up until the banding is gone. If it's set too high, lots of details will be seen as banding and hence be blurred.

[14] https://github.com/Irrational-Encoding-Wizardry/fvsfunc
[15] https://github.com/HomeOfVapourSynthEvolution/mvsfunc/
[16] https://forum.doom9.org/showthread.php?t=161411

- `cb` and `cr` The same as `y` but for chroma. However, banding on the chroma planes is quite uncommon, so you can often leave this off.
- `grainy` and `grainc` In order to keep banding from re-occurring and to counteract smoothing, grain is usually added after the debanding process. However, as this fake grain is quite noticeable, it's recommended to be conservative. Alternatively, you can use a custom grainer, which will get you a far nicer output (see 3.2.10).
- `dynamic_grain` By default, grain added by `f3kdb` is static. This compresses better, since there's obviously less variation, but it usually looks off with live action content, so it's normally recommended to set this to `True` unless you're working with animated content.
- `output_depth` You should set this to whatever the bit depth you want to work in after debanding is. If you're working in 8 bit the entire time, you can just leave out this option.

Here's an example of very simple debanding:

Figure 3: Source on left, `deband = core.f3kdb.Deband(src, y=64, cr=0, cb=0, grainy=32, grainc=0, range=15, keep_tv_range=True)` on right. Zoomed in with a factor of 2.

If you want to use `f3kdb` together with the `gradfun3` mask, you can use `fag3kdb`[17]. I'd recommend trying this for very strong debanding, but `retinex_edgemask` or `debandmask` is better most of the time (albeit the former is a lot slower).

The most commonly used alternative is `gradfun3`, which is likely mainly less popular due to its less straightforward parameters. It works by smoothing the source, limiting this via `mvf.LimitFilter` to the values specified by `thr` and `elast`, then merging with the source via its internal mask (although using an external mask is also possible). While it's possible to use it for descaling, most prefer to do so in another step.
Many people believe `gradfun3` produces smoother results than `f3kdb`. As it has more options than `f3kdb` and one doesn't have to bother with masks most of the time when using it, it's certainly worth knowing how to use:

```
import fvsfunc as fvf
deband = fvf.GradFun3(src, thr=0.35, radius=12, elast=3.0, mask=2, mode=3,
   ampo=1, ampn=0, pat=32, dyn=False, staticnoise=False, smode=2,
```

[17] Fag3kdb: https://gist.github.com/Frechdachs/9d9b50d050fa11e438eae5d967296e0e

```
    thr_det=2 + round(max(thr - 0.35, 0) / 0.3), debug=False, thrc=thr,
    radiusc=radius, elastc=elast,
    planes=list(range(src.format.num_planes)), ref=src,
    bits=src.format.bits_per_sample) # + resizing variables
```

Lots of these values are for `fmtconv` bit depth conversion, so its documentation[18] can prove to be helpful. Descaling in `GradFun3` isn't much different from other descalers, so I won't discuss this. Some of the other values that might be of interest are:

- `thr` is equivalent to `y`, `cb`, and `cr` in what it does. You'll likely want to raise or lower it.
- `radius` has the same effect as `f3kdb`'s `range`.
- `smode` sets the smooth mode. It's usually best left at its default, a bilateral filter[19], or set to 5 if you'd like to use a CUDA-enabled GPU instead of your CPU. Uses `ref` (defaults to input clip) as a reference clip.
- `mask` disables the mask if set to 0. Otherwise, it sets the amount of `std.Maximum` and `std.Minimum` calls to be made.
- `planes` sets which planes should be processed.
- `mode` is the dither mode used by `fmtconv`.
- `ampn` and `staticnoise` set how much noise should be added by `fmtconv` and whether it should be static. Worth tinkering with for live action content.
- `debug` allows you to view the mask.
- `elast` is "the elasticity of the soft threshold." Higher values will do more blending between the debanded and the source clip.

If your debanded clip had very little grain compared to parts with no banding, you should consider using a separate function to add matched grain so the scenes blend together easier. If there was lots of grain, you might want to consider `adptvgrnMod`, `adaptive_grain` or `GrainFactory3`; for less obvious grain or simply for brighter scenes where there'd usually be very little grain, you can also use `grain.Add`. This topic will be further elaborated later in 3.2.10.

[18] `fmtc.bitdepth` documentation: `https://github.com/EleonoreMizo/fmtconv/blob/master/doc/fmtconv.html#L262`

[19] `https://en.wikipedia.org/wiki/Bilateral_filter`

Here's an example from Mirai (full script later):

Figure 4: Source on left, filtered on right. Banding might seem hard to spot here, but I can't include larger images in this PDF. The idea should be obvious, though.

If you want to automate your banding detection, you can use a detection function based on `bandmask`[20] called `banddtct`[21]. Make sure to adjust the values properly and check the full output. A forum post explaining it is linked in 3.2.17. You can also just run `adptvgrnMod` or `adaptive_grain` with a high `luma_scaling` value in hopes that the grain covers it up fully. More on this in 3.2.10.

3.2.4 Fixing Dirty Lines and Improper Borders

Another very common issue, at least with live action content, is dirty lines. These are usually found on the borders of video, where a row or column of pixels exhibits usually a too low luma value compared to its surrounding rows. Oftentimes, this is the due to improper downscaling, more notably downscaling after applying borders. Dirty lines can also occur because video editors often won't know that while they're working in YUV422, meaning their height doesn't have to be mod2, consumer products will be YUV420, meaning the height has to be mod2, leading to extra black rows.
Another form of dirty lines is exhibited when the chroma planes are present on black bars. Usually, these should be cropped out. The opposite can also occur, however, where the planes with legitimate luma information lack chroma information.
There are six commonly used filters for fixing dirty lines:

- `cf`'s[22] `ContinuityFixer`
  `ContinuityFixer` works by comparing the rows/columns specified to the amount of rows/columns specified by `range` around it and finding new values via least squares regression. Its settings look as follows:

```
fix = core.cf.ContinuityFixer(src=clip, left=[0, 0, 0], right=[0, 0,
    0], top=[0, 0, 0], bottom=[0, 0, 0], radius=1920)
```

[20] https://gitlab.com/snippets/1904934
[21] https://git.concertos.live/AHD/awsmfunc/src/branch/detect/detect.py
[22] https://gitlab.com/Ututu/VS-ContinuityFixer

This is assuming you're working with 1080p footage, as `radius`'s value is set to the longest set possible as defined by the source's resolution. I'd recommend a lower value, although not going much lower than 3, as at that point, you may as well be copying pixels (see `FillBorders` below for that). What will probably throw off most newcomers is the array I've entered as the values for rows/columns to be fixed. These denote the values to be applied to the three planes. Usually, dirty lines will only occur on the luma plane, so you can often leave the other two at a value of 0. Do note an array is not necessary, so you can also just enter the amount of rows/columns you'd like the fix to be applied to, and all planes will be processed.
One thing `ContinuityFixer` is quite good at is getting rid of irregularities such as dots. It's also faster than `bbmod` and `FixBrightnessProtect2`, but it should be considered a backup for those two.

- `awsmfunc`'s `bbmod`
  This is a mod of the original BalanceBorders function. It's pretty similar to `ContinuityFixer`, but will lead to better results with high `blur` and `thresh` values. If it doesn't produce decent results, these can be changed, but the function will get more destructive the lower you set the `blur` value. It's also significantly faster than the versions in `havsfunc` and `sgvsfunc` as only necessary pixels are processed.

```
import awsmfunc as awf
bb = awf.bbmod(src=clip, left=0, right=0, top=0, bottom=0,
   thresh=[128, 128, 128], blur=[20, 20, 20], scale_thresh=False,
   cpass2=False)
```

The arrays for `thresh` and `blur` are again y, u, and v values. It's recommended to try `blur=999` first, then lowering that and `thresh` until you get decent values.
`thresh` specifies how far the result can vary from the input. This means that the lower this is, the better. `blur` is the strength of the filter, with lower values being stronger, and larger values being less aggressive. If you set `blur=1`, you're basically copying rows.

- `fb`'s[23] `FillBorders`
  This function pretty much just copies the next column/row in line. While this sounds, silly, it can be quite useful when downscaling leads to more rows being at the bottom than at the top, and one having to fill one up due to YUV420's mod2 height.

```
fill = core.fb.FillBorders(src=clip, left=0, right=0, bottom=0, top=0,
   mode="fillmargins")
```

A very interesting use for this function is one similar to applying `ContinuityFixer` only to chroma planes, which can be used on gray borders or borders that don't match their surroundings no matter what luma fix is applied. This can be done with the following script:

```
fill = core.fb.FillBorders(src=clip, left=0, right=0, bottom=0, top=0,
   mode="fillmargins")
merge = core.std.Merge(clipa=clip, clipb=fill, weight=[0,1])
```

You can also split the planes and process the chroma planes individually, although this is only slightly faster. A wrapper that allows you to specify per-plane values for `fb` is `FillBorders` in `awsmfunc`.

- `edgefixer`'s[24] `ReferenceFixer`
  This requires the original version of `edgefixer` (`cf` is just an old port of it, but it's nicer to use and processing hasn't changed). I've never found a good use for it, but in theory, it's quite neat. It compares with a reference clip to adjust its edge fix.:

[23] https://github.com/Moiman/vapoursynth-fillborders
[24] https://github.com/sekrit-twc/EdgeFixer

```
fix = core.edgefixer.Reference(src, ref, left=[0, 0, 0], right=[0, 0,
   0], top=[0, 0, 0], bottom=[0, 0, 0], radius = 1920)
```

- `rekt`'s `rektlvls`[25]
  This is basically `FixBrightnessProtect2` and `FixBrightness` in one with the additional fact that not the entire frame is processed. Its values are quite straightforward. Raise the adjustment values to brighten, lower to darken. Set `prot_val` to zero and it will function like `FixBrightness`, meaning the adjustment values will need to be changed.

```
from rekt import rektlvls
fix = rektlvls(src, rownum=None, rowval=None, colnum=None,
   colval=None, prot_val=20)
```

  If you'd like to process multiple rows at a time, you can enter a list (e.g. `rownum=[0, 1, 2]`).

One thing that shouldn't be ignored is that applying these fixes (other than `rektlvls`) to too many rows/columns may lead to these looking blurry on the end result. Because of this, it's recommended to use `rektlvls` whenever possible or carefully apply light fixes to only the necessary rows. If this fails, it's better to try `bbmod` before using `ContinuityFixer`.

It's important to note that you should *always* fix dirty lines before resizing, as not doing so will introduce even more dirty lines. However, it is important to note that, if you have a single black line at an edge that you would use `FillBorders` on, you should remove that using your resizer. For example, to resize a clip with a single filled line at the top to 1280×536 from 1920×1080:

```
top_crop = 138
bot_crop = 138
top_fill = 1
bot_fill = 0
src_height = src.height - (top_crop + bot_crop) - (top_fill + bot_fill)
crop = core.std.Crop(src, top=top_crop, bottom=bot_crop)
fix = core.fb.FillBorders(crop, top=top_fill, bottom=bot_fill,
   mode="fillmargins")
resize = core.resize.Spline36(1280, 536, src_top=top_fill,
   src_height=src_height)
```

If you're dealing with diagonal borders, the proper approach here is to mask the border area and merge the source with a `FillBorders` call. An example of this (from the D-Z0N3 encode of Your Name (2016)):

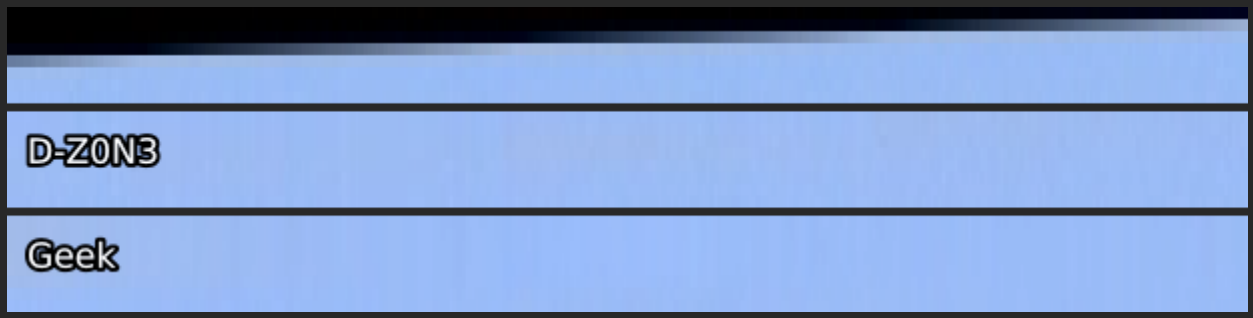

Figure 5: Example of improper borders from Your Name. D-Z0N3 is masked, Geek is unmasked. As such, Geek lacks any resemblance of grain, while D-Z0N3 keeps it in tact whenever possible. It may have been smarter to use the `mirror` mode in `FillBorders`, but hindsight is 20/20.

[25] `https://gitlab.com/Ututu/rekt`

Code used by D-Z0N3 (in 16-bit):

```
mask = core.std.ShufflePlanes(src, 0, vs.GRAY).std.Binarize(43500)
cf = core.fb.FillBorders(src, top=6).std.MaskedMerge(src, mask)
```

An example of why you should be masking this is in the appendix under figure 22.

To illustrate what dirty lines might look like, here's an example of `ContinuityFixer` and chroma-only `FillBorders`:

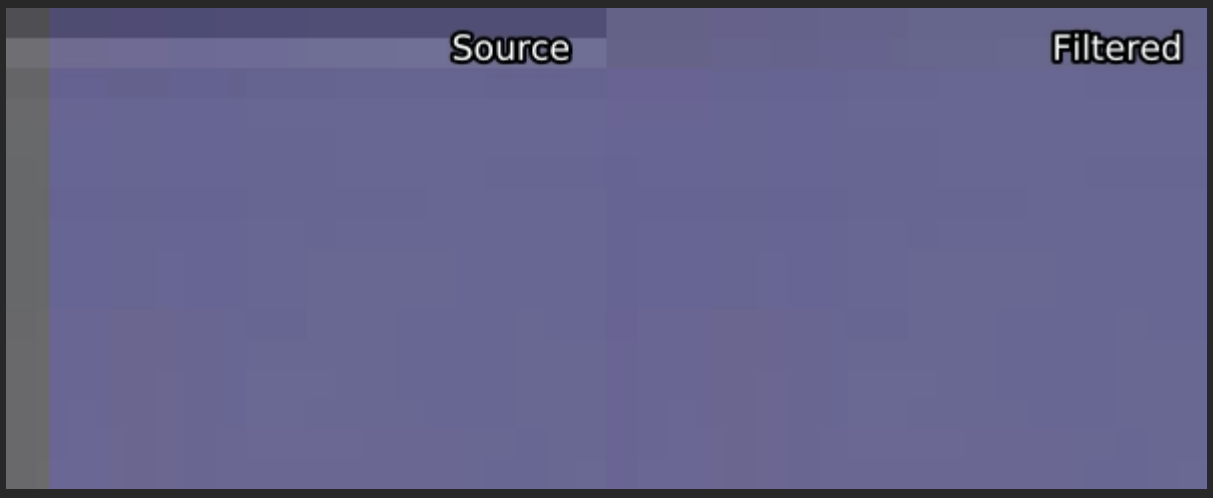

Figure 6: Source vs filtered of a dirty line fix from the D-Z0N3 encode of A Silent Voice. Used `ContinuityFixer` on top three rows and `FillBorders` on the two leftmost columns. Zoomed in with a factor of 15.

Dirty lines can be quite difficult to spot. If you don't immediately spot any upon examining borders on random frames, chances are you'll be fine. If you know there are frames with small black borders on each side, you can use something like the following script[26]:

```
def black_detect(clip, thresh=None):
    if thresh == None:
        thresh = (25 * (1 << clip.format.bits_per_sample) - 1) / 255
    mask = core.std.ShufflePlanes(clip, 0, vs.GRAY).std.Binarize(
        "{0}".format(thresh)).std.Invert().std.Maximum().std.Inflate(
            ).std.Maximum().std.Inflate()
    l = core.std.Crop(mask, right=clip.width / 2)
    r = core.std.Crop(mask, left=clip.width / 2)
    mask_test = core.std.StackHorizontal([r, l])
    t = core.std.Crop(mask_test, top=clip.height / 2)
    b = core.std.Crop(mask_test, bottom=clip.height / 2)
    mask_test = core.std.StackVertical([t, b])
    return mask_test
```

This script will make values under the threshold value (i.e. the black borders) show up as vertical or horizontal white lines in the middle on a mostly black background. You can just skim through your video with this active. You can also try to use `blckdtct`[27], which scans the video for you.

Other kinds of variable dirty lines are a bitch to fix and require checking scenes manually.

An issue very similar to dirty lines is bad borders. During scenes with different crops (e.g. IMAX or 4:3), the black borders may sometimes not be entirely black, or be completely messed up. In order to fix this, simply crop them and add them back. You may also want to fix dirty lines that may have occurred along the way:

[26] https://gitlab.com/snippets/1834089
[27] https://git.concertos.live/AHD/awsmfunc/src/branch/detect/detect.py

```
crop = core.std.Crop(src, left=100, right=100)
clean = core.cf.ContinuityFixer(crop, left=2, right=2, top=0, bottom=0,
   radius=25)
out = core.std.AddBorders(clean, left=100, right=100)
```

3.2.5 Anti-Aliasing

This is likely the most commonly known issue. If you want to fix this, first make sure the issue stems from actual aliasing, not poor upscaling. If you've done this, the tool I'd recommend is the `TAAmbk` suite[28]:

```
import vsTAAmbk as taa

aa = taa.TAAmbk(clip, aatype=1, aatypeu=None, aatypev=None, preaa=0,
   strength=0.0, cycle=0,
            mtype=None, mclip=None, mthr=None, mthr2=None, mlthresh=None,
            mpand=(1, 0), txtmask=0, txtfade=0, thin=0, dark=0.0, sharp=0,
               aarepair=0,
            postaa=None, src=None, stabilize=0, down8=True, showmask=0,
               opencl=False,
            opencl_device=0, **args)
```

The GitHub README is quite extensive, but some additional comments are necessary:

- `aatype`: (Default: 1)
  The value here can either be a number to indicate the AA type for the luma plane, or it can be a string to do the same.

  ```
        0: lambda clip, *args, **kwargs: type('', (), {'out': lambda:
           clip}),
        1: AAEedi2,
        2: AAEedi3,
        3: AANnedi3,
        4: AANnedi3UpscaleSangNom,
        5: AASpline64NRSangNom,
        6: AASpline64SangNom,
        -1: AAEedi2SangNom,
        -2: AAEedi3SangNom,
        -3: AANnedi3SangNom,
        'Eedi2': AAEedi2,
        'Eedi3': AAEedi3,
        'Nnedi3': AANnedi3,
        'Nnedi3UpscaleSangNom': AANnedi3UpscaleSangNom,
        'Spline64NrSangNom': AASpline64NRSangNom,
        'Spline64SangNom': AASpline64SangNom,
        'Eedi2SangNom': AAEedi2SangNom,
        'Eedi3SangNom': AAEedi3SangNom,
        'Nnedi3SangNom': AANnedi3SangNom,
        'PointSangNom': AAPointSangNom,
  ```

  The ones I would suggest are `Eedi3`, `Nnedi3`, `Spline64SangNom`, and `Nnedi3SangNom`. Both of the `SangNom` modes are incredibly destructive and should only be used if absolutely necessary. `Nnedi3` is usually your best option; it's not very strong or destructive, but often good enough, and is fairly fast. `Eedi3` is unbelievably slow, but stronger than `Nnedi3` and not as destructive as the `SangNom` modes.

[28] https://github.com/HomeOfVapourSynthEvolution/vsTAAmbk

- `aatypeu`: (Default: same as `aatype`)
  Select main AA kernel for U plane when clip's format is YUV.
- `aatypev`: (Default: same as `aatype`)
  Select main AA kernel for V plane when clip's format is YUV.
- `strength`: (Default: 0)
  The strength of predown. Valid range is $[0, 0.5]$ Before applying main AA kernel, the clip will be downscaled to $(1-\text{strength})\times\text{clip\_resolution}$ first and then be upscaled to original resolution by main AA kernel. This may be benefit for clip which has terrible aliasing commonly caused by poor upscaling. Automatically disabled when using an AA kernel which is not suitable for upscaling. If possible, do not raise this, and *never* lower it.
- `preaa`: (Default: 0)
  Select the preaa mode.
  - 0: No preaa
  - 1: Vertical
  - 2: Horizontal
  - -1: Both

  Perform a `preaa` before applying main AA kernel. `Preaa` is basically a simplified version of `daa`. Pretty useful for dealing with residual comb caused by poor deinterlacing. Otherwise, don't use it.
- `cycle`: (Default: 0)
  Set times of loop of main AA kernel. Use for very very terrible aliasing and 3D aliasing.
- `mtype`: (Default: 1)
  Select type of edge mask to be used. Currently there are three mask types:
  - 0: No mask
  - 1: `Canny` mask
  - 2: `Sobel` mask
  - 3: `Prewitt` mask

  Mask always be built under 8-bit scale. All of these options are fine, but you may want to test them and see what ends up looking the best.
- `mclip`: (Default: None)
  Use your own mask clip instead of building one. If `mclip` is set, script won't build another one. And you should take care of mask's resolution, bit-depth, format, etc by yourself.
- `mthr`:
  Size of the mask. The smaller value you give, the bigger mask you will get.
- `mlthresh`: (Default None)
  Set luma thresh for n-pass mask. Use a list or tuple to specify the sections of luma.
- `mpand`: (Default: (1, 0))
  Use a list or tuple to specify the loop of mask expanding and mask inpanding.
- `txtmask`: (Default: 0)
  Create a mask to protect white captions on screen. Value is the threshold of luma. Valid range is 0 255. When a area whose luma is greater than threshold and chroma is 128 ± 2, it will be considered as a caption.
- `txtfade`: (Default: 0)
  Set the length of fading. Useful for fading text.
- `thin`: (Default: 0)
  Warp the line by aWarpSharp2 before applying main AA kernel.

- `dark`: (Default: 0.0)
  Darken the line by Toon before applying main AA kernel.
- `sharp`: (Default: 0)
  Sharpen the clip after applying main AA kernel. * 0: No sharpen. * 1 inf: LSFmod(defaults='old') * 0 1: Simlar to Avisynth's sharpen() * -1 0: LSFmod(defaults='fast') * -1: Contra-Sharpen

  Whatever type of sharpen, larger absolute value of sharp means larger strength of sharpen.
- `aarepair`: (Default: 0)
  Use repair to remove artifacts introduced by main AA kernel. According to different repair mode, the pixel in src clip will be replaced by the median or average in 3x3 neighbour of processed clip. It's highly recommend to use repair when main AA kernel contain SangNom. For more information, check `http://www.vapoursynth.com/doc/plugins/rgvs.html#rgvs.Repair`. Hard to get it to work properly.
- `postaa`: (Default: False)
  Whether use soothe to counter the aliasing introduced by sharpening.
- `src`: (Default: clip)
  Use your own `src` clip for sharp, repair, mask merge, etc.
- `stabilize`: (Default: 0)
  Stabilize the temporal changes by `MVTools`. Value is the temporal radius. Valid range is $[0, 3]$.
- `down8`: (Default: True)
  If you set this to `True`, the clip will be down to 8-bit before applying main AA kernel and up it back to original bit-depth after applying main AA kernel. `LimitFilter` will be used to reduce the loss in depth conversion.
- `showmask`: (Default: 0)
 Output the mask instead of processed clip if you set it to not 0. 0: Normal output; 1: Mask only; 2: tack mask and clip; 3: Interleave mask and clip; -1: Text mask only
- `opencl`: (Default: False) Whether use opencl version of some plugins. Currently there are three plugins that can use opencl:
  - TCannyCL
  - EEDI3CL
  - NNEDI3CL

  This may speed up the process, which is obviously great, since anti-aliasing is usually very slow.
- `opencl_device`: (Default: 0)
  Select an opencl device. To find out which one's the correct one, do

  ```
  core.nnedi3cl.NNEDI3CL(clip, 1, list_device=True).set_output()
  ```

- `other parameters`:
  Will be collected into a dict for particular aatype.

Note that there are a ton more very good anti-aliasing methods, as well as many different mask types you can use (e.g. other edgemasks, clamping one method's changes to those of another method etc.). However, most methods are based on very similar methods to those `TAA` implements.

If your entire video suffers from aliasing, it's not all too unlikely that you're dealing with a cheap upscale. In this case, descale or resize first before deciding whether you need to perform any anti-aliasing.

Here's an example of an anti-aliasing fix from Non Non Biyori - Vacation:

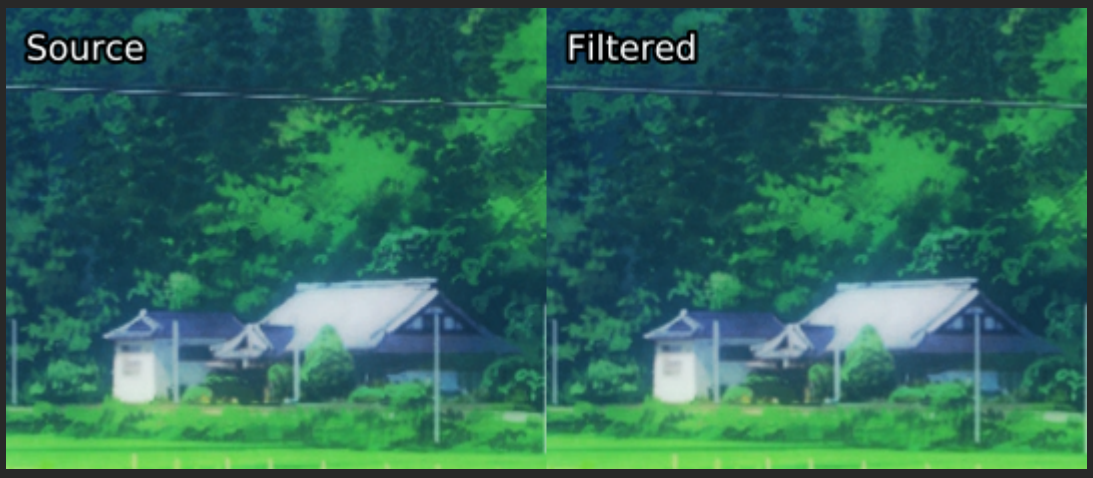

Figure 7: Source with aliasing on left, filtered on right.

In this example, the following was done:

```
mask = kgf.retinex_edgemask(src).std.Binarize(65500).std.Maximum
   ().std.Inflate()
aa = taa.TAAmbk(src, aatype=2, mtype=0, opencl=True)
out = core.std.MaskedMerge(src, aa, mask)
```

3.2.6 Descaling

While most movies are produced at 2K resolution and most anime are made at 720p, Blu-rays are almost always 1080p and UHD Blu-rays are all 4K. This means the mastering house often has to upscale footage. These upscales are pretty much always terrible, but luckily, some are reversible. Since anime is usually released at a higher resolution than the source images, and bilinear or bicubic upscales are very common, most descalers are written for anime, and it's the main place where you'll need to descale. Live action content usually can't be descaled because of bad proprietary scalers (often QTEC or the likes), hence most live action encoders don't know or consider descaling.

So, if you're encoding anime, always make sure to check what the source images are. You can use `https://anibin.blogspot.com/` for this, run screenshots through `getnative`[29], or simply try it out yourself. The last option is obviously the best way to go about this, but getnative is usually very good, too, and is a lot easier. Anibin, while also useful, won't always get you the correct resolution.

In order to perform a descale, you should be using `fvsfunc`:

```
import fvsfunc as fvf

descaled = fvf.Debilinear(src, 1280, 720, yuv444=False)
```

In the above example, we're descaling a bilinear upscale to 720p and downscaling the chroma with `Spline36` to 360p. If you're encoding anime for a site/group that doesn't care about hardware compatibility, you'll probably want to turn on `yuv444` and change your encode settings accordingly.

`Descale` supports bilinear, bicubic, and spline upscale kernels. Each of these, apart from `Debilinear`, also has its own parameters. For `Debicubic`, these are:

[29] `https://github.com/Infiziert90/getnative`

- `b`: between 0 and 1, this is equivalent to the blur applied.
- `c`: also between 0 and 1, this is the sharpening applied.

The most common cases are `b=1/3` and `c=1/3`, which are the default values, `b=0` and `c=1`, which is oversharpened bicubic, and `b=1` and `c=0`, which is blurred bicubic. In between values are quite common, too, however.

Similarly, `Delanczos` has the `taps` option, and spline upscales can be reversed for `Spline36` upscales with `Despline36` and `Despline16` for `Spline16` upscales.

Once you've descaled, you might want to upscale back to the 1080p or 2160p. The preferred way of doing this is via `nnedi3` or more specifically, `edi3_rpow2` or `nnedi3_rpow2`:

```
from edi3_rpow2 import nnedi3_rpow2

descaled = fvf.Debilinear(src, 1280, 720)
upscaled = nnedi3_rpow2(descaled, 2).resize.Spline36(1920, 1080)
out = core.std.Merge(upscaled, src, [0, 1])
```

What we're doing here is descaling a bilinear upscale to 720p, then using `nnedi3` to upscale it to 1440p, downscaling that back to 1080p, then merging it with the source's chroma. There are multiple reasons you might want to do this:

- Most people don't have a video player set up to properly upscale the footage.
- Those who aren't very informed often think higher resolution = better quality, hence 1080p is more popular.
- Lots of private trackers only allow 720p and 1080p footage. Maybe you don't want to hurt the chroma or the original resolution is in between (810p and 900p are very common) and you want to upscale to 1080p instead of downscaling to 720p.

Another important thing to note is that credits and other text is often added after upscaling, hence you need to use a mask to not ruin these. Luckily, you can simply add an `M` after the descale name (`DebilinearM`) and you'll get a mask applied. However, this significantly slows down the descale process, so you may want to scenefilter here.

On top of the aforementioned common descale methods, there are a few more filters worth considering, although they all do practically the same thing, which is downscaling line art (aka edges) and rescaling them to the source resolution. This is especially useful if lots of dither was added after upscaling.

- `DescaleAA`: part of `fvsfunc`, uses a `Prewitt` mask to find line art and rescales that.
- `InsaneAA`: Uses a strengthened `Sobel` mask and a mixture of both `eedi3` and `nnedi3`.

Personally, I don't like upscaling it back and would just stick with a YUV444 encode. If you'd like to do this, however, you can also consider trying to write your own mask. An example would be (going off the previous code):

```
mask = kgf.retinex_edgemask(src).std.Binarize(15000).std.Inflate()
new_y = core.std.MaskedMerge(src, upscaled, mask)
new_clip = core.std.ShufflePlanes([new_y, u, v], [0, 0, 0], vs.YUV)
```

In order to illustrate the difference, here are examples of rescaled footage. Do note that YUV444 downscales scaled back up by a video player will look better.

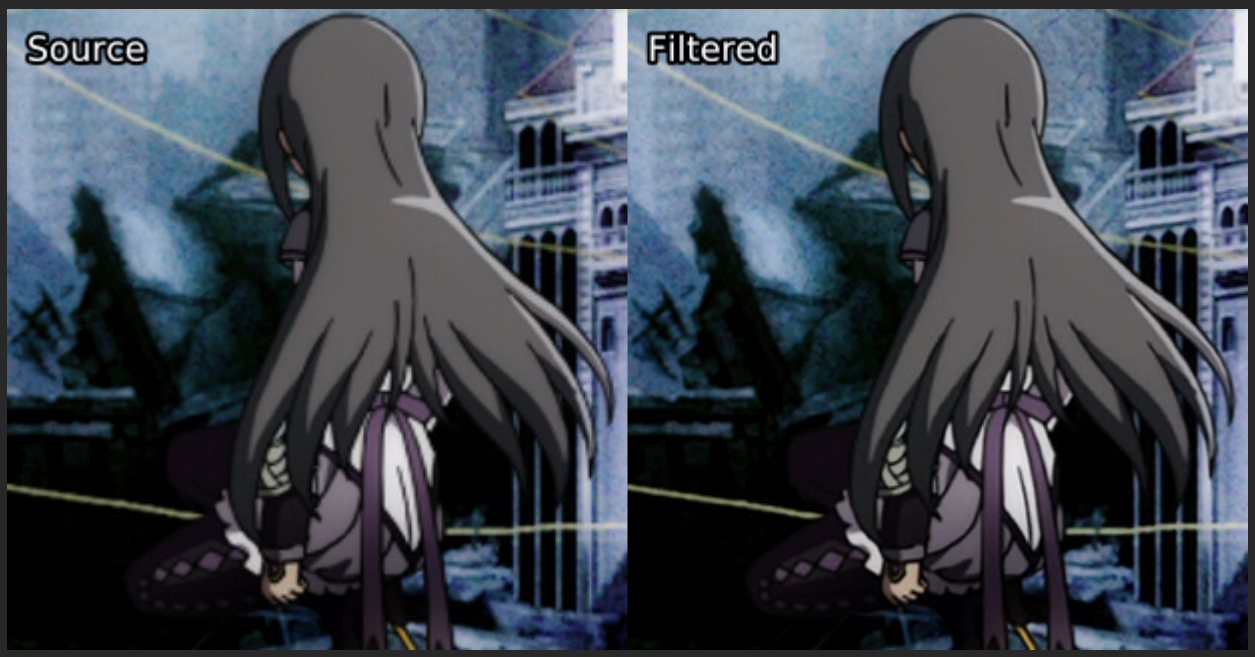

Figure 8: Source Blu-ray with 720p footage upscaled to 1080p via a bicubic filter on the left, rescale with `Debicubic` and `nnedi3` on the right.

It's important to note that this is certainly possible with live action footage as well. An example would be the Game of Thrones season 1 UHD Blu-rays, which are bilinear upscales. While not as noticeable in screenshots, the difference is stunning during playback.

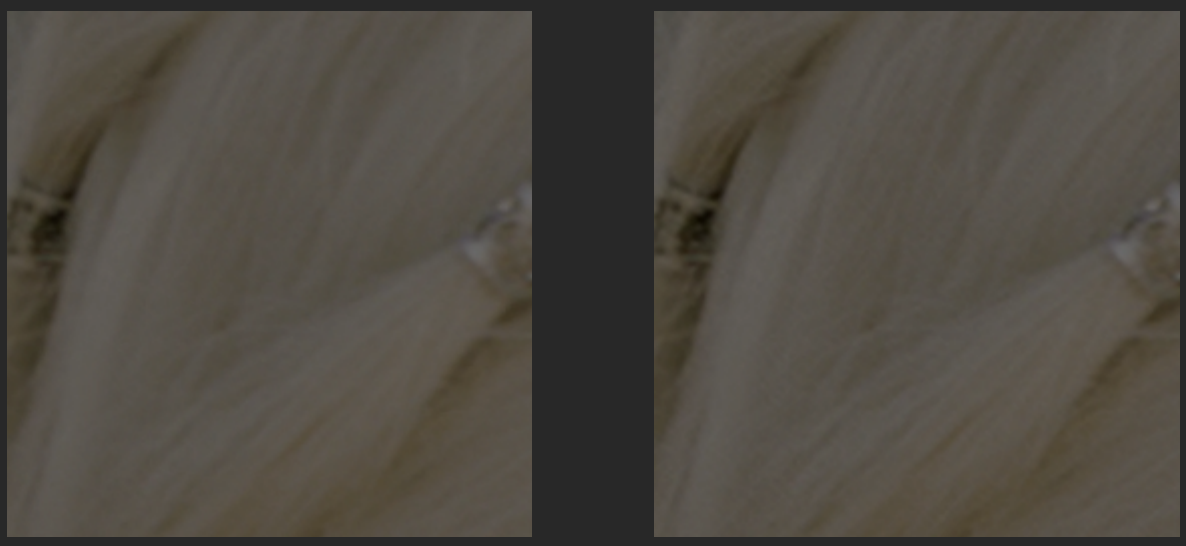

Figure 9: Source UHD Blu-ray with 1080p footage upscaled to 2160p via a bilinear filter on the left, rescale with `Debilinear` and `nnedi3` on the right.

If your video seems to have multiple source resolutions in every frame (i.e. different layers are in different resolutions), which you can notice by `getnative` outputting multiple results, your best bet is to downscale to the lowest resolution via `Spline36`. While you technically can mask each layer to descale all of them to their source resolution, then scale each one back up, this is far too much effort for it to be worth it.

3.2.7 Deringing

The term "ringing" can refer to a lot of edge artifacts, with the most common ones being mosquito noise and edge enhancement artifacts. Ringing is something very common with low quality sources. However, due to poor equipment and atrocious compression methods, even high bitrate concerts are prone to this. To fix this, it's recommended to use something like `HQDeringmod` or `EdgeCleaner` (from `scoll`), with the former being my recommendation. The rough idea behind these is to blur and sharpen edges, then merge via edgemasks. They're quite simple, so you can just read through them yourself and you should get a decent idea of what they'll do. As `rgvs.Repair` can be quite aggressive, I'd recommend playing around with the repair values if you use one of these functions and the defaults don't produce decent enough results.

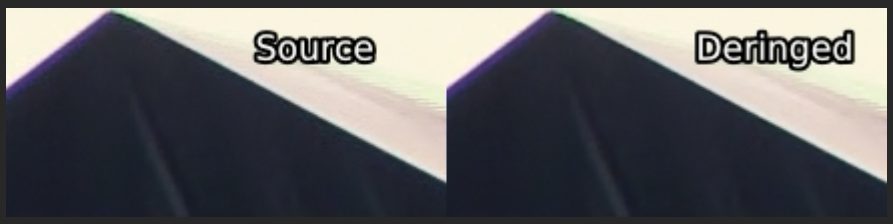

Figure 10: `HQDeringmod(mrad=5, msmooth=10, drrep=0)` on the right, source on the left. This is *very* aggressive deringing that I usually wouldn't recommend. The source image is from a One Ok Rock concert Blu-ray with 37 mbps video.

3.2.8 Dehaloing

Haloing is a lot what it sounds like: thick, bright lines around edges. These are quite common with poorly resized content. You may also find that bad descaling or descaling of bad sources can produce noticeable haloing. To fix this, you should use either `havsfunc`'s `DeHalo_alpha` or its already masked counterpart, `FineDehalo`. If using the former, you'll *have* to write your own mask, as unmasked dehaloing usually leads to awful results. For a walkthrough of how to write a simple dehalo mask, check `encode.moe`'s guide[30].

As `FineDehalo` is a wrapper around `DeHalo_alpha`, they share some parameters:

```
FineDehalo(src, rx=2.0, ry=None, thmi=80, thma=128, thlimi=50, thlima=100,
   darkstr=1.0, brightstr=1.0, showmask=0, contra=0.0, excl=True,
   edgeproc=0.0) # ry defaults to rx
DeHalo_alpha(clp, rx=2.0, ry=2.0, darkstr=1.0, brightstr=1.0, lowsens=50,
   highsens=50, ss=1.5)
```

The explanations on the AviSynth wiki are good enough: `http://avisynth.nl/index.php/DeHalo_alpha#Syntax_and_Parameters` and `http://avisynth.nl/index.php/FineDehalo#Syntax_and_Parameters`

3.2.9 Denoising

Denoising is a rather touchy subject. Live action encoders will never denoise, while anime encoders will often denoise too much. The main reason you'll want to do this for anime is that it shouldn't have any on its own, but compression introduces noise, and bit depth conversion introduces dither. The former is unwanted, while the latter is wanted. You might also encounter intentional grain in things like flashbacks. Removing unwanted noise will aid in compression and kill some slight dither/grain; this is useful for 10-bit, since smoother sources simply encode better

[30] `https://guide.encode.moe/encoding/masking-limiting-etc.html#example-build-a-simple-dehalo-mas`

and get you great results, while 8-bit is nicer with more grain to keep banding from appearing etc. However, sometimes, you might encounter cases where you'll have to denoise/degrain for reasons other than compression. For example, let's say you're encoding an anime movie in which there's a flashback scene to one of the original anime episodes. Anime movies are often 1080p productions, but most series aren't. So, you might encounter an upscale with lots of 1080p grain on it. In this case, you'll want to degrain, rescale, and merge the grain back[31]:

```
degrained = core.knlm.KNLMeansCL(src, a=1, h=1.5, d=3, s=0, channels="Y",
    device_type="gpu", device_id=0)
descaled = fvf.Debilinear(degrained, 1280, 720)
upscaled = nnedi3_rpow2(descaled, rfactor=2).resize.Spline36(1920,
    1080).std.Merge(src, [0,1])
diff = core.std.MakeDiff(src, degrained, planes=[0])
merged = core.std.MergeDiff(upscaled, diff, planes=[0])
```

3.2.10 Graining

As grain and dither are some of the hardest things to compress, many sources will feature very little of this or obviously destroyed grain. To counteract this or simply to aid with compression of areas with no grain, it's often beneficial to manually add grain. In this case of destroyed grain, you will usually want to remove the grain first before re-applying it. This is especially beneficial with anime, as a lack of grain can often make it harder for the encoder to maintain gradients.

As we're manually applying grain, we have the option to opt for static grain. This is almost never noticeable with anime, and compresses a lot better, hence it's usually the best option for animated content. It is, however, often quite noticeable in live action content, hence static grain is not often used in private tracker encodes.

The standard graining function, which the other functions also use, is `grain.Add`:

```
grained = core.grain.Add(clip, var=1, constant=False)
```

The `var` option here signifies the strength. You probably won't want to raise this too high. If you find yourself raising it too high, it'll become noticeable enough to the point where you're better off attempting to match the grain in order to keep the grain unnoticeable.

The most well-known function for adding grain is `GrainFactory3`. This function allows you to specify how `grain.Add` should be applied for three different luma levels (bright, medium, dark). It also scales the luma with `resize.Bicubic` in order to raise or lower its size, as well as sharpen it via functions `b` and `c` parameters, which are modified via the `sharpen` option. It can be quite hard to match here, as you have to modify size, sharpness, and threshold parameters. However, it can produce fantastic results, especially for live action content with more natural grain.

A more automated option is `adaptive_grain`. This works similarly to `GrainFactory3`, but instead applies variable amounts of grain to parts of the frame depending on the overall frame's luma value and specific areas' luma values. As you have less options, it's easier to use, and it works fine for anime. The dependency on the overall frame's average brightness also makes it produce very nice results.

In addition to these two functions, a combination of the two called `adptvgrnMod`[32] is available. This adds the sharpness and size specification options from `GrainFactory3` to `adaptive_grain`. As grain is only added to one (usually smaller than the frame) image in one

[31] It usually makes more sense to build an edgemask on the degrained clip and descale the source clip, but this isn't the worst thing ever, either.

[32] `https://gitlab.com/snippets/1841290`

size, this often ends up being the fastest function. If the grain size doesn't change for different luma levels, as is often the case with digitally produced grain, this can lead to better results than both of the aforementioned functions.

For those curious what this may look like, please refer to the debanding example from Mirai in figure 4, as `adptvgrnMod` was used for graining in that example.

3.2.11 Deblocking

Deblocking is mostly equivalent to smoothing the source, usually with another mask on top. The most popular function here is `Deblock_QED` from `havsfunc`. The main parameters are

- `quant1`: Strength of block edge deblocking. Default is 24. You may want to raise this value significantly.
- `quant2`: Strength of block internal deblocking. Default is 26. Again, raising this value may prove to be beneficial.

Other popular options are `deblock.Deblock`, which is quite strong, but almost always works, `dfttest.DFTTest`, which is weaker, but still quite aggressive, and `fvf.AutoDeblock`, which is quite useful for deblocking MPEG-2 sources and can be applied on the entire video. Another popular method is to simply deband, as deblocking and debanding are very similar. This is a decent option for AVC Blu-ray sources.

3.2.12 Detinting

If you've got a better source with a tint and a worse source without a tint, and you'd like to remove it, you can do so via `timecube` and DrDre's Color Matching Tool. First, add two reference screenshots to the tool, export the LUT, save it, and add it via something like:

```
clip = core.resize.Point(src, matrix_in_s="709", format=vs.RGBS)
detint = core.timecube.Cube(clip, "LUT.cube")
out = core.resize.Point(detint, matrix=1, format=vs.YUV420P16)
```

Figure 11: Source with tint on left, tint removed on right. This example is from the D-Z0N3 encode of Your Name (2016). Some anti-aliasing was also performed on this frame.

Similarly, if you have what's known as a gamma bug, or more precisely, double range compression (applying full to limited range compression to an already limited range clip), just do the following (for 16-bit):

```
out = core.std.Levels(src, gamma=0.88, min_in=4096, max_in=60160,
   min_out=4096, max_out=60160, planes=0)
```

Figure 12: Double range compression on left, gamma bug fix on right.

0.88 is usually going to be the required value, but it's not unheard of to have to apply different gamma values. This is necessary if blacks have a luma value of 218 instead of 235. Do not

perform this operation in low bit depth. The reasoning for this can be seen in figure 23. If the chroma planes are also affected, you'll have to deal with them separately:

```
out = core.std.Levels(src, gamma=0.88, min_in=4096, max_in=61440,
   min_out=4096, max_out=61440, planes=[1, 2])
```

You can also use the `fixlvls` wrapper in `awsmfunc` to do all these operations.

If you have a source with an improper color matrix, you can fix this with the following:

```
out = core.resize.Point(src, matrix_in_s='470bg', matrix_s='709')
```

The `'470bg'` is what's usually known as 601. The reason for the resizers is that matrix conversion takes place between YUV to RGB conversion, meaning we need to upscale the chroma. We use point resizing, as it's involutory. To know if you should be doing this, you'll need some reference sources, preferably not web sources. Technically, you can identify bad colors and realize that it's necessary to change the matrix.

Figure 13: Example of matrix conversion with Burning (2018). Used the TayTO encode for this example. Most notable area is her pink bra and the red in the background.

3.2.13 Dehardsubbing and Delogoing

While this issue is particularly common with anime, it does also occur in some live action sources, and many music videos or concerts on played on TV stations with logos, hence it's worth looking at how to remove hardsubs or logos. For logos, the `Delogo` plugin is well worth considering. To use it, you're going to need the `.lgd` file of the logo. You can simply look for this via your favorite search engine and something should show up. From there, it should be fairly straightforward what to do with the plugin.

The most common way of removing hardsubs is to compare two sources, one with hardsubs and one reference source with no hardsubs. The functions I'd recommend for this are `hardsubmask` and `hardsubmask_fades` from `kagefunc`[33]. The former is only useful for sources with black and white subtitles, while the latter can be used for logos as well as moving subtitles. Important parameters for both are the `expand` options, which imply `std.Maximum` calls. Depending on how good your sources are and how much gets detected, you may want to lower these values.

We can also perform something similar with `Delogo` to create a mask of sorts:

```
No example script yet.
```

Once you have your mask ready, you'll want to merge in your reference hardsub-less source with the main source. You may want to combine this process with some tinting, as not all sources will have the same colors. It's important to note that doing this will yield far better results than swapping out a good source for a bad one. If you're lazy, these masks can usually be applied to the entire clip with no problem, so you won't have to go through the entire video looking for hardsubbed areas.

[33] https://github.com/Irrational-Encoding-Wizardry/kagefunc

3.2.14 Masking

This is the most complex part, and is something that most encoders outside of anime encoders tend to ignore. Masks help protect important details from being destroyed by your filters. The most popular kind of mask that private tracker encoders use are binarize masks:

```
y = core.std.ShufflePlanes(src, 0, vs.GRAY)
mask = core.std.Binarize(y, 5000)
merge = core.std.MaskedMerge(filtered, src, mask)
```

In this case, I'm assuming we're working in 16-bit. What `std.Binarize` is doing here is making every value under 5000 the lowest and every value above 5000 the maximum value allowed by our bit depth. This means that every pixel above 5000 will be copied from the source clip. This is usually referred to as a "luma mask", and is commonly used for debanding, where often only dark areas require any work done.

We can also do this with one of the chroma planes:

```
u = core.std.ShufflePlanes(src, 1, vs.GRAY)
mask = core.std.Binarize(u, 5000)
mask = core.resize.Bilinear(mask, 1920, 1080)
mask = core.std.Inflate(mask)
merge = core.std.MaskedMerge(filtered, src, mask)
```

As you may have noticed, I've performed the same binarize, but am both resizing and inflating the mask. The reason for resizing is obviously because the chroma planes are lower resolution in `YUV420`, but the choice of a resizer might seem peculiar to some; using a bilinear resizer leads to blurring, which means that surrounding pixels will also be affected, which is often useful for anti-aliasing. I've added an additional `std.Inflate` for the same reason, although it's usually more useful for luma masks than chroma masks.

Far more interesting and useful masks are edgemasks and debanding specific masks. For edgemasks, VapourSynth users have a big advantage, as `kgf.retinex_edgemask` is incredibly accurate and leads to fantastic results. This edgemask takes the source image, uses the retinex algorithm to raise contrast and brightness in dark areas, then layers a TCanny mask of the output on top of a Kirsch mask. Two very common use cases for it are debanding and anti-aliasing

```
retinex = kgf.retinex_edgemask(src)
antialiasingmask = retinex.std.Binarize(65000).std.Inflate()
antialiasingmerge = core.std.MaskedMerge(src, antialiasing,
   antialiasingmask)

debandmask = retinex.std.Binarize(7000).std.Maximum().std.Inflate()
merge = core.std.MaskedMerge(deband, src, debandmask)
```

For debanding, you'll usually want to get as much from the source as possible so as not to destroy details, hence we're binarizing at a low value and growing the mask with `std.Maximum` as well as `std.Inflate`. We want to use this mask to add content from the source to the debanded clip. There are a lot of different ways one might want to manipulate the mask, be it multiplying everything above a threshold with a certain value (`std.Expr(retinex, "x 7000 > x 10 * x ?"`), only maximizing and inflating, leaving it simply be or what have you.

In a very different yet similar fashion, anti-aliasing usually only wants to be applied to obvious edges, hence we're binarizing at a high value. The `std.Inflate` call is important so we actually get the full result of the anti-aliasing applied. Here, we want to add the anti-aliasing to the source via our mask.

Other helpful edgemasks include:

- `std.Prewitt`
- `std.Sobel` is usually more accurate than Prewitt, although it's recommended to test both if Kirsch or Retinex-type masks aren't an option.
- `tcanny.TCanny` This is basically a Sobel mask thrown over a blurred clip.
- `kgf.kirsch` will generate almost identical results to `retinex_edgemask` in bright scenes, as it's one of its components. Slower than the others, as it uses more directions, but will get you great results.

Comparisons of all of these can be found in the appendix under figure 24 and 25.

While edgemasks are great for debanding, they'll often also detect the edges of the banding itself, and are usually quite slow. Great alternatives include `GradFun3`'s mask and `debandmask`. The latter is very fast and gets you better results, but should probably be inflated. For `GradFun3`, you can use the `Fag3kdb` wrapper by Frechdachs. I'd recommend going with `debandmask` whenever possible, but an edgemask will usually prove to produce better results in darker scenes, so do do some testing. To get a mask output out of `GradFun3`, you can do the following:

```
mask = fvf.GradFun3(src, debug=1)
```

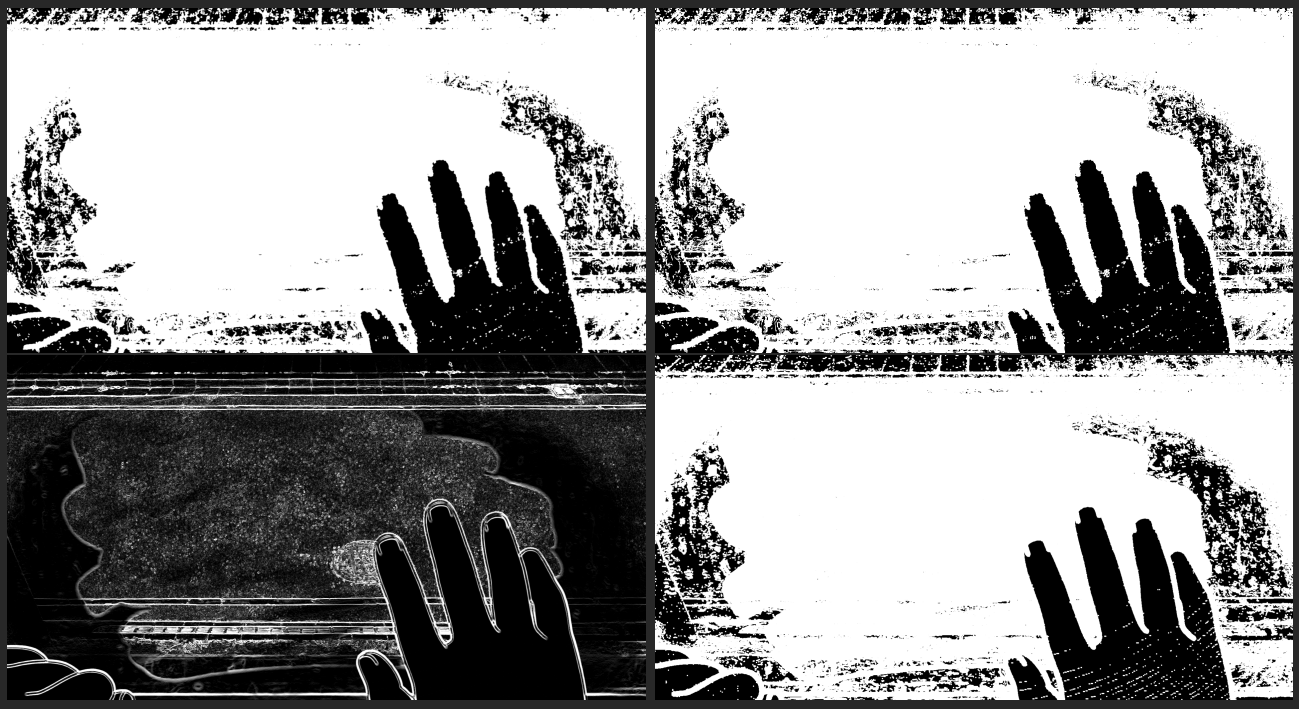

Figure 14: Comparison of `GradFun3` (top left), `debandmask` (top right), `retinex_edgemask` (bottom left), and `retinex_edgemask.std.Maximum().std.Inflate()` (bottom right).

For those curious about the difference between strong debanding with and without a mask, please refer to figure 26 in the appendix to see a comparison of the influence a simple edgemask can have on debanding. When using a luma/chroma mask, please don't forget to test whether you should be using a proper edgemask or debanding-specific mask on top of your previous mask, as a simple brightness mask won't keep edges from being destroyed! Examples of this are included in the appendix under figures 27 and 28.

Now that we've covered common masks, let's move on to the functions that will help you manipulate masks or create your own. I'll be brief, as their documentation is great:

- `std.Maximum/Minimum`: Use this to grow or shrink your mask, you may additionally want to apply `coordinates=[0, 1, 2, 3, 4, 5, 6, 7]` with whatever numbers work for you in order to specify weights of the surrounding pixels.

- `std.Inflate/Deflate`: Similar to the previous functions, but instead of applying the maximum of pixels, it merges them, which gets you a slight blur of edges. Useful at the end of most masks so you get a slight transition between masked areas.
- `std.Expr`: Known to be a very complicated function. Applies logic via reverse Polish notation. If you don't know this, read up on Wikipedia. Some cool things you can do with this are make some pixels brighter while keeping others the same (instead of making them dark as you would with `std.Binarize`): `std.Expr("x 2000 > x 10 * x ?")`. This would multiply every value above 2000 by ten and leave the others be. One nice use case is for in between values: `std.Expr("x 10000 > x 15000 < and x {} = x 0 = ?".format(2**src.format.bits_per_sample - 1))`.
  This makes every value between 10 000 and 15 000 the maximum value allowed by the bit depth and makes the rest zero, just like how a `std.Binarize` mask would. Almost every function can be or already is expressed via this.
- `std.MakeDiff` and `std.MergeDiff`: These should be self-explanatory. Use cases can be applying something to a degrained clip and then merging the clip back, as was elaborated in the Denoising section.
- `std.Convolution`: In essence, apply a matrix to your pixels. The documentation explains it well, so just read that if you don't get it. Lots of masks are defined via convolution kernels. You can use this to do a whole lot of stuff, like `std.Expr`. For example, if you want to average all the values surrounding a pixel, do `std.Convolution([1, 1, 1, 1, 0, 1, 1, 1, 1])`.
- `std.Transpose`: Transpose (i.e. flip) your clip.
- `std.Turn180`: Turns by 180 degrees.
- `std.BlankClip`: Just a frame of a solid color. You can use this to replace bad backgrounds or for cases where you've added grain to an entire movie but you don't want the end credits to be full of grain. To maintain TV range, you can use `std.BlankClip(src, color=[16, 128, 128])` for 8-bit black. Also useful for making area based masks.
- `std.Invert`: Self-explanatory. You can also just swap which clip gets merged via the mask instead of doing this.
- `std.Limiter`: You can use this to limit pixels to certain values. Useful for maintaining TV range (`std.Limiter(min=16, max=235)`).
- `std.Median`: This replaces each pixel with the median value in its neighborhood. Mostly useless.
- `std.StackHorizontal/std.StackVertical`: Stack clips on top of/next to each other.
- `std.Merge`: This lets you merge two clips with given weights. A weight of 0 will return the first clip, while 1 will return the second. The first thing you give it is a list of clips, and the second item is a list of weights for each plane. Here's how to merge chroma from the second clip into luma from the first: `std.Merge([first, second], [0, 1])`. If no third value is given, the second one is copied for the third plane.
- `std.MaskedMerge`: What it's all about; merges the second clip to the first according to the given mask.
- `std.ShufflePlanes`: Extract or merge planes from a clip. For example, you can get the luma plane with `std.ShufflePlanes(src, 0, vs.GRAY)`.

If you want to apply something to only a certain area, you can use the wrapper `rekt`[34] or `rekt_fast`. The latter only applies you function to the given area, which speeds it up and is quite useful for anti-aliasing and similar slow filters. Some wrappers around this exist already, like `rektaa` for anti-aliasing. Functions in `rekt_fast` are applied via a lambda function, so

[34] `https://gitlab.com/Ututu/rekt`

instead of
`core.f3kdb.Deband(src)`, you input
`rekt_fast(src, lambda x: core.f3kdb.Deband(x))`.

One more very special function is `std.FrameEval`. What this allows you to do is evaluate every frame of a clip and apply a frame-specific function. This is quite confusing, but there are some nice examples in VapourSynth's documentation: `http://www.vapoursynth.com/doc/functions/frameeval.html`. Now, unless you're interested in writing a function that requires this, you likely won't ever use it. However, many functions use it, including `kgf.adaptive_grain`, `awf.FrameInfo`, `fvf.AutoDeblock`, `TAAmbk`, and many more. One example I can think of to showcase this is applying a different debander depending on frame type:

```
import functools
def FrameTypeDeband(n, clip):
  if clip.get_frame(n).props._PictType.decode() == "B":
    return core.f3kdb.Deband(clip, y=64, cr=0, cb=0, grainy=64, grainc=0,
       keep_tv_range=True, dynamic_grain=False)
  elif clip.get_frame(n).props._PictType.decode() == "P":
    return core.f3kdb.Deband(clip, y=48, cr=0, cb=0, grainy=64, grainc=0,
       keep_tv_range=True, dynamic_grain=False)
  else:
    return core.f3kdb.Deband(clip, y=32, cr=0, cb=0, grainy=64, grainc=0,
       keep_tv_range=True, dynamic_grain=False)

out = core.std.FrameEval(src, functools.partial(FrameTypeDeband, clip=src))
```

If you'd like to learn more, I'd suggest reading through the Irrational Encoding Wizardry GitHub group's guide: `https://guide.encode.moe/encoding/masking-limiting-etc.html` and reading through most of your favorite Python functions for VapourSynth. Pretty much all of the good ones should use some mask or have developed their own mask for their specific use case.

3.2.15 Filter Order

Filtering in the wrong order can lead to destructive or botched filtering. Because of this, it's recommended to use the following order:

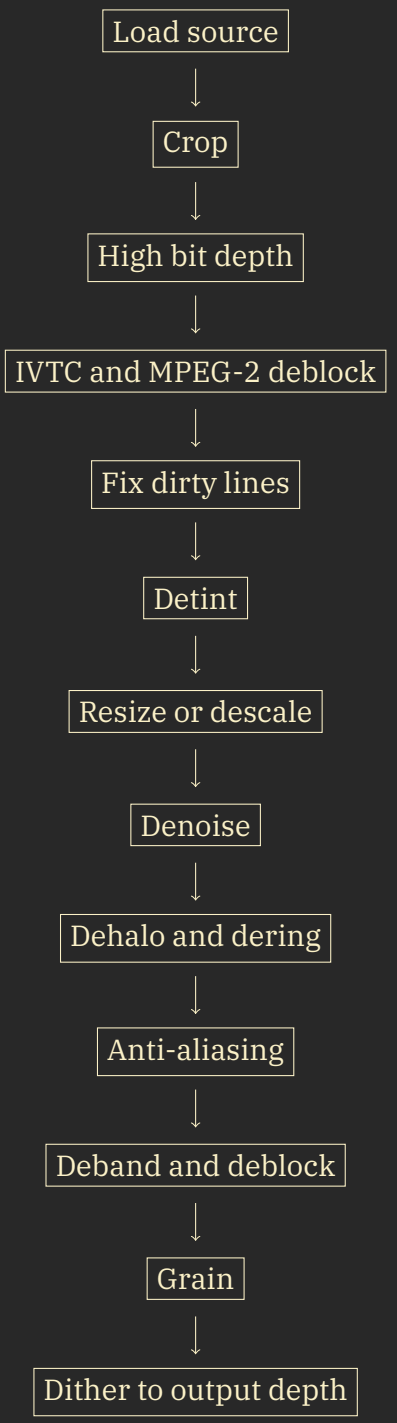

Figure 15: Recommended order in which to apply each filter. Denoising may have to be done before resizing in some cases.

3.2.16 Example Scripts

Mirai (2018):

```
import vapoursynth as vs
core = vs.get_core()
import fvsfunc as fvf
import mvsfunc as mvf
import kagefunc as kgf
import vsTAAmbk as taa
import havsfunc as haf
from debandmask import *
from rekt import *
from adptvgrnMod import *

src = core.ffms2.Source("")

# Going up to 16-bit, as I like to work in this depth.
b16 = mvf.Depth(src, 16).std.Crop(top=20, bottom=22)

# Filling the first row and filling the chroma of the first two rows.
fb1 = core.fb.FillBorders(b16, top=1)
fb2 = core.fb.FillBorders(b16, top=2)
b16 = core.std.Merge(fb1, fb2, [0,1])

# Super light denoising. No point in BM3D for denoising this weak.
b16 = core.knlm.KNLMeansCL(b16, a=3, h=0.1, d=2, device_type='gpu',
    device_id=0, channels='Y')
b16 = core.knlm.KNLMeansCL(b16, a=2, h=0.2, d=1, device_type='gpu',
    device_id=0, channels='UV')

# Soft dehalo.
b16 = haf.FineDehalo(b16, rx=2.0, thmi=80, thma=128, thlimi=25, thlima=100,
    darkstr=0.5, brightstr=0.5)

# Dirty lines during credits. Cleaning edges, then halos.
cf = core.std.Crop(b16, left=94, top=292, right=1018,
    bottom=290).fb.FillBorders(top=1, left=1,
    bottom=1).edgefixer.ContinuityFixer(left=[2, 2, 2], top=[2, 2, 2],
    right=2, bottom= [0, 2, 2], radius=15)
fb = rekt_fast(b16, lambda x: core.fb.FillBorders(x, left=2, top=2,
    right=1, bottom=2).std.Merge(cf, [1,0]), left=94, top=292, right=1018,
    bottom=290)
dh = rekt_fast(fb, lambda x: haf.FineDehalo(x, rx=2.0, thmi=80, thma=128,
    thlimi=25, thlima=100, darkstr=0.5, brightstr=2.3), left=94, top=292,
    right=1018, bottom=290)
sf = fvf.rfs(b16, dh, "[1434 2296]")

cf = core.std.Crop(b16, left=94, top=302, right=1018,
    bottom=300).fb.FillBorders(left=1).edgefixer.ContinuityFixer(left=[2,
    2, 2], top=1, right=1, bottom= [1, 2, 2], radius=5)
fb = rekt_fast(b16, lambda x: core.fb.FillBorders(x, left=2, top=1,
    right=1, bottom=2).std.Merge(cf, [1,0]), left=94, top=302, right=1018,
    bottom=300)
dh = rekt_fast(fb, lambda x: haf.FineDehalo(x, rx=2.0, thmi=80, thma=128,
    thlimi=25, thlima=100, darkstr=0.5, brightstr=1.5), left=94, top=302,
    right=1018, bottom=300)
sf = fvf.rfs(sf, dh, "[133711 135117] [135360 136057] [136143 137216]
    [137282 138288] [138377 138757] [138820 140782]")

cf = core.std.Crop(b16, left=94, top=302, right=1018,
```

```
    bottom=300).fb.FillBorders(left=1).edgefixer.ContinuityFixer(left=[2,
    2, 2], top=1, right=1, bottom= [1, 2, 2], radius=5)
fb = rekt_fast(b16, lambda x: core.fb.FillBorders(x, left=2, top=1,
    right=1, bottom=2).std.Merge(cf, [1,0]), left=94, top=302, right=1018,
    bottom=300)
dh = rekt_fast(fb, lambda x: haf.FineDehalo(x, rx=2.0, thmi=80, thma=128,
    thlimi=25, thlima=100, darkstr=0.5, brightstr=1.5).f3kdb.Deband(y=48,
    cb=0, cr=0, range=5, grainy=64, grainc=32, output_depth=16,
    keep_tv_range=True), left=94, top=302, right=1018, bottom=300)
sf = fvf.rfs(sf, dh, "[135118 135296] [138305 138376]")

mask = core.std.ShufflePlanes(b16, 0, vs.GRAY).std.Trim(2400, 2401) *
    src.num_frames
mask = rekt(mask, core.std.BlankClip(b16, 1920, 1038, format=vs.GRAY16),
    left=666, top=292, right=1114, bottom=744)
dh_lim = core.std.MaskedMerge(dh, b16, mask)
sf = fvf.rfs(sf, dh_lim, "[2297 2329]")

# 4:3 cropped scene. Replacing borders with my own black borders in order
    to keep them from having a different shade of black.
crop = core.std.Crop(b16, left=254, right=254)
fb = core.fb.FillBorders(crop, left=1, right=1).std.Merge(crop,
    [1,0]).edgefixer.ContinuityFixer(left=1, right=1, top=0, bottom=0,
    radius=50).std.AddBorders(left=254, right=254, color=[4096, 32768,
    32768])
sf = fvf.rfs(sf, fb, "[33448 34196]")

# Placebo edgemask binarized so we only get the obvious edges, then
    inflated.
mask = kgf.retinex_edgemask(b16).std.Binarize(65500).std.Maximum
    ().std.Inflate()

# Strong aliasing.
aa = taa.TAAmbk(b16, aatype=2, mtype=0, opencl=False)
aa = core.std.MaskedMerge(b16, aa, mask)
sf = fvf.ReplaceFramesSimple(sf, aa, mappings="[4225 4727] [18340 18387]
    [129780 131148]")

# Mild aliasing.
aa = taa.TAAmbk(b16, aatype=3, mtype=0, opencl=False)
aa = core.std.MaskedMerge(b16, aa, mask)
sf = fvf.ReplaceFramesSimple(sf, aa, mappings="[55394 55451] [55649 55782]
    [120840 120901]")

# Very strong aliasing.
aa = taa.TAAmbk(b16, aatype=6, mtype=0, repair=16)
aa = core.std.MaskedMerge(b16, aa, mask)
sf = fvf.ReplaceFramesSimple(sf, aa, mappings="[107405 107462]")

# Strong aliasing that I tried to fix with a terrible mask.
mask =
    kgf.retinex_edgemask(b16).std.Binarize(65500).std.Maximum().std.Minimum(
   coordinates=[1,0,1,0,0,1,0,1]).std.Deflate().std.Deflate()
aa = taa.TAAmbk(b16, aatype=6, mtype=0, opencl=False)
aa = core.std.MaskedMerge(b16, aa, mask)
sf = fvf.ReplaceFramesSimple(sf, aa, mappings="[55510 55580]")

# I simply marked this, it would require a lot of work, so I just decided
    against doing this.
#sf = fvf.rfs(sf, ?, "[65880 66478]")
#sf = fvf.rfs(sf, ?, "[120902 121051] [121790 121905] [122388 122528]
```

```
    [123038 123153] [126686 126812] [128740 128953]") #Banding? [121063
    121095] [121906 121968] [122530 122576]

# Graining an area with no grain.
gr = adptvgrnMod(b16, strength=2.5, size=1.25, sharp=35, static=False,
    luma_scaling=3, grain_chroma=False)
sf = fvf.rfs(sf, gr, "[120840 120901]")

# Debanding with the standard debandmask. All of these debanding areas had
    almost no grain, so I added some on top.
dbmask = debandmask(b16, lo=6144, hi=12288, lothr=320, hithr=384, mrad=2)
deband = core.f3kdb.Deband(b16, y=34, cb=0, cr=0, range=10, grainy=16,
    grainc=8, output_depth=16, keep_tv_range=True)
merge = core.std.MaskedMerge(deband, b16, dbmask)
merge = adptvgrnMod(merge, strength=2, size=1.5, sharp=25, static=False,
    luma_scaling=5, grain_chroma=True)
sf = fvf.rfs(sf, merge, "[3174 3254] [3540 3655] [7463 7749] [41056 41597]
    [63482 64106] [91033 91164]")

# Debanding with retinex.
mask = kgf.retinex_edgemask(b16).std.Maximum().std.Inflate().std.Maximum
    ().std.Inflate()
deband = core.f3kdb.Deband(b16, y=48, cb=48, cr=48, range=15, grainy=16,
    grainc=16, output_depth=16, keep_tv_range=True)
merge = core.std.MaskedMerge(deband, b16, mask)
merge = adptvgrnMod(merge, strength=2.2, size=1.25, sharp=15, static=False,
    luma_scaling=5, grain_chroma=True)
sf = fvf.rfs(sf, merge, "[77952 78034] [93358 93443]")

# Debanding with gradfun3 mask.
deband = Fag3kdb(b16, thry=54, thrc=54, radiusy=10, radiusc=6, grainy=32,
    grainc=16)
sf = fvf.rfs(sf, deband, "[25 263]")

# Dithering back to 8-bit.
final = mvf.Depth(sf, 8, dither=7)

# Replacing black areas with a simple black screen in order to keep slight
    variations from happening. Usually not necessary, though.
blank = core.std.BlankClip(src.std.Crop(top=20, bottom=22), 1920, 1038,
    color=[16, 128, 128])
final = fvf.rfs(final, blank, "[0 24] [1352 1433] [58945 59016] [75563
    75633] [78351 78421] [81130 81141] [81261 81272] [93967 94062] [99889
    99959] [118093 118147] [140928 140951]")

final.set_output()
```

Sword Art Online: The Movie - Ordinal Scale (2017):
Sword.Art.Online.The.Movie.Ordinal.Scale.2017.ITA.1080p.BluRay.AC3.x264.D-Z0N3

```
import vapoursynth as vs
core = vs.get_core()
import fvsfunc as fvf
import kagefunc as kgf
import havsfunc as hvf
import vsTAAmbk as taa
import fag3kdb
import nnedi3_rpow2 as nnrp
```

```
src = core.ffms2.Source("")
resize = src # I called this and was too lazy to change it.

# Rescaling a flashback with grain.
dn = core.knlm.KNLMeansCL(src, d=3, a=1, s=0, h=1.5, device_type="gpu",
    device_id=1, channels="Y")
diff = core.std.MakeDiff(src, dn, planes=[0])
ds = fvf.Debicubic(dn, 1280, 720)
us = nnrp.nnedi3_rpow2(ds, 2, 1920, 1080, kernel="Spline36")
merged = core.std.MergeDiff(us, diff, planes=[0])
src = fvf.ReplaceFramesSimple(resize, merged, mappings="[3418 3507] [3508
    5145] [75916 76205] [76253 76323] [77720 77790]")

# Rescaling a flashback without grain.
ds = fvf.DescaleAA(dn, 1280, 720).std.MergeDiff(diff, planes=[0])
src = fvf.ReplaceFramesSimple(src, ds, mappings="[3298 3417]")

# Going to 16-bit. The above parts are in 8-bit because I was scared of
    performance issues.
src = fvf.Depth(src, 16)

# I like to establish a separate variable for 16-bit and leave src for
    8-bit, but didn't do that here. This is so I could copy-paste commands.
b16 = src

# Anti-aliasing. As you might be able to tell, the crop and stacking could
    now be replaced by rekt_fast or simply rektaa.
aa = core.std.Crop(b16, left=400, right=1006)
aa = taa.TAAmbk(aa, aatype=-3, preaa=-1, strength=0, mtype=2, opencl=True)
left = core.std.Crop(b16, right=1920 - 400)
right = core.std.Crop(b16, left=1920 - 1006)
aa = core.std.StackHorizontal([left, aa, right]).std.Crop(top=208,
    bottom=456)
top = core.std.Crop(b16, bottom=1080 - 208)
bottom = core.std.Crop(b16, top=1080 - 456)
aa = core.std.StackVertical([top, aa, bottom])
sfaa = fvf.ReplaceFramesSimple(b16, aa, mappings="[42583 42813] [58812
    59050] [65211 65281] [92132 92274]")

# Debanding with a standard ass mask.
db = b16.f3kdb.Deband( range=15, y=60, cb=60, cr=60, grainy=22, grainc=22,
    output_depth=16)
mask = kgf.retinex_edgemask(b16).std.Inflate()
merged = core.std.MaskedMerge(db, b16, mask)
sfdb = fvf.ReplaceFramesSimple(sfaa, merged, mappings="[3508 3603] [17600
    17706] [41865 42113] [76922 77488] [78444 78598] [81054 81280] [150853
    150933] [152057 152288] [152324 152424] [152443 152508] [152521 152686]
    [171669 172433] [172561 172643] [170283 170557]")

# Debanding values that were outside of the range of 10000-25000.
db = b16.f3kdb.Deband(range=10, y=160, cb=0, cr=0, grainy=28, grainc=0,
    output_depth=16)
mask = core.std.ShufflePlanes(b16, 0, vs.GRAY).std.Expr("x 10000 < x 25000
    > or x 10 * x 10 / ?")
merged = core.std.MaskedMerge(db, b16, mask)
sfdb = fvf.ReplaceFramesSimple(sfdb, merged, mappings=" [96133 96273]")

# Fixing dirty lines during credits. Again, rekt_fast would've been useful
    back then.
bot = core.std.Crop(sfdb, top=1080 - 330)
middle = core.std.Crop(sfdb, top=318,
```

```
    bottom=330).edgefixer.ContinuityFixer(top=1, bottom=1, left=0, right=0,
    radius=5)
fb = core.fb.FillBorders(middle, top=2, bottom=2)
middle = core.std.Merge(fb, middle, [1, 0])
top = core.std.Crop(sfdb, bottom=1080 - 318)
merge = core.std.StackVertical([top, middle, bot])
right = core.std.Crop(merge, left=1920 - 134)
middle = core.std.Crop(merge, left=1018,
    right=134).edgefixer.ContinuityFixer(left=2, right=2, top=0, bottom=0,
    radius=5)
fb = core.fb.FillBorders(middle, left=2, right=2)
middle = core.std.Merge(fb, middle, [1, 0])
left = core.std.Crop(merge, right=1920 - 1018)
merge = core.std.StackHorizontal([left, middle, right])
sfc = fvf.ReplaceFramesSimple(sfdb, merge, mappings="[165067 167168]
    [167403 169466] [169842 170557] [170558 171041]")

# Dithering the result back to 8-bit.
final = fvf.Depth(sfc, 8)

final.set_output()
```

BTS - Blood, Sweat and Tears

`BTS – Blood, Sweat & Tears 2016 1080p ProRes FLAC 2.0 AVC x264 10-bit - A.R.M.Y`

A Quick explanation of `bandmask`[35]:

Edge masks work by finding areas where gradients are large, so the most simple convolution is

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix},$$

meaning we're checking the difference between the pixel to the left and the pixel to the right. We can move the subtraction to identify areas with no grain by checking the difference with the original pixel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This isn't exactly what's happening here, though; we're performing the subtraction after multiple iterations of the convolution, so we get the gradient over a larger number of pixels. We then binarize this, minimize to remove noise, then maximize again.

```
import vapoursynth as vs
core = vs.get_core()
import fvsfunc as fvf
import kagefunc as kgf
import havsfunc as hvf
from adptvgrnMod import *
from rekt import *
from bandmask import *


# Load, go to high bit depth, and crop.
src = core.ffms2.Source("Blood, Sweat & Tears 피땀눈물_BTS 방탄
    소년단-187617728.mov")
hbd = fvf.Depth(src, 16)
crp = hbd.std.Crop(top=134, bottom=134)
```

[35] https://gitlab.com/snippets/1904934

```
# One scene has four lines missing at the bottom and dirty lines elsewhere.
# Every plane was fixed individually.
acr = crp.std.Crop(bottom=4)
ycr = acr.std.ShufflePlanes(0, vs.GRAY)
ucr = acr.std.ShufflePlanes(1, vs.GRAY)
vcr = acr.std.ShufflePlanes(2, vs.GRAY)

ufx = rektlvls(ucr, [ucr.height - 2], [-6.5],
    prot_val=0).fb.FillBorders(top=1,
    mode="fillmargins").cf.ContinuityFixer(bottom=1, radius=3)
vfx = vcr.cf.ContinuityFixer(bottom=3, radius=5).fb.FillBorders(bottom=2,
    top=1, mode="fillmargins")
lvl = rektlvls(ycr, [ycr.height - 2, ycr.height - 3], [-30, -6],
    prot_val=10)
fmg = lvl.cf.ContinuityFixer(bottom=2, radius=3).fb.FillBorders(bottom=1,
    top=1, mode="fillmargins")

# Merge fixes and change subsampling to 4:2:0, then add borders.
acr = core.std.ShufflePlanes([fmg, ufx, vfx], [0, 0, 0], vs.YUV)
rsz = acr.resize.Spline36(format=vs.YUV420P16)
adb = rsz.std.AddBorders(bottom=4)

# Rest of the video only has one line missing top and bottom. Fixed and
    subsampling changed.
fmg = crp.fb.FillBorders(top=1, bottom=1, mode="fillmargins")
rsz = fmg.resize.Spline36(format=vs.YUV420P16)

# Spliced the fix in with the rest.
trm = rsz.std.Trim(0, 5232) + adb.std.Trim(5233, 5260) + rsz.std.Trim(5261)

# Luma debanding during a couple scenes with graining.
dbn = trm.f3kdb.Deband(y=64, cr=0, cb=0, range=6, grainy=0, grainc=0,
    output_depth=16)
msk = kgf.retinex_edgemask(trm).std.Binarize(11000).std.Maximum
    ().std.Inflate()
mrg = dbn.std.MaskedMerge(trm, msk)
grn = adptvgrnMod(mrg, size=1.4, sharp=90, luma_scaling=4, grainer=lambda
    x: core.grain.Add(x, var=1.6, uvar=1.0, constant=False))
snf = fvf.rfs(trm, grn, "[1612 1616] [7779 7794]")

# Some blocking was fixed with bandmask, since no edges were present.
bmk = bandmask(trm, 200).std.Crop(top=500).std.AddBorders(top=500)
grn = adptvgrnMod(dbn, size=1.3, sharp=90, luma_scaling=4, grainer=lambda
    x: core.grain.Add(x, var=.4, uvar=.3, constant=False))
mrg = trm.std.MaskedMerge(grn, bmk)
snf = fvf.rfs(snf, mrg, "[7964 7980] [8009 8018]")

# One scene has missing grain, so used bandmask and adptvgrnMod to fix this.
# I got frustrated and started copy pasting random Maximize/Inflate calls.
bmk = bandmask(trm, thr=300)
lmk = trm.std.ShufflePlanes(0, vs.GRAY).std.Binarize(55000).std.Maximum
    ().std.Maximum().std.Maximum().std.Maximum().std.Maximum()
gmk = core.std.Expr([bmk, lmk], "x y -").std.Crop(bottom=225,
    top=100).std.AddBorders(bottom=225, top=100).std.Maximum().std.Inflate
    ().std.Maximum().std.Inflate().std.Maximum().std.Maximum().std.Maximum
    ().std.Maximum().std.Maximum().std.Maximum().std.Inflate().std.Inflate
    ().std.Inflate().std.Inflate().std.Inflate().std.Inflate().std.Inflate()
gmk = kgf.iterate(gmk, core.std.Maximum, 3)
grn = adptvgrnMod(trm, size=1.2, sharp=80, luma_scaling=1, grainer=lambda
    x: core.grain.Add(x, var=1.4, uvar=1.0, constant=False))
mrg = core.std.MaskedMerge(trm, grn, gmk)
```

```
snf = fvf.rfs(snf, mrg, "[7579 7778]")

# Dehalo on one scene.
fdh = hvf.FineDehalo(trm)
grn = adptvgrnMod(fdh, size=1.2, sharp=80, luma_scaling=8, grainer=lambda
    x: core.grain.Add(x, var=.8, uvar=.5, constant=False))
snf = fvf.rfs(snf, grn, "[6523 6591]")

# Dither to output depth.
out = fvf.Depth(snf, 10)

out.set_output()
```

Other fun scripts to read through:

- `https://gist.github.com/blaze077/9025d2c1a9a59d63f0168e5fd6f9cd31` Kaiji episode 1 by blaze077
- `https://pastebin.com/q469qUcU` How to IVTC by eXmendiC
- `https://github.com/Beatrice-Raws/encode-scripts` Various scripts by Beatrice-Raws
- `https://github.com/LightArrowsEXE/Encoding-Projects` Various scripts by LightArrowsEXE
- `https://git.kageru.moe/kageru/vs-scripts/src/branch/master/abyss1.py` Made in Abyss episode 1 by kageru
- `https://pastebin.com/JB8aEGgf` Something by Nginx from TnP
- `https://github.com/Ichunjo/encode-scripts` - Various scripts by Vardë
- `https://git.concertos.live/OpusGang/EncodeScripts` - Various crowdsourced scripts
- Please send me more of these. It's super hard to find decent scripts, especially live action stuff.

3.2.17 Forum and Blog Posts

Public:

- Descaling: `https://guide.encode.moe/encoding/descaling.html`
- Masking: `https://guide.encode.moe/encoding/masking-limiting-etc.html`
- Adaptive grain: `https://blog.kageru.moe/legacy/adaptivegrain.html`
- Kirsch and retinex edge masks: `https://blog.kageru.moe/legacy/edgemasks.html`
- Denoisers: `https://blog.kageru.moe/legacy/grain.html`

Private:

- Explanation of bandmask: `https://i.fiery.me/Atwjs.png`
- Explanation of adptvgrnMod's graining params: `https://i.fiery.me/bELRN.png`
- Automating debanding (without `banddtct`): `https://awesome-hd.me/forums.php?action=viewthread&threadid=27426&post=4#post201503`[36]

[36] `https://privatebin.at/?c7bb6074f0b40a6f#98AThVWCCTjzVwGieifezXweCZ2EEKzKx2MU3gZJ87mu`

- Spotting double range compression: https://passthepopcorn.me/forums.php?action=viewthread&threadid=9197&postid=1622921#post1622921[37]
- A take on when to use which method to fix dirty lines (Fixer is ContinuityFixer): https://passthepopcorn.me/forums.php?action=viewthread&threadid=35149&postid=1656050#post1656050[38]
- Using banddtct and dirtdtct: https://git.concertos.live/AHD/awsmfunc/issues/13#issuecomment-62

3.3 x264 and x265

The two main video encoders we recommend are x264 and x265. While x265 should technically be superior, as it's the newer encoder, they both do have their upsides and downsides.

Being the more mature codec, x264 is more consistent, meaning you're less likely to be left with random scenes inexplicably looking awful. It also has better hardware support and is far faster to encode. The difference in encode speeds is usually at least a factor of 5.

On the flipside, x265 has the advantage that it is capable of storing HDR information and is substantially more efficient, especially simple stuff like anime at low bitrates.

3.3.1 8-bit and 10-bit Explained

The two bit depths we recommend considering for x264 are 8-bit and 10-bit. If you're using x265, just always use 10-bit, as it's better in every way. With x264, 8-bit has far more hardware compatibility (10-bit has practically none) and is significantly faster. However, 10-bit x264 is usually capable of saving quite a bit of space while remaining faster than x265. Additionally, 10-bit encoding is far better at preserving gradients (i.e. you won't have to be as afraid of banding reoccurring), which makes it very popular for encoding grain-free content such as anime.

3.3.2 x264 Settings

From the AHD guide (hardware compatibility stuff is in gray, a few items changed):
General settings:

- `--level 4.1` for DXVA.
- `--b-adapt 2` uses the best algorithm (that x264 has) to decide how B frames are placed.
- `--min-keyint` should typically be the frame rate of your video, e.g. if you were encoding 23.976 fps content, then you use 24. This is setting the minimum distance between I-frames.
- `--vbv-bufsize 78125 --vbv-maxrate 62500` for DXVA (The old guide used lower values to account for the possibility of writing the encode to BD for playback, this is no longer a consideration as other settings break this compatibility. The new values are the max level 4.1 can do, if your device breaks because of this the encode is not at fault, your device doesn't meet DXVA spec).
- `--rc-lookahead 250` if using mbtree, 60 or higher else. This sets how many frames ahead x264 can look, which is critical for mbtree. You need lots of memory for this. (Personally I just leave this at 250 now as the impact on memory usage is 2 GB or so.) Definitely lower this if you're encoding without mbtree and have a lot of ReplaceFramesSimple calls in your script.

[37] https://privatebin.at/?5f86bab33960876c#FXTAkApNbzQabAugep4jDYzp6nK4qYDthJaZgG2acVrQ
[38] https://privatebin.at/?aebe974a92324a02#4QCrc8jhbSNojzr5gYZVsjG3rwPX8DVjdNk3T6eENySw

- `--me umh` is the lowest you should go. If your CPU is fast enough, you might endure the slowdown from tesa. esa takes as much time as tesa without any benefit, so if you want to slow down your encode to try and catch more movement vectors, just use tesa, although the increase is not necessarily always worth it. This isn't really a setting you need to test, but on tough sources, you might squeeze some more performance out of x264 if you use tesa.
- `--direct auto` will automatically choose the prediction mode (spatial/temporal)
- `--subme 10` or `11` (personally I just set this to 11 the difference in encode speed is within 3-4%)
- `--trellis 2`
- `--no-dct-decimate` dct-decimate is a speed up that sacrifices quality. Just leave it off, since your computers can likely handle it.
- `--no-fast-pskip` Similar to the above.
- `--preset veryslow` or `placebo`, although the stuff we're changing will make veryslow be placebo, anyway.

Source-specific settings:

- `--bitrate` / `--crf` Bitrate is in Kbps (Kilobits per second) and CRF takes a float, where lower is better quality. This is the most important setting you have; bitstarve an encode and it is guaranteed to look like crap. Use too many bits and you've got bloat (and if people wanted to download massive files, they would get a remux). Of course, the bitrate need can vary drastically depending on the source.
- `--deblock -3:-3` to `1:1`. For live action, most people just stick with -3:-3. For anime, values between -3:-2 and 0:0 are common, with offsets of 1 between the two values being the norm (common values are -1:0, 0:-1, and -2:-1).
- `--qcomp 0.6` (default) to `0.8` might prove useful. Don't set this too high or too low, or the overall quality of your encode will suffer. This setting has a heavy effect on mbtree. A higher qcomp value will make mbtree weaker, hence something around 0.80 is usually optimal for anime. `--qcomp 0` will cause a constant bitrate, while `--qcomp 1` will cause a constant quantizer.
- `--aq-mode 1` to 3: 1 distributes bits on a per frame basis, 2 tends to allocate more bits to the foreground and can distribute bits in a small range of frames, 3 is a modified version of 2 that attempts to allocate more bits to dark parts of the frames. The only way to know what's best for sure it to test. Almost every source ends up looking best with aq-mode 3, however.
- `--aq-strength 0.5` to `1.3` are worth trying. Higher values might help with blocking. Lower values will tend to allocate more bits to the foreground, similar to aq-mode 2. This setting drastically affects the bitrate when encoding with CRF, so it might be worthwhile to test this setting with 2 pass if you intend on using CRF for the final encode.
- `--merange 24` (the lowest that should ever be used) to `64`, setting this too high can hurt (more than 128), 32 or 48 will be fine for most encodes. In general, 32-48 for 1080p and 32 for 720p (when using umh) for movies with lots of motion this can help (e.g. action movies). Talking heads can get away with low values like 24. The impact on encode speed is noticeable but not horrible. I prefer to use 48 for 1080p and 32 for 720p when using umh or 32 for 1080p and 32 for 720p when using tesa.
- `--no-mbtree` I highly recommend testing with both mbtree enabled and disabled, as generally it will result in two very different encodes. mbtree basically attempts to degrade the quality of blocks rather than frames, so that only the unimportant parts of a frame get less bits. To do this, it needs to know how often a block is referenced later, which is why `--rc-lookahead` should be set to 250. Useful for things with static backgrounds,

like anime. Or for things where you've used a high qcomp (.75 or above) and mbtree will have a lowered impact. When testing whether this is a decent option, you'll likely have to retest every setting, especially qcomp, psy-rd, and ipratio.

- `--ipratio 1.15` to `1.40`, with 1.30 usually being the go-to option. This is the bitrate allocation ratio between I and P frames.
- `--pbratio 1.05` to `1.30`, with 1.20 being the usual go-to. This is the bitrate allocation ratio between P and B frames. This value should always be around 0.10 lower than –ipratio, so lower it while testing ipratio. If you're using mbtree, this setting will have no affect, as mbtree determines it itself.
- `--psy-rd 0.40:0 to 1.15:0`: 0.95:0 to 1.15:0 for live action. The first number is psy-rd strength, second is psy-trellis strength. This tries to keep x264 from making things blurry and instead keep the complexity. For anime, between 0.40 and 1.00:0.00 is the usual range. Psy-trellis usually introduces a lot of ringing, but can help with maintaining dither. You can try values between 0.00 and 0.15 for live action and try values up to 0.50 for anime, although you'll usually get better results if you raise your aq-strength instead.
- `--bframes 6` to `16`, This is setting the maximum amount of consecutive P frames that can be replaced with B frames. Test with 16 for your first test run, and set according to the x264 log:

  ```
  x264 [info]: consecutive B-frames: 1.0% 0.0% 0.0% 0.0% 14.9% 23.8%
  13.9% 15.8% 8.9% 9.9% 0.0% 11.9% 0.0% 0.0% 0.0% 0.0% 0.0%
  ```

  Start counting with the first percentage as 0 and choose the highest number with more than 1%, which is 11 in this example. (Or just leave this at 16 as allowing more bframes will not harm your encode and will aid in compression, the impact on speed isn't that enormous.)
- –ref set the number of previous frames each P frame can use as references. If you don't care about hardware compatibility (and/or are doing a 10-bit encode), set this to 16. The impact on performance is quite large, but it's worth it most of the time. Calculate the number you can use or if you're using the provided CLI inputs at the end of this it will be calculated for you by x264. Always use the max that you can.
  The max `--ref` value can be calculated as follows:
  For `--level 4.1`, according to the H.264 standard, the max DPB (Decoded Picture Buffer) size is 12,288 kilobytes.
  Since each frame is stored in YV12 format, or 1.5 bytes per pixel, a 1920x1088 frame is $1920 \times 1088 \times 1.5 = 3133440\,\text{bytes} = 3060$ kilobytes.
  $12,288 \div 3060\,\text{kilobytes} = 4.01568627$, so you can use a maximum of 4 reference frames.
  Remember, round both dimensions up to a mod16 value when doing the math, even if you're not encoding mod16!
  Let's do the math for 1920x800.
  $1920 \times 800 \times 1.5 = 2304000\,\text{bytes} = 2250$ kilobytes. $12,288 \div 2250\,\text{kilobytes} = 5.45777778$, so you can use a maximum of 5 reference frames Note that these conversions use base 2, so 1 Kilobyte == 1024 bytes. If you get the math wrong, that's okay too - x264 will display a warning if you use too many, so you'll know if you need to change it.
- `--zones` is quite useful for debanding and blocking, as these areas require a larger bitrate to maintain transparency. The syntax is
  `--zones 0,100,crf=10/101,200,crf=15` or
  `--zones 0,100,b=5/101,200,b=10` with `b` being a bitrate multiplier in this case. You can also use this for areas that don't get enough bits allocated. Especially common areas are darker scenes or scenes with lots of reds. Fades can also suffer from bitstarving and require zoning. One can also lower the bitrate during credits to save that little something.
- `--output-depth 8` or `10` depending on what you're encoding in.
- `--output-csp i444` if you're encoding 4:4:4, else leave this out.

3.3.3 x265 Settings

The documentation here is very good, so I'll just go over recommended values:
Source-independent settings:

- `--preset veryslow` or `slower`
- `--no-rect` for slower computers. There's a slight chance it'll prove useful, but it probably isn't worth it.
- `--no-amp` is similar to `rect`, although it seems to be slightly more useful.
- `--no-open-gop`
- `--no-cutree` since this seems to be a poor implementation of `mbtree`.
- `--no-rskip` `rskip` is a speed up that gives up some quality, so it's worth considering with bad CPUs.
- `--no-sao` because `sao` is one of the stupidest things in x265.
- `--ctu 64`
- `--min-cu-size 8`
- `--rdoq-level 2`
- `--max-merge 5`
- `--rc-lookahead 60` although it's irrelevant as long as it's larger than min-keyint
- `--ref 6` for good CPUs, something like `4` for worse ones.
- `--bframes 16` or whatever your final bframes log output says.
- `--rd 3` or `4` (they're currently the same)
- `--subme 5`. You can also change this to `7`, but this is known to sharpen.
- `--merange 57` just don't go below `32` and you should be fine.
- `--high-tier`
- `--range limited`
- `--aud`
- `--repeat-headers`

Source-dependent settings:

- `--colorprim 9` for HDR, `1` for SDR.
- `--colormatrix 9` for HDR, `1` for SDR.
- `--transfer 16` for HDR, `1` for SDR.
- `--hdr10` for HDR.
- `--hdr10-opt` for 4:2:0 HDR, `--no-hdr10-opt` for 4:4:4 HDR and SDR.
- `--master-display "G(8500,39850)B(6550,2300)R(35400,14600)WP(15635,16450)L(10000000,20)"` with the values for L coming from your source's MediaInfo output.
- `--max-cll "711,617"` again from your source's MediaInfo.
- `--cbqpoffs` and `--crqpoffs` should usually be between -3 and 0. This sets an offset between the bitrate applied to the luma and the chroma planes.
- `--qcomp` between `0.60` and `0.80`.
- `--aq-mode` 4, 3, 2, 1, or `--hevc-aq` with `4` and `3` usually being the two best options.

- `--aq-strength` between `0.80` and `1.50`.
- `--deblock -4:-4` to `0:0`, as with x264. You can default `-3:-3` with live action.
- `--ipratio` and `--pbratio` same as x264 again.
- `--psy-rd 0.50` to `2.00`, similar-ish to x264.
- `--psy-rdoq` anything from `0.00` to `2.00` usually.
- `--no-strong-intra-smoothing` on sharp/grainy content, you can leave this on for blurry content, as it's an additional blur that'll help prevent banding.
- `--output-depth 10` for 10-bit output.

Experimental settings:

- `--scenecut-aware-qp`
- `--scenecut-window 550`
- `--max-qp-delta 2`
- `--hist-scenecut`
- `--hist-threshold 0.02`

3.4 Testing Settings

To start off, you'll want to select a smaller region of your video file to use as reference, since testing on the entire thing would take forever. The recommended way of doing this is by using `awsmfunc`'s `SelectRangeEvery`:

```
import awsmfunc as awf
out = awf.SelectRangeEvery(clip, every=15000, length=250, offset=[1000,
    5000])
```

Here, the first number is the offset between sections, the second one is the length of each section, and the offset array is the offset from start and end.

You'll want to use a decently long clip (a couple thousand frames usually) that includes both dark, bright, static, and action scenes, however, these should be roughly as equally distributed as they are in the entire video.

When testing settings, you should always use 2-pass encoding, as many settings will substantially change the bitrate CRF gets you. For the final encode, both are fine, although CRF is faster.

To find out what setting is best, compare them all to each other and the source. You can do so by interleaving them either individually or by a folder via `awsmfunc`. You'll usually also want to label them, so you actually know which clip you're looking at:

```
# Load the files before this
src = awf.FrameInfo(src, "Source")
test1 = awf.FrameInfo(test1, "Test 1")
test2 = awf.FrameInfo(test2, "Test 2")
out = core.std.Interleave([src, test1, test2])

# You can also place them all in the same folder and do
src = awf.FrameInfo(src, "Source")
folder = "/path/to/settings_folder"
out = awf.InterleaveDir(src, folder, PrintInfo=True, first=extract,
    repeat=True
```

If you're using yuuno, you can use the following iPython magic to get the preview to switch between two source by hovering over the preview screen:

```
%vspreview --diff
clip_A = core.ffms2.Source("settings/crf/17.0")
clip_A.set_output()
clip_B = core.ffms2.Source("settings/crf/17.5")
clip_B.set_output(1)
```

Usually, you'll want to test for the bitrate first. Just encode at a couple different CRFs and compare them to the source to find the highest CRF value that is indistinguishable from the source. Now, round the value, preferably down, and switch to 2-pass. For standard testing, test qcomp (intervals of 0.05), aq-modes, aq-strength (intervals of 0.05), merange (32, 48, and 64), psy-rd (intervals of 0.05), ipratio/pbratio (intervals of 0.05 with distance of 0.10 maintained), and then deblock (intervals of 1). If you think mbtree could help (i.e. you're encoding animation), redo this process with mbtree turned on. You probably won't want to change the order much, but it's certainly possible to do so.

For x265, the order should be qcomp, aq-mode, aq-strength, psy-rd, psy-rdoq, ipratio and pbratio, and then deblock.

If you want that little extra efficiency, you can redo the tests again with smaller intervals surrounding the areas around which value you ended up deciding on for each setting. It's recommended to do this after you've already done one test of each setting, as they do all have a slight effect on each other.

Once you're done testing the settings with 2-pass, switch back to CRF and repeat the process of finding the highest transparent CRF value.

4 Audio

4.1 SoX

If your source audio is hi-res (24-bit or >48 kHz), you may want to consider changing this in order to save space, as hi-res audio is very much placebo for consumers if proper conversion is done. In order to do this, we recommend SoX. If you want to convert to 16-bit 48 kHz, you can use the following input:

```
$ sox foo.wav -G -b 16 bar.wav -v -L 48000 dither -s -f gesemann
```

Use this if your source audio's sample rate is a multiple of 48 kHz (e.g. 96 kHz). For multiples of 44.1 kHz, you can simple switch out the number 48000 with 44100.

We recommend doing these conversions for encodes, as higher sample rates are usually more destructive than anything else, and the difference between 24-bit and properly done 16-bit is so small that it'd only be noticeable at extremely high volumes.

4.2 Lossy Codecs

4.2.1 Opus

Opus, while the most efficient codec and the preferred lossy codec on Concertos, has almost no hardware compatibility. This is not an issue for anyone using an HTPC, a desktop, a laptop, or even a phone, but users trying to use their Blu-ray players for playback won't be able to play it back. As we don't care as much about hardware compatibility, this is likely the easiest way to go.

Apart from being the most efficient codec, it's also fully open source, just like FLAC, and works perfectly on every operating system.

The biggest issue with Opus is that it doesn't work with matroska's ordered chapters; however, unless you're encoding something with a lot of the same intros/outros repeated, this shouldn't matter.

In order to encode opus, you need to have `opus-tools` installed, from there on it's simply

```
$ opusenc foo.wav bar.opus
```

Additionally, you can specify the `--bitrate` if you don't want to use the (admittedly low) defaults, although Opus automatically detects what should be the optimal target bitrate for it.

4.2.2 qAAC and fdkaac

As a close second to Opus in terms of efficiency, qAAC is also a viable option. Unlike Opus, AAC has a lot more hardware compatibility, especially in stereo. It also has no issues with ordered chapters. However, qAAC requires iTunes and, under Unix, wine. In order to encode the highest quality qAAC, your input would be:

```
qaac.exe -V 127 foo.wav bar.m4a
```

The `-V` value specifies quality, with higher numbers being better at `127` being the maximum.

4.2.3 Dolby Digital aka. Audio Codec 3

The king of hardware compatibility is AC-3. As it was the go-to codec for DVDs, it has very good support on most playback devices. However, its efficiency is worse than MP3. If you want to encode this at hoome for hardware compatibility (e.g. you want to encode something for AHD/HDB/etc), the recommended way of doing this is via Sound Forge (Windows) or Dolby Media Producer (OS X).

In order to encode AC-3 with Sound Forge, open the file (in w64 format), wait for it to index, then go to the Save As option. From here, select Dolby Digital AC-3 Pro. Then, select the following options (copied from AHD's guide):

- Bitstream mode: Main audio service: Complete main
- Audio coding mode: 3/2 (L, C, R, Ls, Rs) with LFE enabled
- Sample rate: 48 kHz
- Data rate: 640 kbps
- Dialog normalization: -32 dB
- Do not select Save data in Intel byte order
- Center mix level: -3 dB
- Surround mix level: -3 dB
- Do not set copyright bit
- do not mark as original bitstream
- Do not include audio production information
- Do not enable extended bitstream information
- Do not enable digital de-emphasis

- Enable DC high-pass filter
- Enable Bandwidth low-pass filter
- Enable LFE low-pass filter
- Do not enable 90-degree phase shift
- Do not enable 3 dB attenuation
- Line mode profile: None
- RF mode profile: None
- Do not enable RF overmodulation protection

The new E-AC-3 codec has already risen in popularity lately. What this brings to the table is support for Atmos and more channels. However, the only crack for the encoder doesn't include Atmos support, and its efficiency is the same as AC-3, as it's simply another layer packed on top. Because of these reasons, we feel no need to elaborate on how to encode this beyond stating that it requires Dolby Media Producer.

4.2.4 DTS

This codec is very common for 1080p private tracker encodes. Frankly put, it's only because of the bloated bitrate. DTS actually manages to be worse at 1509 kbps than AC-3 at 448 kbps. Just don't use this. The only reason to even consider it is laziness, since almost every movie source will feature a DTS core.

4.3 Lossless Codecs

4.3.1 FLAC

It's free, it's open source, it works everywhere, it's the most efficient. If you're in stereo or mono, it should be fine in terms of hardware compatibility. Otherwise, it may have issues with some receivers.

```
$ flac --compression-level-8 foo.wav bar.flac
```

4.3.2 TrueHD

This codec is less popular than DTS-HD MA due to inferior hardware support, but it's a bit more efficient, especially at lower bitrates, where the DTS core bloat doesn't inflate the bitrate. Sadly, due to its inferior hardware compatibility, lots of groups insist on including a separate AC-3 compatibility track. At Concertos, we'd prefer if you not do this, but understand that this may be necessary to upload elsewhere or simply something you want for hardware compatibility.

I don't know how to encode this. Ask Scrooge.

4.3.3 DTS-HD MA

Because of better support for using the core as fallback, this codec is quite popular for surround sound remuxes. You can encode it by extracting your track to multiple wavs, one for each channel, then dragging them into the encode suite and clicking the big red encode button. At Concertos, we don't care for this codec and prefer the use of TrueHD or FLAC.

5 Muxing

5.1 Appending Files

Sometimes, your encode will crash or you'll want to edit a small portion of an encode. In situations like these, we can simply append files in `mkvtoolnix`. If you just want to add something after another file, just right click the file in the source file field in `mkvtoolnix`.

For edits within videos, first locate the appropriate I-frame at the end so you actually encode enough, then encode your segment, open each file in `mkvtoolnix`, go to the splitting option in the output tab, then set the frames for the part of the video. Keep in mind counting starts at 1 here unlike Python where it starts at 0. Do this for the part leading up to and after where you want to add your change, then append them like before. You can also do this to create a hybrid video remux.

Make sure to check you have all the necessary frames after muxing, as you can easily end up with wrong I-frames being used or an incorrect frame count due to bad trimming if you don't pay perfect attention.

5.2 Seamless Linking

Just read this forum post: `https://forums.animesuki.com/showthread.php?t=66444`. There's nothing more that really has to be added.

6 Recommended Guides and Resources

- Irrational Encoding Wizardry's fansubbing guide: `https://guide.encode.moe`
- Kageru's blog: `https://kageru.moe/blog/`
- eXmendiC's filtering guide: `https://iamscum.wordpress.com/_test1/_test2/`
- AHD Guides

7 Contributors in Alphabetical Order

- Aicha @ D-Z0N3, NSDAB, REEEEEEEE
- Anon @ Concertos, NSDAB
- Q2KTyrant @ Concertos
- Ryuu @ Concertos, NSDAB
- Scrooge @ Concertos, COC, Fidelity™
- xCreamEnte @ EPSiLON, HDBEE, SiGMA

8 Appendix

Figure 16: Comparison of resizers for downscaling. The source was downscaled from 1080p to 720p, then cropped. The parameters were all left at their defaults.

Figure 17: Comparison of resizers for upscaling. The source (Commie encode of Your Name (2016)) was cropped, then upscaled by a factor of 2.

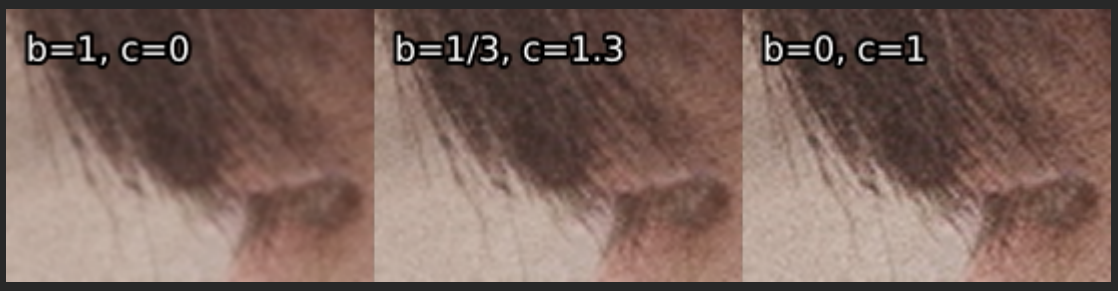

Figure 18: Comparison of bicubic upscales with different parameters. Keep in mind that these are just small crops of a video, and bicubic upscales will always lead to artifacts, especially aliasing. Sharp bicubic (b=0, c=1) is especially prone to aliasing artifacts.

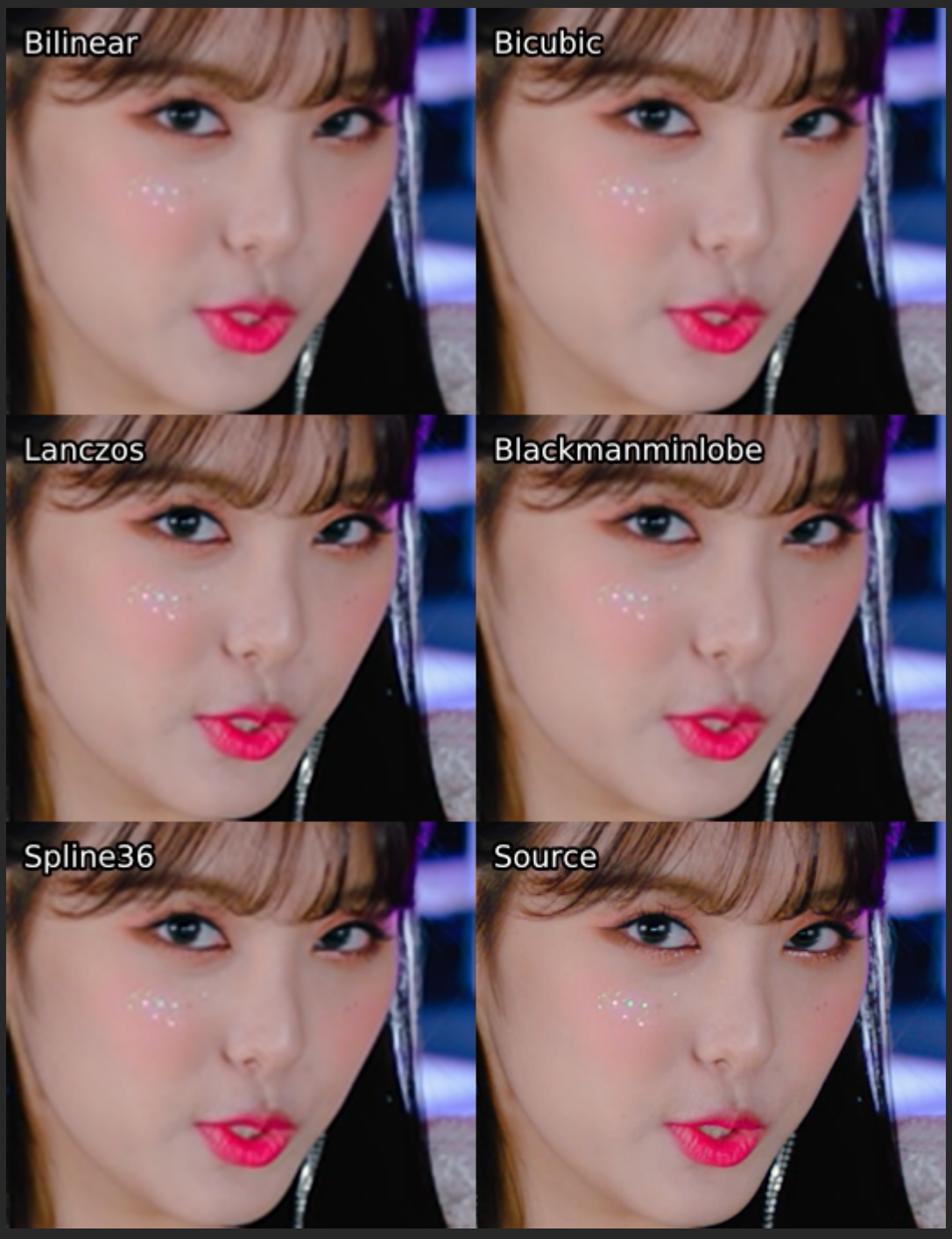

Figure 19: Comparison of resizers by downscaling and upscaling back to original resolution. Everything was downscaled to half the source resolution, then upscaled back with `nnedi3_rpow2`. The parameters were all left at their defaults.

Figure 20: Comparison of resizers by downscaling and upscaling back to original resolution. Everything was downscaled to half the source resolution, then upscaled back with the same resizer. The parameters were all left at their defaults.

Figure 21: Comparison of dither types. Input source was 8-bit video, dithered up and debanded by `f3kdb` with 16-bit output, then dithered to 8-bit.

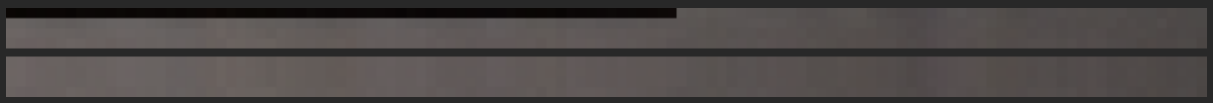

Figure 22: Example of why an improper fix of diagonal borders is a bad idea. Top is source, bottom is Geek's filtered. Taken from the Geek encode of Hotel Transylvania 2.

Figure 23: Comparison of gamma bug fix in 16-bit (right) and 8-bit (left).

Figure 24: Comparison of different edgemasks. From top left to bottom right: source, Prewitt, Sobel, Kirsch, TCanny, and `retinex_edgemask`. All settings were left at their defaults.

Figure 25: Comparison of the more accurate edgemasks. From top left to bottom right: source, `retinex_edgemask`, Kirsch, Sobel. Note that this is on a very poor quality source with terrible lighting, hence the difference between retinex and Kirsch masks is larger than usual.

Figure 26: Comparison of debanding with and without an edgemask. From top left to bottom right: source, source with cropped area marked, cropped area, mask on cropped area (`kgf.retinex_edgemask(src).std.Binarize(5000).std.Maximum().std.Inflate()`), maskless deband (`f3kdb.Deband(y=90, grainy=32, grainc=16)`), and masked deband. Gamma was raised by 50% via `SmoothLevels`.

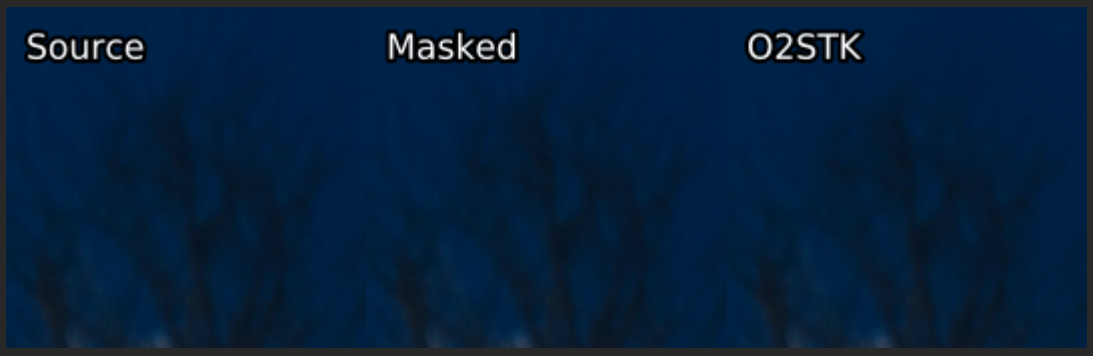

Figure 27: An example of why simple luma masks often aren't enough. Taken from the 1080p O2STK encode of Fantastic Mr. Fox. O2STK used `GradFun3` along with a luma mask, the Masked screenshot is `f3kdb` masked with a simple Sobel mask of the RGB screenshot.

Figure 28: From Geek's encode of Hotel Transylvania 2: another example of why you should always be wary of detail smoothing or even destruction when not masking properly. Although one can't be certain whether a mask was used here, it's likely this is the result of unmasked `f3kdb`. sunnighty seems to usually use `GradFun3` masking, which may have helped quite a bit in this case. Not sure what happened here, but I'm guessing they had issues with the mask picking up unintended stuff.