

# **Avisynth 2.5 Selected External Plugin Reference**

Wilbert Dijkhof

# Table of Contents

<b><u>1 Avisynth 2.5 Selected External Plugin Reference</u></b> .....	<b>1</b>
<u>1.1 General info</u> .....	1
<u>1.2 Deinterlacing &amp; Pulldown Removal</u> .....	2
<u>1.3 Spatio-Temporal Smoothers</u> .....	3
<u>1.4 Spatial Smoothers</u> .....	4
<u>1.5 Temporal Smoothers</u> .....	4
<u>1.6 Sharpen/Soften Plugins</u> .....	4
<u>1.7 Resizers</u> .....	5
<u>1.8 Subtitle (source) Plugins</u> .....	5
<u>1.9 MPEG Decoder (source) Plugins</u> .....	5
<u>1.10 Audio Decoder (source) Plugins</u> .....	6
<u>1.11 Plugins to compare video quality</u> .....	6
<u>1.12 Broadcast Video Plugins</u> .....	6
<u>1.13 Misc Plugins</u> .....	7
<b><u>2 AviSynth Frequently Asked Questions</u></b> .....	<b>9</b>
<u>2.1 Contents</u> .....	9
<u>2.1.1 General information</u> .....	9
<u>2.1.2 Loading clips (video, audio and images) into AviSynth</u> .....	9
<u>2.1.3 Opening scripts in encoder and player applications</u> .....	9
<u>2.1.4 Some common error messages</u> .....	9
<u>2.1.5 Recognizing and processing different types of content</u> .....	9
<u>2.1.6 The color format YV12 and related processing and encoding issues</u> .....	10
<u>2.1.7 How to use VirtualDub's plugins in AviSynth</u> .....	10
<b><u>3 AviSynth FAQ – Some common error messages</u></b> .....	<b>11</b>
<u>3.1 Contents</u> .....	11
<u>3.1.1 I got the message "LoadPlugin: unable to load "xxx (some plugin)" is not an AviSynth 1.0/AviSynth 2.0 plugin"?</u> .....	11
<u>3.1.2 When frameserving I got the following message: "Script error, there is no function named "xxx (some filter)"" ?</u> .....	11
<u>3.1.3 I installed AviSynth and got the following error message: "Couldn't locate decompressor for format 'YV12' (unknown)."?</u> .....	11
<u>3.1.4 When encoding I got the following error "ACM failed to suggest a compatible PCM format"?</u> .....	12
<u>3.1.5 When encoding I got the following error: "framesize xyz x 56 not supported"?</u> .....	12
<u>3.1.6 When frameserving I got the following message: "AVISource: couldn't locate a decompressor for fourcc (...)"?</u> .....	12
<u>3.1.7 When frameserving I got the following message: "DirectShowSource: Could not open as video or audio Video Returned: "DirectShowSource: the filter graph manager won't talk to me"?"?</u> .....	13
<u>3.1.8 I am trying to load a script that has a path name with a mix of japanese characters (someone's name) and Ascii test. I got an import error and the path that is displayed has some strange characters (not the japanese characters)?</u> .....	13
<u>3.1.9 When frameserving I got the following message: "CAVISTreamSynth: System exception – Access Violation at 0x0, reading from 0x0"?</u> .....	13
<u>3.1.10 When frameserving I got a message similar to: "Avisynth open failure: Script error: Invalid arguments to function "xxx (some filter)" (I:\Video.avs, line 5)"?</u> .....	13

# Table of Contents

<b><u>4 Avisynth FAQ – Recognizing and processing different types of content</u></b> .....	<b>15</b>
<u>4.1 Contents</u> .....	15
<u>4.1.1 The video and audio in my final encoding is out of sync, what should I do?</u> .....	15
<u>4.1.2 How do I recognize progressive, interlaced, telecined, hybrid and blended content?</u> .....	16
<u>4.1.3 How do I process interlaced content?</u> .....	16
<u>4.1.4 How do I process telecined content?</u> .....	17
<u>4.1.5 How do I process hybrid content?</u> .....	17
<u>4.1.6 What is variable framerate video?</u> .....	17
<u>4.1.7 How do I import variable framerate video into Avisynth and how do I process it?</u> .....	17
<b><u>5 Avisynth FAQ – Opening scripts in encoder and player applications</u></b> .....	<b>19</b>
<u>5.1 Contents</u> .....	19
<u>5.1.1 What is frameserving and what is it good for?</u> .....	19
<u>5.1.2 How do I use Avisynth as a frameserver?</u> .....	19
<u>5.1.3 How do I frameserve my AVS–file to encoder/application X?</u> .....	20
<u>5.1.3.1 Direct frameserving to compatible applications</u> .....	20
<u>5.1.3.2 Direct frameserving to applications using additional plugins</u> .....	20
<u>5.1.3.3 Direct frameserving to special or modified versions of encoders</u> .....	20
<u>5.1.3.4 Frameserving to applications via fake AVI files and proxy utilities</u> .....	21
<u>5.1.3.5 Frameserving via pipe from auxiliary programs to application–encoders</u> .....	21
<u>5.1.4 How do I solve problems when opening/reading scripts in encoders and players?</u> .....	21
<u>5.1.5 How do I frameserve from Premiere/Ulead/Vegas to Avisynth?</u> .....	22
<b><u>6 Avisynth FAQ – General information</u></b> .....	<b>23</b>
<u>6.1 Contents</u> .....	23
<u>6.1.1 What is Avisynth?</u> .....	23
<u>6.1.2 Who is developing Avisynth?</u> .....	24
<u>6.1.3 Where can I download the latest versions of Avisynth?</u> .....	24
<u>6.1.4 What are the main bugs in these versions?</u> .....	24
<u>6.1.5 Where can I find documentation about Avisynth?</u> .....	24
<u>6.1.6 How do I install/uninstall Avisynth?</u> .....	24
<u>6.1.7 What is the main difference between v1.0x, v2.0x, v2.5x, v2.6x and v3.x?</u> .....	24
<u>6.1.8 How do I know which version number of Avisynth I have?</u> .....	25
<u>6.1.9 How do I make an AVS–file?</u> .....	25
<u>6.1.10 Where do I save my AVS–file?</u> .....	25
<u>6.1.11 Are plugins compiled for v2.5x/v2.6x compatible with v1.0x/v2.0x and vice versa?</u> .....	25
<u>6.1.12 How do I use a plugin compiled for v2.0x in v2.5x?</u> .....	25
<u>6.1.13 How do I switch between different Avisynth versions without re–install?</u> .....	25
<u>6.1.14 VirtualdubMod, WMP6.4, CCE and other programs crash every time on exit (when previewing an avs file)?</u> .....	26
<u>6.1.15 My computer seems to crash at random during a second pass in any encoder?</u> .....	26
<u>6.1.16 Is there a command line utility for encoding to DivX/XviD using Avisynth?</u> .....	26
<u>6.1.17 Does Avisynth have a GUI (graphical user interface)?</u> .....	26
<b><u>7 Avisynth FAQ – Loading clips (video, audio and images) into Avisynth</u></b> .....	<b>28</b>
<u>7.1 Contents</u> .....	28
<u>7.1.1 Which media formats can be loaded into Avisynth?</u> .....	28
<u>7.1.2 Which possibilities exist to load my clip into Avisynth?</u> .....	28

# Table of Contents

<b><u>7 Avisynth FAQ – Loading clips (video, audio and images) into Avisynth</u></b>	
<u>7.1.3 What are the advantages and disadvantages of using DirectShowSource to load your media files?</u>	29
<u>7.1.4 Has Avisynth a direct stream copy mode like VirtualDub?</u>	29
<u>7.1.5 How do I load AVI files into Avisynth?</u>	29
<u>7.1.6 Can I load video with audio from AVI into Avisynth?</u>	30
<u>7.1.7 How do I load MPEG–1/MPEG–2/DVD VOB/TS/PVA into Avisynth?</u>	30
<u>7.1.8 How do I load QuickTime files into Avisynth?</u>	30
<u>7.1.9 How do I load raw source video files into Avisynth?</u>	31
<u>7.1.10 How do I load RealMedia files into Avisynth?</u>	31
<u>7.1.11 How do I load Windows Media Video files into Avisynth?</u>	31
<u>7.1.12 How do I load MP4/MKV/M2TS/EVO into Avisynth?</u>	32
<u>7.1.13 How do I load WAVE PCM files into Avisynth?</u>	32
<u>7.1.14 How do I load MP1/MP2/MP3/MPA/AC3/DTS/LPCM into Avisynth?</u>	33
<u>7.1.15 How do I load aac/flac/ogg files into Avisynth?</u>	33
<u>7.1.16 How do I load pictures into Avisynth?</u>	33
<b><u>8 Avisynth FAQ – Using VirtualDub plugins in Avisynth</u></b>	<b>35</b>
<u>8.1 Contents</u>	35
<u>8.1.1 Where can I download the latest version of scripts which import filters from VirtualDub?</u>	35
<u>8.1.2 Which filters can be imported?</u>	35
<u>8.1.3 Do these scripts work in RGB–space or in YUV–space?</u>	35
<u>8.1.4 How do I make such a script?</u>	35
<b><u>9 Avisynth FAQ – The color format YV12 and related processing and encoding issues</u></b>	<b>37</b>
<u>9.1 Contents</u>	37
<u>9.1.1 What is YV12?</u>	37
<u>9.1.2 Where can I download the latest stable Avisynth version which supports YV12?</u>	37
<u>9.1.3 Where can I download the DGIndex/DGDecode plugin, which supports YV12, to import MPEG–1/MPEG–2/TS/PVA in Avisynth ?</u>	37
<u>9.1.4 Which encoding programs support YV12?</u>	38
<u>9.1.5 How do I use v2.5x if the encoding programs can't handle YV12 (like TMPGEnc or CCE SP)?</u>	38
<u>9.1.6 What will be the main advantages of processing in YV12?</u>	38
<u>9.1.7 How do I use VirtualDub/VirtualDubMod such that there are no unnecessary color conversions?</u>	39
<u>9.1.8 Which internal filters support YV12?</u>	39
<u>9.1.9 Which external plugins support YV12?</u>	39
<u>9.1.10 Are there any disadvantages of processing in YV12?</u>	39
<u>9.1.11 How do I know which colorspace I'm using at a given place in my script?</u>	39
<u>9.1.12 The colors are swapped when I load a DivX file in Avisynth v2.5?</u>	39
<u>9.1.13 I got a green (or colored line) at the left or at the right of the clip, how do I get rid of it?</u>	40
<u>9.1.14 I installed Avisynth v2.5 and get the following error message: "Couldn't locate decompressor for format 'YV12' (unknown)."</u>	40
<u>9.1.15 I installed Avisynth v2.5 and DivX5 (or one of the latest Xvid builds of Koepi), all I got is a black screen when opening my avs in VirtualDub/VirtualDubMod/MPEG–2</u>	

# Table of Contents

<b><u>9 AviSynth FAQ – The color format YV12 and related processing and encoding issues</u></b>	
<u>encoder?</u>	40
<u>9.1.16 Are there any lossless YV12 codecs, which I can use for capturing for example?</u>	40
<u>9.1.17 Some important links:</u>	41
<u>9.3.0.1 Linear Editing:</u>	41
<u>9.3.0.2 Non-Linear Editing:</u>	42
<u>9.2 Filters with multiple input clips</u>	42
<u>9.3 Getting started</u>	43
<u>9.4 AviSynth Internet Links</u>	44
<u>9.5 Scripting reference</u>	45
<u>9.6 Arrays</u>	45
<u>9.7 Block statements</u>	47
<u>9.8 Background</u>	48
<u>9.8.1 Features enabling construction of block statements</u>	49
<u>9.9 Implementation Guide</u>	50
<u>9.9.1 The if..else block statement</u>	50
<u>9.9.1.1 Using Eval() and three-double-quotes quoted strings</u>	50
<u>9.9.1.2 Using separate scripts as blocks and the Import() function</u>	52
<u>9.9.1.3 Using functions (one function for each block)</u>	54
<u>9.9.2 The if..elif..else block statement</u>	55
<u>9.9.2.1 Using Eval() and three-double-quotes quoted strings</u>	55
<u>9.9.2.2 Using separate scripts as blocks and the Import() function</u>	56
<u>9.9.2.3 Using functions (one function for each block)</u>	56
<u>9.9.3 The for..next block statement</u>	56
<u>9.9.3.1 For..Next loop with access to variables in local scope</u>	57
<u>9.9.3.2 For..Next loop without access to variables in local scope</u>	58
<u>9.9.4 The do..while and do..until block statements</u>	58
<u>9.10 Deciding which implementation to use</u>	58
<u>9.10.1 The if..else and if..elif..else block statements</u>	59
<u>9.10.2 The for..next block statement</u>	59
<u>9.10.3 The do..while and do..until block statements</u>	59
<u>9.11 References</u>	59
<u>9.12 The script execution model</u>	59
<u>9.13 The script execution model – Evaluation of runtime scripts</u>	60
<u>9.14 Contents</u>	60
<u>9.14.1 Runtime environment initialisation</u>	61
<u>9.14.2 Runtime script parsing and evaluation</u>	61
<u>9.14.3 Runtime environment cleanup and delivery of the resulting frame</u>	61
<u>9.14.4 The runtime environment in detail</u>	61
<u>9.14.4.1 Elements of the runtime environment</u>	62
<u>9.14.4.2 Runtime functions and current frame</u>	62
<u>9.14.4.3 Checklist for developing correct runtime scripts</u>	63
<u>9.15 The script execution model – The fetching of frames</u>	63
<u>9.16 The script execution model – The filter graph</u>	64
<u>9.16.1 An example of a filter graph</u>	64
<u>9.17 The script execution model – Scope and lifetime of variables</u>	65
<u>9.18 Contents</u>	65
<u>9.18.1 Global scope</u>	65

# Table of Contents

<b><u>9 Avisynth FAQ – The color format YV12 and related processing and encoding issues</u></b>	
<u>9.18.2 Local scope</u>	65
<u>9.18.3 Lifetime of variables</u>	66
<u>9.18.4 A variables scope and lifetime example</u>	66
<u>9.18.5 Code–injecting language facilities and scopes</u>	67
<u>9.18.5.1 Import and Eval</u>	67
<u>9.18.5.2 Runtime scripts</u>	69
<u>9.18.5.3 The try...catch block</u>	70
<u>9.19 The script execution model – Performance considerations</u>	70
<u>9.19.1 Plugin auto–loading</u>	71
<u>9.19.2 Frame caching and the effect on splitting filter graph's paths</u>	71
<u>9.19.3 What not to include in runtime scripts</u>	72
<u>9.20 The script execution model – Sequence of events</u>	73
<u>9.20.1 The initialisation phase</u>	73
<u>9.20.2 The script loading and parsing phase</u>	74
<u>9.20.3 The video frames serving phase</u>	74
<u>9.21 User functions</u>	74
<u>9.21.1 Manipulating arguments</u>	75
<u>9.21.1.1 Optional arguments</u>	75
<u>9.21.1.2 var arguments</u>	75
<u>9.21.2 Manipulating globals</u>	75
<u>9.21.3 Recursion</u>	75
<u>9.21.4 Tuning performance</u>	75
<u>9.21.5 Design and coding–style considerations</u>	75
<u>9.21.6 Organising user defined functions</u>	75
<u>9.22 Avisynth Syntax</u>	75
<u>9.23 Avisynth Clip properties</u>	76
<u>9.24 Avisynth – Colors</u>	78
<u>9.25 Avisynth Syntax – Control structures</u>	79
<u>9.26 Contents</u>	79
<u>9.26.1 The try..catch statement</u>	79
<u>9.26.2 Other control structures (in the broad sense)</u>	80
<u>9.26.2.1 Example 1: Create a function that returns a n–times repeated character sequence</u>	81
<u>9.26.2.2 Example 2: Create a function that selects frames of a clip in arbitrary intervals</u>	81
<u>9.27 Avisynth Syntax – Internal functions</u>	82
<u>9.28 Avisynth Syntax – Boolean functions</u>	83
<u>9.29 Avisynth Syntax – Control functions</u>	84
<u>9.30 Avisynth Syntax – Conversion functions</u>	87
<u>9.31 Avisynth Syntax – Multithreading functions</u>	88
<u>9.32 Avisynth Syntax – Runtime functions</u>	89
<u>9.33 Avisynth Syntax – String functions</u>	91
<u>9.34 Avisynth Syntax – Version functions</u>	94
<u>9.35 Avisynth Syntax – Operators</u>	94
<u>9.35.1 Available Operators per Type</u>	94
<u>9.35.2 Operator Precedence</u>	96
<u>9.36 Avisynth Syntax – Plugins</u>	96
<u>9.37 Plugin autoload and name precedence v2</u>	97
<u>9.38 Plugin autoload and conflicting function names v2.55</u>	97

# Table of Contents

<b><u>9 Avisynth FAQ – The color format YV12 and related processing and encoding issues</u></b>	
<u>9.39 Avisynth Syntax</u>	98
<u>9.40 Avisynth Runtime environment</u>	100
<u>9.40.1 Contents</u>	100
<u>9.40.2 Definition</u>	100
<u>9.40.3 Runtime filters</u>	101
<u>9.40.4 Special runtime variables and functions</u>	101
<u>9.40.5 How to script inside the runtime environment</u>	102
<u>9.41 Avisynth Syntax – Script variables</u>	102
<u>9.42 Avisynth Syntax – User defined script functions</u>	104
<u>9.42.1 Definition and Structure</u>	104
<u>9.42.2 Facts about user defined script functions</u>	105
<u>9.42.3 Related Links</u>	106
<u>9.43 Troubleshooting</u>	106
<u>9.43.1 Contents</u>	106
<u>9.43.1.1 Installation problems</u>	106
<u>9.43.1.2 Other problems</u>	107
<u>9.43.1.3 Write Simple</u>	107
<u>9.43.1.4 Always check parameters</u>	107
<u>9.43.1.5 Test scripts using Virtualdub</u>	108
<u>9.43.1.6 Go through the script step by step</u>	108
<u>9.43.1.7 Check your autoloading plugin directory for a files</u>	108
<u>9.43.1.8 Use conservative image sizes</u>	109
<u>9.43.1.9 Finally check the Avisynth Q&amp;A</u>	109
<u>9.43.1.10 Reporting bugs / Asking for help</u>	109
<u>9.44 Advanced Topics</u>	110
<u>9.44.1 Interlaced and field-based video</u>	110
<u>9.44.2 Color format conversions, the Chroma Upsampling Error and the 4:2:0 Interlaced Chroma Problem</u>	110
<u>9.44.3 Colorspace Conversions</u>	110
<u>9.44.4 Wrong levels and colors upon playback</u>	110
<u>9.44.5 Avisynth, variable framerate (vfr) video and Hybrid video</u>	110
<u>9.44.6 Importing your media in Avisynth</u>	111
<u>9.44.7 Resizing</u>	111
<u>9.45 Color conversions</u>	111
<u>9.45.1 Converting to programming values</u>	112
<u>9.46 References</u>	112
<b><u>10 Avisynth, variable framerate (vfr) video and hybrid video</u></b>	<b>113</b>
<u>10.1 Table of contents</u>	113
<u>10.2 Variable framerate and hybrid video</u>	113
<u>10.3 How to recognize vfr content (mkv/mp4)</u>	114
<u>10.4 Opening MPEG-2 hybrid video in Avisynth and re-encoding</u>	115
<u>10.4.1 encoding to cfr (23.976 fps or 29.97 fps)</u>	115
<u>10.4.2 encoding to cfr – 120 fps</u>	116
<u>10.4.3 encoding to vfr (mkv)</u>	117
<u>10.4.4 encoding to vfr (mp4)</u>	118
<u>10.4.5 summary of the methods</u>	118

# Table of Contents

<b><u>10 Avisynth, variable framerate (vfr) video and hybrid video</u></b>	
<u>10.5 Opening non MPEG-2 hybrid video in AviSynth and re-encoding</u>	118
<u>10.5.1 opening non-avi vfr content in AviSynth</u>	118
<u>10.5.2 re-encoding 120 fps video</u>	119
<u>10.5.3 converting vfr to cfr avi for AviSynth</u>	119
<u>10.5.4 encoding to MPEG-2 vfr video</u>	120
<u>10.6 Audio synchronization</u>	120
<u>10.7 References</u>	121
<b><u>11 Importing media into AviSynth</u></b>	<b>122</b>
<u>11.1 Contents</u>	122
<u>11.2 Loading clips into AviSynth</u>	122
<u>11.2.1 1.1) Loading clips with video and audio into AviSynth</u>	123
<u>11.2.1.1 1.1.1) AVI with audio</u>	123
<u>11.2.1.2 1.1.2) Other containers with audio</u>	124
<u>11.2.2 1.2) Loading video clips into AviSynth</u>	127
<u>11.2.3 1.3) Loading audio clips into AviSynth</u>	129
<u>11.2.4 1.4) Loading images into AviSynth</u>	132
<u>11.3 2) Changing the merit of DirectShow Filters</u>	132
<u>11.4 3) Using GraphEdit to make graphs of DirectShow filters and loading these graphs in AviSynth</u>	132
<u>11.5 To Do</u>	135
<u>11.6 Interlaced and Field-based video</u>	135
<u>11.6.1 Color conversions and interlaced / field-based video</u>	135
<u>11.6.2 Color conversions, interlaced / field-based video and the interlaced flag of dvd2avi</u>	136
<u>11.6.3 Changing the order of the fields of a clip</u>	137
<u>11.6.3.1 Swapping fields</u>	137
<u>11.6.3.2 Reversing field dominance</u>	138
<u>11.6.4 The parity (= order) of the fields in AviSynth</u>	139
<u>11.6.5 About DV / DVD in relation to field dominance</u>	139
<u>11.6.6 References</u>	139
<b><u>12 Resampling</u></b>	<b>140</b>
<u>12.1 Resampling</u>	140
<u>12.1.1 The Nyquist-Shannon sampling theorem</u>	140
<u>12.1.2 Example: the bilinear resampling filter – upsampling</u>	141
<u>12.1.3 Example: the bilinear resampling filter – downsampling</u>	143
<u>12.1.4 Example: the bicubic resampling filter – upsampling</u>	145
<u>12.2.0.1 AviSynth's implementation</u>	146
<u>12.2 An implementation of the resampling filters</u>	147
<u>12.3 Resizing Algorithms</u>	149
<u>12.3.1 Nearest Neighbour resampler</u>	149
<u>12.3.2 Bilinear resampler</u>	150
<u>12.3.3 Bicubic resampler</u>	151
<u>12.3.4 Windowed sinc resamplers</u>	153
<u>12.3.4.1 Sinc resampler</u>	153
<u>12.3.4.2 Lanczos resampler</u>	154
<u>12.3.4.3 Blackman resampler</u>	155



# Table of Contents

## 12 Resampling

<a href="#">12.3.5 Spline resampler</a> .....	156
<a href="#">12.3.6 Gaussian resampler</a> .....	159
<a href="#">12.4</a> .....	160

## 13 Sampling.....

<a href="#">13.1 1. Color format conversions and the Chroma Upsampling Error</a> .....	162
<a href="#">13.1.1 1.1 Chroma Upsampling Error (or CUE)</a> .....	163
<a href="#">13.1.2 1.2 Correcting video having the Chroma Upsampling Error</a> .....	164
<a href="#">13.2 2. Theoretical Aspects</a> .....	164
<a href="#">13.2.1 2.1 The color formats: RGB, YUY2 and YV12</a> .....	164
<a href="#">13.2.1.1 YUV 4:4:4 color format</a> .....	164
<a href="#">13.2.1.2 YUY2 color format</a> .....	164
<a href="#">13.2.1.3 YV12 color format</a> .....	165
<a href="#">13.2.2 2.2 Subsampling</a> .....	166
<a href="#">13.2.2.1 RGB -&gt; YUY2 conversion</a> .....	166
<a href="#">13.2.3 2.3 Upsampling</a> .....	168
<a href="#">13.2.3.1 YUY2 conversion -&gt; RGB</a> .....	168
<a href="#">13.2.3.2 YV12 interlaced conversion -&gt; YUY2</a> .....	169
<a href="#">13.2.4 2.4 References</a> .....	172
<a href="#">13.2.5 3.1 MPEG-1 versus MPEG-2 sampling</a> .....	172
<a href="#">13.2.6 3.2 DV sampling</a> .....	173
<a href="#">13.2.7 3.3 References</a> .....	174
<a href="#">13.25.0.1 To strictly preserve the original fields and just fill in the missing lines</a> .....	174
<a href="#">13.33.0.1 Other internal functions</a> .....	174
<a href="#">13.3 4. 4:2:0 Interlaced Chroma Problem (or ICP)</a> .....	174
<a href="#">13.4 Introduction</a> .....	175
<a href="#">13.5 Media file filters</a> .....	176
<a href="#">13.6 Color conversion and adjustment filters</a> .....	176
<a href="#">13.7 Overlay and Mask filters</a> .....	177
<a href="#">13.8 Geometric deformation filters</a> .....	177
<a href="#">13.9 Pixel restoration filters</a> .....	178
<a href="#">13.10 Timeline editing filters</a> .....	178
<a href="#">13.11 Interlace filters</a> .....	179
<a href="#">13.12 Audio processing filters</a> .....	179
<a href="#">13.13 Meta filters</a> .....	180
<a href="#">13.14 Conditional filters</a> .....	180
<a href="#">13.15 Debug filters</a> .....	181
<a href="#">13.16 AddBorders</a> .....	182
<a href="#">13.17 RGBAdjust</a> .....	183
<a href="#">13.18 Amplify / AmpifydB</a> .....	185
<a href="#">13.19 Animate / ApplyRange</a> .....	185
<a href="#">13.20 AssumeSampleRate</a> .....	186
<a href="#">13.21 AudioDub / AudioDubEx</a> .....	187
<a href="#">13.22 AVISource / OpenDMLSource / AVIFileSource / WAVSource</a> .....	189
<a href="#">13.23 BlankClip / Blackness</a> .....	189
<a href="#">13.24 Blur / Sharpen</a> .....	191
<a href="#">13.25 Bob</a> .....	191

# Table of Contents

## 13 Sampling

<a href="#">13.26 ColorBars</a> .....	192
<a href="#">13.27 ColorYUV</a> .....	193
<a href="#">13.28 Compare</a> .....	194
<a href="#">13.29 ConditionalFilter</a> .....	194
<a href="#">13.30 ScriptClip</a> .....	195
<a href="#">13.31 FrameEvaluate</a> .....	195
<a href="#">13.32 ConditionalReader</a> .....	196
<a href="#">13.33 Runtime Functions</a> .....	196
<a href="#">13.34 Advanced conditional filtering: part I</a> .....	197
<a href="#">13.35 Advanced conditional filtering: part II</a> .....	199
<a href="#">13.36 Advanced conditional filtering: part III</a> .....	199
<a href="#">13.37 ConditionalReader</a> .....	202
<a href="#">13.37.1 Parameters</a> .....	202
<a href="#">13.37.2 File format</a> .....	202
<a href="#">13.37.3 Types</a> .....	203
<a href="#">13.37.4 Examples</a> .....	203
<a href="#">13.37.4.1 Basic usage</a> .....	203
<a href="#">13.37.4.2 Adjusting Overlay</a> .....	204
<a href="#">13.37.4.3 Complicated ApplyRange</a> .....	205
<a href="#">13.38 ConvertBackToYUY2 / ConvertToRGB / ConvertToRGB24 / ConvertToRGB32 / ConvertToY8 / ConvertToYUY2 / ConvertToYV12 / ConvertToYV16 / ConvertToYV24 / ConvertToYV411</a> .....	205
<a href="#">13.39 ConvertAudioTo8bit / ConvertAudioTo16bit / ConvertAudioTo24bit / ConvertAudioTo32bit / ConvertAudioToFloat</a> .....	209
<a href="#">13.40 ConvertToMono</a> .....	209
<a href="#">13.41 GeneralConvolution</a> .....	209
<a href="#">13.42 Crop / CropBottom</a> .....	211
<a href="#">13.43 Memory alignment</a> .....	212
<a href="#">13.44 Crop restrictions</a> .....	212
<a href="#">13.45 DelayAudio</a> .....	213
<a href="#">13.46 DeleteFrame</a> .....	213
<a href="#">13.47 DirectShowSource</a> .....	213
<a href="#">13.47.1 Examples</a> .....	215
<a href="#">13.47.2 Troubleshooting video and audio problems</a> .....	216
<a href="#">13.47.2.1 RenderFile, the filter graph manager won't talk to me</a> .....	216
<a href="#">13.47.2.2 The samplerate is wrong</a> .....	216
<a href="#">13.47.2.3 My sound is choppy</a> .....	216
<a href="#">13.47.2.4 My sound is out of sync</a> .....	216
<a href="#">13.47.2.5 My ASF renders start fast and finish slow</a> .....	216
<a href="#">13.47.3 Common tasks</a> .....	217
<a href="#">13.47.3.1 Opening GRF files</a> .....	217
<a href="#">13.47.3.2 Downmixing AC3 to stereo</a> .....	217
<a href="#">14.7.0.1 Remark1</a> .....	221
<a href="#">14.7.0.2 Remark2</a> .....	221
<a href="#">13.48 Dissolve</a> .....	222
<a href="#">13.49 DoubleWeave</a> .....	222
<a href="#">13.50 DuplicateFrame</a> .....	222

# Table of Contents

## **13 Sampling**

<a href="#"><u>13.51 EnsureVBRMP3Sync</u></a> .....	224
<a href="#"><u>13.52 FadeIn / FadeIn0 / FadeIn2 / FadeIO0 / FadeIO / FadeIO2 / FadeOut / FadeOut0 / FadeOut2</u></a> .....	225
<a href="#"><u>13.53 FixBrokenChromaUpsampling</u></a> .....	225

## **14 FixLuminance**.....225

<a href="#"><u>14.1 FlipHorizontal / FlipVertical</u></a> .....	226
<a href="#"><u>14.2 AssumeFPS</u></a> .....	227
<a href="#"><u>14.3 AssumeScaledFPS</u></a> .....	227
<a href="#"><u>14.4 ChangeFPS</u></a> .....	229
<a href="#"><u>14.5 ConvertFPS</u></a> .....	229
<a href="#"><u>14.6 FreezeFrame</u></a> .....	230
<a href="#"><u>14.7 GetChannel</u></a> .....	231
<a href="#"><u>14.8 Greyscale</u></a> .....	231
<a href="#"><u>14.9 Histogram</u></a> .....	231
<a href="#"><u>14.9.1 Classic mode</u></a> .....	231
<a href="#"><u>14.9.2 Levels mode</u></a> .....	232
<a href="#"><u>14.9.3 Color mode</u></a> .....	232
<a href="#"><u>14.9.4 Color2 mode</u></a> .....	233
<a href="#"><u>14.9.5 Luma mode</u></a> .....	234
<a href="#"><u>14.9.6 Stereo and StereoOverlay mode</u></a> .....	234
<a href="#"><u>14.9.7 AudioLevels mode</u></a> .....	235
<a href="#"><u>14.33.0.1 Parameters</u></a> .....	236
<a href="#"><u>14.33.1 RGB considerations</u></a> .....	237
<a href="#"><u>14.33.2 Conditional Variables</u></a> .....	238
<a href="#"><u>14.33.3 Examples</u></a> .....	239
<a href="#"><u>14.33.4 Changelog</u></a> .....	239
<a href="#"><u>14.10 ImageReader / ImageSource</u></a> .....	240
<a href="#"><u>14.11 ImageWriter</u></a> .....	240
<a href="#"><u>14.12 Import</u></a> .....	240
<a href="#"><u>14.13 Info</u></a> .....	242
<a href="#"><u>14.14 Interleave</u></a> .....	242
<a href="#"><u>14.15 Invert</u></a> .....	242
<a href="#"><u>14.16 KillAudio / KillVideo</u></a> .....	243
<a href="#"><u>14.17 Layer [yuy2][rgb32]</u></a> .....	243
<a href="#"><u>14.18 Mask [rgb32]</u></a> .....	244
<a href="#"><u>14.19 ResetMask [rgb32]</u></a> .....	245
<a href="#"><u>14.20 ColorKeyMask [rgb32]</u></a> .....	246
<a href="#"><u>14.21 Letterbox</u></a> .....	247
<a href="#"><u>14.22 Levels</u></a> .....	248
<a href="#"><u>14.23 Limiter</u></a> .....	249
<a href="#"><u>14.24 Loop</u></a> .....	250
<a href="#"><u>14.25 MaskHS</u></a> .....	250
<a href="#"><u>14.26 Merge / MergeChroma / MergeLuma</u></a> .....	250
<a href="#"><u>14.27 MergeChannels</u></a> .....	251
<a href="#"><u>14.28 MergeARGB / MergeRGB</u></a> .....	252
<a href="#"><u>14.29 MessageClip</u></a> .....	252

# Table of Contents

## 14 FixLuminance

<a href="#"><u>14.30 MixAudio</u></a> .....	254
<a href="#"><u>14.31 MonoToStereo</u></a> .....	254
<a href="#"><u>14.32 Normalize</u></a> .....	255
<a href="#"><u>14.33 Overlay</u></a> .....	257
<a href="#"><u>14.34 AssumeFrameBased / AssumeFieldBased</u></a> .....	257
<a href="#"><u>14.35 AssumeTFF / AssumeBFF</u></a> .....	257
<a href="#"><u>14.36 ComplementParity</u></a> .....	257
<a href="#"><u>14.37 PeculiarBlend</u></a> .....	258
<a href="#"><u>14.38 Pulldown</u></a> .....	258
<a href="#"><u>14.39 HorizontalReduceBy2 / VerticalReduceBy2 / ReduceBy2</u></a> .....	259
<a href="#"><u>14.40 ResampleAudio</u></a> .....	260
<a href="#"><u>14.41 BicubicResize / BilinearResize / BlackmanResize / GaussResize / LanczosResize / Lanczos4Resize / PointResize / SincResize / Spline16Resize / Spline36Resize / Spline64Resize</u></a> .....	260
<a href="#"><u>14.41.1 General information</u></a> .....	261
<a href="#"><u>14.41.2 BilinearResize</u></a> .....	261
<a href="#"><u>14.41.3 BicubicResize</u></a> .....	262
<a href="#"><u>14.41.4 BlackmanResize</u></a> .....	263
<a href="#"><u>14.41.5 GaussResize</u></a> .....	263
<a href="#"><u>14.41.6 LanczosResize / Lanczos4Resize</u></a> .....	263
<a href="#"><u>14.41.7 PointResize</u></a> .....	263
<a href="#"><u>14.41.8 Spline16Resize/Spline36Resize/Spline64Resize</u></a> .....	264
<a href="#"><u>14.41.9 SincResize</u></a> .....	264
<a href="#"><u>14.42 Reverse</u></a> .....	264
<a href="#"><u>14.43 SegmentedAVISource / SegmentedDirectShowSource</u></a> .....	264
<a href="#"><u>14.44 SelectEven / SelectOdd</u></a> .....	265
<a href="#"><u>14.45 SelectEvery</u></a> .....	265
<a href="#"><u>14.46 SelectRangeEvery</u></a> .....	266
<a href="#"><u>14.47 SeparateFields</u></a> .....	266
<a href="#"><u>14.48 ShowAlpha, ShowRed, ShowGreen, ShowBlue</u></a> .....	267
<a href="#"><u>14.49 ShowFiveVersions</u></a> .....	267
<a href="#"><u>14.50 ShowFrameNumber</u></a> .....	268
<a href="#"><u>14.51 ShowSMPTE</u></a> .....	268
<a href="#"><u>14.52 ShowTime</u></a> .....	269
<a href="#"><u>14.53 SpatialSoften / TemporalSoften</u></a> .....	269
<a href="#"><u>14.54 SoundOut</u></a> .....	271
<a href="#"><u>14.54.1 Installation and Usage</u></a> .....	271
<a href="#"><u>14.54.2 Output Modules</u></a> .....	271
<a href="#"><u>14.54.2.1 WAV/AIF/CAF</u></a> .....	271
<a href="#"><u>14.54.2.2 FLAC</u></a> .....	271
<a href="#"><u>14.54.2.3 APE</u></a> .....	271
<a href="#"><u>14.54.2.4 MP2</u></a> .....	272
<a href="#"><u>14.54.2.5 MP3</u></a> .....	272
<a href="#"><u>14.54.2.6 AC3</u></a> .....	272
<a href="#"><u>14.54.2.7 OGG</u></a> .....	272
<a href="#"><u>14.54.2.8 Commandline Output</u></a> .....	272
<a href="#"><u>14.54.3 Exporting from script</u></a> .....	273
<a href="#"><u>14.54.3.1 General Parameters</u></a> .....	273

# Table of Contents

## 14 FixLuminance

<a href="#">14.54.3.2 WAV/AIF/CAF Script Parameters:</a>	273
<a href="#">14.54.3.3 FLAC Script Parameters:</a>	274
<a href="#">14.54.3.4 APE Script Parameters:</a>	274
<a href="#">14.54.3.5 MP2 Script Parameters:</a>	274
<a href="#">14.54.3.6 MP3 Script Parameters:</a>	275
<a href="#">14.54.3.7 AC3 Script Parameters:</a>	275
<a href="#">14.54.3.8 OGG Script Parameters:</a>	276
<a href="#">14.54.3.9 Wavpack Script Parameters:</a>	276
<a href="#">14.54.3.10 Commandline Output Script Parameters:</a>	276
<a href="#">14.54.4 Examples</a>	277
<a href="#">14.54.5 Implementation notes</a>	277
<a href="#">14.54.6 Changelist</a>	277
<a href="#">14.58.0.1 Parameters</a>	280
<a href="#">14.55 AlignedSplice / UnalignedSplice</a>	280
<a href="#">14.56 SSRC</a>	281
<a href="#">14.57 StackHorizontal / StackVertical</a>	282
<a href="#">14.58 Subtitle</a>	282
<a href="#">14.59 Subtract</a>	285
<a href="#">14.60 SuperEQ</a>	285
<a href="#">14.61 SwapUV / UToY / VToY / YToUV</a>	286
<a href="#">14.62 SwapFields</a>	287
<a href="#">14.63 TCPServer / TCPSource</a>	287
<a href="#">14.63.1 Syntax</a>	287
<a href="#">14.63.1.1 Server:</a>	287
<a href="#">14.63.1.2 Client:</a>	288
<a href="#">14.63.2 Examples</a>	288
<a href="#">14.63.3 Usability Notes</a>	289
<a href="#">14.63.4 Changelog</a>	289
<a href="#">14.64 TimeStretch</a>	289
<a href="#">14.65 Tone</a>	292
<a href="#">14.66 Trim</a>	293
<a href="#">14.67 TurnLeft / TurnRight / Turn180</a>	293
<a href="#">14.68 Tweak</a>	293
<a href="#">14.68.1 Usage and examples: adjusting contrast and brightness</a>	294
<a href="#">14.68.2 Usage and examples: adjusting saturation</a>	295
<a href="#">14.71.0.1 Usage is best explained with some simple examples:</a>	297
<a href="#">14.71.0.2 There are easier ways to write numbers in a file, BUT:</a>	297
<a href="#">14.71.0.3 More examples:</a>	298
<a href="#">14.69 Version</a>	299
<a href="#">14.70 Weave</a>	299
<a href="#">14.71 WriteFile / WriteFileIf / WriteFileStart / WriteFileEnd</a>	299
<a href="#">14.72 AviSynth License</a>	300
<a href="#">14.72.1 AviSynth documentation license</a>	300
<a href="#">14.72.2 AviSynth logo</a>	301
<a href="#">14.72.3 Avisynth uses codes from following projects:</a>	302
<a href="#">14.72.4 Thanks also to everyone else contributing to the AviSynth project!</a>	303

# 1 Avisynth 2.5 Selected External Plugin Reference

## 1.1 General info

AviSynth plugins are external linked modules (libraries) implementing one or more additional functions (filters) for deinterlacing, noise reduction, etc. Great number of plugins (200 or more) were written by independent developers as free software. You can get them at [collection by WarpEnterprises](#) or (most recent) at homepages of authors (see also AviSynth Web site and forums). This documentation includes authors descriptions of selected popular plugins with typical useful functions. It can be good reading even if you will use some other (similar) plugins.

The description of a plugin starts with some general info about the plugin. For example

**author:** Rainer Wittmann (aka kassandro)

**version:** 0.6.1

**download:** <http://www.removedirt.de.tf/>

**category:** Temporal Smoothers

**requirements:**

- YV12 & YUY2 Colorspace
- SSEMMX support
- width and the height of the clip must be a multiple of 8

**license:** GPL

The first line "**author**" gives the author(s) of the plugin. It can be his/her real name, the forum nickname, or both.

The second line "**version**" gives the version of the plugin on which is described in *this* documentation. Note that it might not be the most recent version of the plugin.

The third line "**download**" gives the download page of the plugin.

The fourth line "**category**", is the category under which the plugin can be found.

The fifth line "**requirements**", are the requirements for being able to use the plugin. Requirements can be the supported color space (YV12 and YUY2 in this case), the supported CPU (which is SSEMMX here) or that the width/height must be a multiple of some number (usually 8 or 16). The latter is a requirement for the optimizations in the plugin.

The last line "**license**" gives the license of the plugin. Usually it is none (ie closed source), GPL or just open source (without a specific license).

A final note about the CPUs which might be required for a plugin. If you don't know what kind of optimizing instructions are used by your CPU, you can look it up in the following table

optimizing instructions	CPU
MMX	Pentium MMX, Pentium II, K6, K6II, K6III and later
iSSE (also called SSEMMX)	Pentium III, all Duron (called 3DNow extension), all Athlon (called 3DNow extension)
SSE	

## Avisynth 2.5 Selected External Plugin Reference

	Pentium III, Duron (core Morgan), Athlon XP and later
SSE2	P-IV, Opteron, Athlon 64
SSE3	P-IV Prescott

## 1.2 Deinterlacing & Pulldown Removal

*All PAL, NTSC, and SECAM video is interlaced, which means that only every other line is broadcast at each refresh interval. Deinterlacing filters let you take care of any problems caused by this. IVTC (inverse telecine, aka pulldown removal) filters undo the telecine process, which comes from differences between the timing of your video and its original source.*

<a href="#">Decomb Filter package (by Donald Graft)</a>	This package of plugin functions for AviSynth provides the means for removing combing artifacts from telecined progressive streams, interlaced streams, and mixtures thereof. Functions can be combined to implement inverse telecine for both NTSC and PAL streams. <a href="#">[discussion]</a> .
<a href="#">DGBob (by Donald Graft)</a>	This filter splits each field of the source into its own frame and then adaptively creates the missing lines either by interpolating the current field or by using the previous field's data. <a href="#">[discussion]</a> .
<a href="#">FDecimate (by Donald Graft)</a>	This filter provides extended decimation capabilities not available from Decimate(). It can remove frames from a clip to achieve the desired frame rate, while retaining audio/video synchronization. It preferentially removes duplicate frames where possible. <a href="#">[discussion]</a> .
<a href="#">GreedyHMA (by Tom Barry)</a>	DScaler's Greedy/HM algorithm code to perform pulldown matching, filtering, and video deinterlacing. <a href="#">[discussion]</a> .
<a href="#">IBob (by Kevin Atkinson)</a>	This simple filter works identically to the Avisynth built-in Bob filter except that it uses linear interpolation instead of bicubic resizing. <a href="#">[discussion]</a> .
<a href="#">KernelDeint (by Donald Graft)</a>	This filter deinterlaces using a kernel approach. It gives greatly improved vertical resolution in deinterlaced areas compared to simple field discarding. <a href="#">[discussion]</a> .
<a href="#">LeakKernelDeint (mod of KernelDeint by Leak)</a>	This filter deinterlaces using a kernel approach. It gives greatly improved vertical resolution in deinterlaced areas compared to simple field discarding. <a href="#">[discussion]</a> .
<a href="#">MultiDecimate (by Donald Graft)</a>	Removes N out of every M frames, taking the frames most similar to their predecessors. <a href="#">[discussion]</a> .
<a href="#">SmartDecimate (by Kevin Atkinson)</a>	This filter removes telecine by combining telecine fields and decimating at the same time, which is different from the traditional approach of matching telecine frames and then removing duplicates. <a href="#">[discussion]</a> .
<a href="#">TDeint (by tritical)</a>	TDeint is a bi-directionally, motion adaptive (sharp) deinterlacer. It can also adaptively choose between using per-field and per-pixel motion adaptivity. It can use cubic interpolation, kernel interpolation (with temporal direction switching), or one of two

## Avisynth 2.5 Selected External Plugin Reference

	forms of modified ELA interpolation which help to reduce "jaggy" edges in moving areas where interpolation must be used. TDeint also supports user overrides through an input file, and can act as a smart bobber or same frame rate deinterlacer, as well as an IVTC post-processor. <a href="#">[discussion]</a> .
<a href="#">TIVTC Filter package (by tritical)</a>	This package of plugin functions for AviSynth provides the means for removing combing artifacts from telecined progressive streams, interlaced streams, and mixtures thereof. Functions can be combined to implement inverse telecine for both NTSC and PAL streams.
<a href="#">TomsMoComp "Motion compensated deinterlace filter" (by Tom Barry)</a>	This filter uses motion compensation and adaptive processing to deinterlace video source (not for NTSC film). <a href="#">[discussion]</a> .
<a href="#">UnComb IVTC (by Tom Barry)</a>	Filter for matching up even and odd fields of properly telecined NTSC or PAL film source video. <a href="#">[discussion]</a> .

### 1.3 Spatio-Temporal Smoothers

*These filters use color similarities and differences both within and between frames to reduce noise and improve compressed size. They can greatly improve noisy video, but some care should be taken with them to avoid blurred movement and loss of detail.*

<a href="#">Deen (by Marc FD)</a>	Several denoisers. <a href="#">[discussion]</a>
<a href="#">Convolution3D / Convolution3DYV12 (by Vlad59)</a>	Convolution3D is a spatio-temporal smoother, it applies a 3D convolution filter to all pixels of consecutive frames. <a href="#">[discussion]</a> .
<a href="#">FluxSmooth (by SansGrip)</a>	Fluctuating pixels are wiped from existence by averaging it with its neighbours. <a href="#">[discussion]</a> .
<a href="#">FFT3DFilter (by Fizick)</a>	FFT3DFilter is 3D Frequency Domain filter – strong denoiser and moderate sharpener. <a href="#">[discussion]</a> .
<a href="#">FFT3DGPU (by tsp)</a>	FFT3dGPU is a GPU version of FFT3DFilter. <a href="#">[discussion]</a> .
<a href="#">NoMoSmooth (by SansGrip)</a>	A motion-adaptive spatio-temporal smoother. <a href="#">[discussion]</a> .
<a href="#">MipSmooth (by Sh0dan)</a>	It takes the source frame, and creates three new versions, each half the size of the previous. They are scaled back to original size. They are compared to the original, and if the difference is below the threshold, the information is used to form the final pixel. <a href="#">[discussion]</a> .
<a href="#">PeachSmoother (by Lindsey Dubb)</a>	An adaptive smoother optimized for TV broadcasts. The Peach works by looking for good pixels and gathering orange smoke from them. When it has gathered enough orange smoke, it sprinkles that onto the bad pixels, making them better. <a href="#">[discussion]</a> .
<a href="#">STMedianFilter "SpatioTemporal Median Filter" (by Tom Barry)</a>	STMedianFilter is a (slightly motion compensated) spatial/temporal median filter.



## 1.4 Spatial Smoothers

*These use color similarities and differences within a frame to improve the picture and reduce compressed size. They can smooth out noise very well, but overly aggressive settings for them can cause a loss of detail.*

<a href="#">MSmooth "Masked Smoother" (by Donald Graft)</a>	This filter is effective at removing mosquito noise as well as effectively smoothing flat areas in anime. [ <a href="#">discussion</a> ].
<a href="#">SmoothUV (by Kurosu)</a>	This filter can be used to reduce rainbows, as done by SmartSmoothIQ. [ <a href="#">discussion</a> ].
<a href="#">TBilateral (by tritical)</a>	TBilateral is a spatial smoothing filter that uses the bilateral filtering algorithm. It does a nice job of smoothing while retaining picture structure. [ <a href="#">discussion</a> ]
<a href="#">VagueDenoiser (by Lefungus)</a>	A simple denoiser that uses wavelets. [ <a href="#">discussion</a> ].

## 1.5 Temporal Smoothers

*These filters use color similarities and differences between frames to improve the picture and reduce compressed size. They can get rid of most noise in stationary areas without losing detail, but overly strong settings can cause moving areas to be blurred.*

<a href="#">Cnr2 "Chroma Noise Reducer" (by Marc FD)</a>	Reduces the noise on the chroma (UV) and preserves the luma (Y). [ <a href="#">discussion</a> ].
<a href="#">GrapeSmoother (by Lindsey Dubb)</a>	When colors change just a little, the filter decides that it is probably noise, and only slightly changes the color from the previous frame. As the change in color increases, the filter becomes more and more convinced that the change is due to motion rather than noise, and the new color gets more and more weight. [ <a href="#">discussion</a> ].
<a href="#">RemoveDirt (by kassandra)</a>	A temporal cleaner with strong protection against artifacts. [ <a href="#">discussion</a> ].
<a href="#">TemporalCleaner (by Jim Casaburi; ported to AviSynth by Vlad59)</a>	A simple but very fast temporal denoiser, aimed to improve compressibility.
<a href="#">TTempSmooth (by tritical)</a>	TTempSmooth is a motion adaptive (it only works on stationary parts of the picture), temporal smoothing filter. [ <a href="#">discussion</a> ]

## 1.6 Sharpen/Soften Plugins

*These are closely related to the Spatial Smoothers, above. They attempt to improve image quality by sharpening or softening edges.*

<a href="#">asharp (by Marc FD)</a>	Adaptive sharpening filter. [ <a href="#">discussion</a> ].
<a href="#">aWarpSharp (by Marc FD)</a>	A warp sharpening filter.

## Avisynth 2.5 Selected External Plugin Reference

<a href="#">MSharpen (by Donald Graft)</a>	This plugin for AviSynth implements an unusual concept in spatial sharpening. Although designed specifically for anime, it also works quite well on normal video. The filter is very effective at sharpening important edges without amplifying noise. <a href="#">[discussion]</a> .
<a href="#">TUnsharp (by tritical)</a>	TUnsharp is a basic sharpening filter that uses a couple different variations of unsharpmasking and allows for controlled sharpening based on edge magnitude and min/max neighborhood value clipping. The real reason for its existence is that it sports a gui with real time preview. <a href="#">[discussion]</a> .
<a href="#">Unfilter plugin (by Tom Barry)</a>	This filter softens/sharpens a clip. It implements horizontal and vertical filters designed to (slightly) reverse previous efforts at softening or edge enhancement that are common (but ugly) in DVD mastering. <a href="#">[discussion]</a> .
<a href="#">WarpSharp</a>	WarpSharp.
<a href="#">Xsharpen</a>	This filter performs a subtle but useful sharpening effect.

## 1.7 Resizers

*Plugins for resizing your clip.*

<a href="#">BicublinResize (by Marc FD)</a>	This is a set of resamplers: FastBilinear (similar to tbarry's simpleresize), FastBicubic (an unfiltered Bicubic resampler) and Bicublin (uses bicubic on Y plane and bilinear on UV planes). <a href="#">[discussion]</a> .
<a href="#">SimpleResize (by Tom Barry)</a>	Very simple and fast two tap linear interpolation. It is unfiltered which means it will not soften much.
<a href="#">YV12InterlacedReduceBy2 (by Tom Barry)</a>	InterlacedReduceBy2 is a fast Reduce By 2 filter, usefull as a very fast downsize of an interlaced clip. <a href="#">[discussion]</a> .

## 1.8 Subtitle (source) Plugins

*Plugins which let you import various subtitle formats (hard-coded).*

<a href="#">VSFilter (by Gabest)</a>	Lets you import various formats of subtitles, like *.sub, *.srt, *.ssa, *.ass, etc. <a href="#">[discussion]</a> .
--------------------------------------	--

## 1.9 MPEG Decoder (source) Plugins

*Plugins which let you import mpeg2 files (including hdtv transport files).*

<a href="#">DGDecode (by Donald Graft)</a>	A MPEG2Dec3 modification. Supports in addition MPEG-1 files, 4:2:2 input, and a lot of other things. See changelist for more info. Incompatible with the dvd2avi 1.xx versions and requires DGIndex. <a href="#">[discussion]</a> .
<a href="#">MPEG2Dec (by dividee and</a>	Mpeg2dec is a plugin which lets AviSynth import MPEG2 files.

## Avisynth 2.5 Selected External Plugin Reference

<a href="#">others</a> )	(outputs to YUY2)
<a href="#">MPEG2Dec3 (by Marc FD and others)</a>	A MPEG2Dec2.dll modification with deblocking and deringing. Note that the colorspace information of dvd2avi is ignored when using mpeg2dec. [ <a href="#">discussion</a> ].

### 1.10 Audio Decoder (source) Plugins

*Plugins which let you import audio files.*

<a href="#">MPASource (by Warpenterprises)</a>	A mp1/mp2/mp3 audio decoder plugin. [ <a href="#">discussion</a> ].
<a href="#">NicAudio (by Nic)</a>	Audio Plugins for MPEG Audio/AC3/DTS/LPCM. NicLPCMSource expects raw LPCM files or LPCM WAV files. However, at present it only supports 2-channel LPCM WAV files. [ <a href="#">discussion</a> ].

### 1.11 Plugins to compare video quality

<a href="#">SSIM (by Lefungus)</a>	Filter to compare video quality (similar as psnr, but using a different video quality metric). [ <a href="#">discussion</a> ].
<a href="#">VqmCalc (by Lefungus)</a>	Filter to compare video quality (similar as psnr, but using a different video quality metric). [ <a href="#">discussion</a> ].

### 1.12 Broadcast Video Plugins

*These are meant to take care of various problems which show up when over the air video is captured. Some help with luma/chroma separation; Others reduce interference problems or compensate for overscan.*

<a href="#">AutoCrop plugin (by CropsyX)</a>	Automatically crops black borders from a clip. [ <a href="#">discussion</a> ].
<a href="#">BorderControl (by Simon Walters)</a>	After capturing video you might want to crop your video to get rid of rubbish. BorderControl enables you to smear added borders instead of adding solid borders preventing artefacts between picture and border. [ <a href="#">discussion</a> ].
<a href="#">DeScratch (by Fizick)</a>	This plugin removes vertical scratches from films. [ <a href="#">discussion</a> ].
<a href="#">DeSpot (by Fizick)</a>	This filter is designed to remove temporal noise in the form of dots (spots) and streaks found in some videos. The filter is also useful for restoration (cleaning) of old telecined 8mm (and other) films from spots (from dust) and some stripes (scratches). [ <a href="#">discussion</a> ].
<a href="#">FillMargins (by Tom Barry)</a>	A similar filter as BorderControl. [ <a href="#">discussion</a> ].
<a href="#">Guava Comb (by Lindsey Dubb)</a>	This is a comb filter, meant to get rid of rainbows, dot crawl, and shimmering in stationary parts of an image. [ <a href="#">discussion</a> ].
<a href="#">Reinterpolate411 (by Tom Barry)</a>	It seems that even chroma pixels are just being duplicated in the MainConcept codec (NTSC). The new filter will help that by discarding the odd chroma pixels and recreating them as the average of the 2 adjacent even pixels. [ <a href="#">discussion</a> ].

## Avisynth 2.5 Selected External Plugin Reference

<a href="#">TComb (by tritical)</a>	TComb is a temporal comb filter (it reduces cross-luminance (rainbowing) and cross-chrominance (dot crawl) artifacts in static areas of the picture). It will ONLY work with NTSC material, and WILL NOT work with telecined material where the rainbowing/dotcrawl was introduced prior to the telecine process! In terms of what it does it is similar to guavacomb/dedot.
-------------------------------------	--

### 1.13 Misc Plugins

<a href="#">AddGrain (by Tom Barry)</a>	AddGrain generates film like grain or other effects (like rain) by adding random noise to a video clip. This noise may optionally be horizontally or vertically correlated to cause streaking.
<a href="#">AudioGraph (by Richard Ling, modified by Sh0dan)</a>	Displays the audio waveform on top of the video. [ <a href="#">discussion</a> ].
<a href="#">avsmon "AviSynth monitor" (by johann.Langhofer)</a>	This plugin enables you to preview the video during the conversion and to determine the exact audio delay. [ <a href="#">discussion</a> ].
<a href="#">Blockbuster (by Sansgrip)</a>	With this filter one can use several methods to reduce or eliminate DCT blocks: adding noise (Gaussian distributed), sharpening, or blurring. [ <a href="#">discussion</a> ].
<a href="#">ChromaShift (by Simon Walters)</a>	ChromaShift shifts the chrominance information in any direction, to compensate for incorrect Y/UV registration. [ <a href="#">discussion</a> ].
<a href="#">ColorMatrix (by Wilbert Dijkhof)</a>	ColorMatrix corrects the colors of MPEG-2 streams. More correctly, many MPEG-2 streams use slightly different coefficients (called Rec.709) for storing the color information than AviSynth's color conversion routines or the XviD/DivX decoders (called Rec.601) do, with the result that DivX/XviD clips or MPEG-2 clips encoded by TMPGEnc/QuEnc are displayed with slightly off colors. This can be checked by opening the MPEG-2 stream directly in VDubMod. [ <a href="#">discussion</a> ].
<a href="#">DctFilter (by Tom Barry)</a>	Reduces high frequency noise components using Discrete Cosine Transform and its inverse. Results in a high compressibility gain, when it is used at the end of your script. Height/width must be a multiple of 16. [ <a href="#">discussion</a> ].
<a href="#">DePan (by Fizick)</a>	DePan tools estimates global motion (pan) in frames, and makes full or partial global motion compensation. [ <a href="#">discussion</a> ].
<a href="#">Dup (by Donald Graft)</a>	This is intended for use in clips that have a significant number of duplicate content frames, but which differ due to noise. Typically anime has many such duplicates. By replacing noisy duplicates with exact duplicates, a bitrate reduction can be achieved. [ <a href="#">discussion</a> ].
<a href="#">DVinfo (by WarpEnterprises)</a>	This filter grabs the timestamp and recording date info out of a DV-AVI. It should work with Type-1 and Type-2, standard AVI and openDML. [ <a href="#">discussion</a> ].
<a href="#">ffavisynth (by Milan Cutka)</a>	A plugin which lets you directly use ffdshow image processing filters from AviSynth scripts. [ <a href="#">discussion</a> ].
<a href="#">GiCoCU (by E-Male)</a>	Reproduces photoshop's handling amp-files and gimp's handling of color curve files. [ <a href="#">discussion</a> ].

## Avisynth 2.5 Selected External Plugin Reference

<a href="#">MaskTools (by Kurosu and Manao)</a>	This plugin deals with the creation, the enhancement and the manipulating of such mask for each component of the YV12 colorspace. [ <a href="#">discussion</a> ].
<a href="#">MVTools (by Manao)</a>	Collection of filters (Blur, ConvertFPS, Denoise, Interpolate, Mask and others) which uses motion vectors generated by this plugin. [ <a href="#">discussion</a> ].
<a href="#">RawSource (by WarpEnterprises)</a>	This filter loads raw video data. [ <a href="#">discussion</a> ].
<a href="#">ReverseFieldDominance (by Simon Walters)</a>	Reverses the field dominance of PAL DV. [ <a href="#">discussion</a> ].
<a href="#">TMonitor (by tritical)</a>	TMonitor is a filter very similar to AVSMon. It enables monitoring of an AviSynth clip via previewing the video, viewing clip information (such as video width, height, colorspace, number of frames, audio samples, sample rate, number of audio channels, and more), and adjusting the audio delay. It also supports multiple instances per script, allowing viewing of differences between different parts of a processing chain.
<a href="#">Undot (by Tom Barry)</a>	UnDot is a simple median filter for removing dots, that is stray orphan pixels and mosquito noise. It clips each pixel value to stay within min and max of its eight surrounding neighbors. [ <a href="#">discussion</a> ].
<a href="#">VideoScope (by Randy French)</a>	You can use this plugin to graph the colors of a frame. It shows a waveform monitor (wfm) and a vectorscope. [ <a href="#">discussion</a> ].

\$Date: 2006/12/18 22:10:10 \$

# 2 AviSynth Frequently Asked Questions

## 2.1 Contents

1. [General information](#)
2. [Loading clips \(video, audio and images\) into AviSynth](#)
3. [Opening scripts in encoder and player applications](#)
4. [Some common error messages](#)
5. [Recognizing and processing different types of content](#)
6. [The color format YV12 and related processing and encoding issues](#)
7. [How to use VirtualDub's plugins in AviSynth](#)

### 2.1.1 General information

[This section](#) contains some general information about AviSynth. It's about its history, the different versions floating around. It explains how to get it to work, how to create a script and how to use plugins. It also discusses some command line utilities and graphical user interfaces which are able to open AviSynth scripts.

### 2.1.2 Loading clips (video, audio and images) into AviSynth

[This section](#) explains how to import the most common media formats (video, audio and images) into AviSynth. It explains which internal filters or external plugins can be used to import them and how this should be done. If this section doesn't provide enough information to load your media format, you should read the following document: [Importing media](#). It discusses many media formats, and gives a systematic way (using multiple ways) of loading these formats into AviSynth.

### 2.1.3 Opening scripts in encoder and player applications

[This section](#) describes frameserving and how to frameserve from and to AviSynth. It explains how to load AviSynth scripts in supported encoders and players, and what you can do if they don't support AviSynth scripts.

### 2.1.4 Some common error messages

[This section](#) describes some common error messages and explains what to do about them.

### 2.1.5 Recognizing and processing different types of content

[This section](#) explains what you should do when the video and audio in your final encoding is not in sync. It discusses variable framerate video. It explains how to recognize and process progressive, interlaced, telecined, hybrid and blended content.

## **2.1.6 The color format YV12 and related processing and encoding issues**

[This section](#) contains specific information about the color format YV12 and related processing and encoding issues.

## **2.1.7 How to use VirtualDub's plugins in AviSynth**

[This section](#) explains how to use plugins written for VirtualDub in AviSynth.

`$Date: 2009/09/12 20:57:20 $`

## 3 AviSynth FAQ – Some common error messages

### 3.1 Contents

1. [I got the message "LoadPlugin: unable to load "xxx" is not an AviSynth 1.0/AviSynth 2.0 plugin"?](#)
2. [When frameserving I got the following message: "Script error, there is no function named "xxx \(some filter\)"" ?](#)
3. [I installed AviSynth and get the following error message: "Couldn't locate decompressor for format 'YV12' \(unknown\)."?](#)
4. [When encoding I got the following error "ACM failed to suggest a compatible PCM format"?](#)
5. [When encoding I got the following error: "framesize xxx not supported"?](#)
6. [When frameserving I got the following message: "AVISource: couldn't locate a decompressor for fourcc \(...\) "?](#)
7. [When frameserving I got the following message: "DirectShowSource: Could not open as video or audio Video Returned: "DirectShowSource: the filter graph manager won't talk to me" "?](#)
8. [I am trying to load a script that has a path name with a mix of japanese characters \(someone's name\) and Ascii test. I got an import error and the path that is displayed has some strange characters \(not the japanese characters\)?](#)
9. [When frameserving I got the following message: " CAVIStreamSynth: System exception – Access Violation at 0x0, reading from 0x0"?](#)
10. [When frameserving I got a message similar to: "Avisynth open failure: Script error: Invalid arguments to function "xxx \(some filter\)" \(L:\Video.avs, line 5\)"](#)

#### 3.1.1 I got the message "LoadPlugin: unable to load "xxx (some plugin)" is not an AviSynth 1.0/AviSynth 2.0 plugin"?

You are using a plugin which is not compatible with that version of AviSynth. As explained [here](#), plugins compiled for AviSynth v2.5 are not compatible with AviSynth v1.0x/v2.0x and vice versa.

#### 3.1.2 When frameserving I got the following message: "Script error, there is no function named "xxx (some filter)"" ?

You probably installed/registered a version of AviSynth which doesn't contain that specific filter. Make sure that there are no other versions floating around on your hard disk (there's a possibility that a version will be registered while it is not in your system directory). Make sure that the latest stable version is registered, see also [here](#).

#### 3.1.3 I installed AviSynth and got the following error message: "Couldn't locate decompressor for format 'YV12' (unknown)."?

Install a codec which supports YV12. DivX5 or one of the recent [XviD builds of Koepi](#) or [Helix YUV codec](#) or some other (ffvfw, ffdshow). If that still doesn't work, modify your registry as explained in the next question.



### 3.1.4 When encoding I got the following error "ACM failed to suggest a compatible PCM format"?

This error means that you are using AviSource to open your AVI file, but you don't have an ACM codec to decode the audio stream. The most common problem is that your audio is an AC3 or a MP3 stream, but you don't have the corresponding ACM codec installed. It can also happen that your audio is a "crippled" (that is, with an incorrect header) MP3 [\[1\]](#) [\[2\]](#) [\[3\]](#).

There are several solutions for this problem:

- Get the proper ACM codec. You can use [GSpot](#) to find out which audio stream you are dealing with. The needed ACM codecs can be found [here](#).
- Demux your audio with an application like [AVI-Mux GUI](#) or [VirtualDub](#), and import your audio with a specific plugin. See also [here](#).
- Use [DirectShowSource](#) to open the audio. Install [ffdshow](#) and enable the specific audio format in the "Audio Decoder Properties" tab. Create the following script:

```
Vid = AviSource("Blah.avi", audio=false)
Aud = DirectShowSource("Blah.avi", video=false)
AudioDub(Vid, Aud)
```

### 3.1.5 When encoding I got the following error: "framesize xyz x 56 not supported"?

This usually is an indicator of a script error, where the input is actually the error message being displayed. Here, xyz is the length of the error message text and 56 is the height (xyz will vary depending on the error message whilst the height will always be 56 pixels). Your encoder is seeing the error message as an RGB32 input source and hence the error. Opening the script with WMP or VirtualDub should display the error message. Fix the error in the script and retry to encode.

### 3.1.6 When frameserving I got the following message: "AVISource: couldn't locate a decompressor for fourcc (...)?"

Usually, this error message shows up if you don't have the right Vfw codec installed for decoding your video. Get [Gspot](#) to find out which codec you need. Get, for example, [XviD](#) for your MPEG-4 ASP clips and [Cedocida codec](#) for your DV clips. If you have problems finding the right one, ask around on the video forums.

But it can also show up if you call [AviSource](#) too many times. The dll for the decompression codec is loaded separately for every AviSource call. Eventually an OS-imposed limit is reached, the codec can't be loaded and you get that error message. More discussion can be found [here](#). A good solution is to use a number of scripts (keeping each below the problematic limit of avi calls) and encode them separately, and join them afterwards in some application.

### 3.1.7 When frameserving I got the following message: "DirectShowSource: Could not open as video or audio Video Returned: "DirectShowSource: the filter graph manager won't talk to me""?

This is a common error that occurs when DirectShow isn't able to deliver any format that is readable to AviSynth. Try creating a filter graph manually. See if you are able to construct a filter graph that delivers any output that AviSynth can open. If not, you might need to download additional DirectShow filters that can deliver correct material. If you can play the graph in [GraphEdit](#), make sure to remove the video and audio renderers, before saving the graph and opening it in AviSynth. Some examples can be found [here](#).

### 3.1.8 I am trying to load a script that has a path name with a mix of japanese characters (someone's name) and Ascii test. I got an import error and the path that is displayed has some strange characters (not the japanese characters)?

AviSynth has problems with non-ANSI chars on filenames. It only supports [8 bit character ANSI text](#). Some discussion about this: [\[4\]](#) and [\[5\]](#).

### 3.1.9 When frameserving I got the following message: "CAVISTreamSynth: System exception – Access Violation at 0x0, reading from 0x0"?

Access Violation at 0x0, reading from 0x0 is usually caused by running out of memory (memory leak ???). It can be caused by a plugin which is leaking memory, but apparently it can also be caused by other things (codecs, applications ???) [\[1\]](#) [\[2\]](#). Add SetMemoryMax(...) at the beginning of the script. If that doesn't help, report the issue in the doom9 forums, and we will try to help finding the cause of it.

### 3.1.10 When frameserving I got a message similar to: "Avisynth open failure: Script error: Invalid arguments to function "xxx (some filter)" (I:\Video.avs, line 5)"

It means you are passing incorrect arguments (that is of the correct type) to your script, filter or plugin. For example:

```
# passing a float (2.0), while Loop expects an int:
Loop(clip, 2.0)

# passing three clips to Overlay instead of two:
AviSource("anime_raw.avi")
karaoke = AviSource("karaoke.avi")
Trim(0,999) + Trim(1000,1030).Overlay(last, karaoke, mask=sign.ShowAlpha()) + Trim(1031,0)
# last should be omitted as argument to Overlay
```

So make you the passed arguments are of the correct type and read the corresponding documentation if necessary.

## Avisynth 2.5 Selected External Plugin Reference

[| Main Page](#) | [| General Info](#) | [| Loading Clips](#) | [| Loading Scripts](#) | **Common Error Messages** | [| Processing Different Content](#) | [| Dealing with YV12](#) | [| Processing with Virtualdub Plugins](#) |

\$Date: 2008/07/04 17:58:20 \$

# 4 AviSynth FAQ – Recognizing and processing different types of content

## 4.1 Contents

1. [The video and audio in my final encoding is out of sync, what should I do?](#)
2. [How do I recognize progressive, interlaced, telecined, hybrid and blended content?](#)
3. [How do I process interlaced content?](#)
4. [How do I process telecined content?](#)
5. [How do I process hybrid content?](#)
6. [What is variable framerate video?](#)
7. [How do I import variable framerate video into AviSynth and how do I process it?](#)

### 4.1.1 The video and audio in my final encoding is out of sync, what should I do?

Assuming that you processed your video and or audio with AviSynth, there can be several reasons why your final encoding is not in sync (synchronization). The most common ones are:

- 1) Your source is already out of sync (thus before any AviSynth processing or any encoding). It's a pain to correct this, but that's not the scope of this FAQ.
- 2) The audio has a constant delay, and you forgot to add the delay (either in AviSynth if you imported the audio in AviSynth or in an encoder if you imported the audio directly in your encoder). As an example, the demuxed audio stream from a VOB has often a delay. When demuxing this audio stream with DGIndex, the delay (actually how the delay should be corrected) is written into the name of the demuxed audio stream. You can use [DelayAudio](#) to add the delay in AviSynth.

```
vid = MPEG2Source("D:\movie.d2v")
aud = NicAC3Source("D:\movie T01 2_0ch 448Kbps DELAY -218ms.ac3")
AudioDub(vid, aud)
DelayAudio(-0.218)
```

- 3) The audio has a variable delay (with a zero delay at the beginning and a maximal delay at the end). This can be caused when you load a clip into AviSynth which has a variable framerate. Pretty much anything except video contained in an AVI or MPEG-2/VOB file can be variable framerate. If you used [DirectShowSource](#) the load your clip, you can use

```
# a mkv-file is used here as an example:
DirectShowSource("D:\movie.mkv", fps=xxx, convertfps=true) # fps = average framerate
```

to ensure sync. What happens is that frames are added or removed to ensure sync, thus converting it to a constant framerate video.

If you are not using DirectShowSource or you don't want to add or remove frames, you need to create a timecodes file first and use it later on in your final encoding. Have a look at [this article](#) for more information on this subject.

## 4.1.2 How do I recognize progressive, interlaced, telecined, hybrid and blended content?

It is important to know your content if you want to process it. The most important ones are: progressive, interlaced, telecined, hybrid and blended content, and they should be processed differently.

- Progressive and interlaced content:  
Most filters assume that your content is progressive (which means that every frame is taken at a different time–instant), unless the filter has an option `interlaced=true/false`. When the option is present you can use `interlaced=true` for interlaced content. For interlaced content, every field (a frame consists of two fields) is taken at a different time–instant. This is explained in the [Analog Capture Guide](#) and the [Force Film, IVTC, and Deinterlacing](#) tutorial.
- Telecined content:  
Usually movies are shot at 24 fps (frames per second). When putting this on a dvd, fields are added to get the required frame rate of 30 fps (well, it's actually 29.97 fps, but that's not important here). When doing this, the content is called "telecined content" (this holds for the conversions 25 fps → 30 fps and 24 fps → 25 fps as well, provided fields are added). More about this can be found in the [Force Film, IVTC, and Deinterlacing](#) tutorial.
- Hybrid content:  
Hybrid content is content with different base frame rates (for example 8, 12, and 16 fps at which anime is often drawn). Start Trek is a different example consisting of telecined (at 30 fps) and interlaced content (at 30 fps). Have a look at [this article](#) for more information on this subject.
- Blended content:  
Blended content is content which consists of blended fields (in some fields there is content from different time–instants visible). It's usually the result of bad NTSC to PAL conversions (and vice-versa), or messed-up deinterlacing. Some examples can be found [here](#) or [here](#).

## 4.1.3 How do I process interlaced content?

There are two ways to process your interlaced content (assuming that you use a filter which has no `interlaced=true` option). The first one is the most accurate, but also the slowest: bobbing, processing and reinterlacing. The second one is the fastest, but also less accurate one: processing the fields separately.

1) bobbing:

```
AssumeTFF() # or AssumeBFF (set the video's field order correctly)
TDeint(mode=1, type=3) # or any other smart Bob
Filter(...)
AssumeTFF() # or AssumeBFF (set the video's field order correctly)
Separatefields()
Selectevery(4,0,3)
Weave()
```

2) processing the fields separately:

```
SeparateFields()
even = SelectEven(last).Filter(...)
odd = SelectOdd(last).Filter(...)
Interleave(even, odd)
Weave()
```

#### 4.1.4 How do I process telecined content?

You need to inverse telecine (IVTC) before you do any processing. You can use the plugin Decomb for example, which can be downloaded [here](#). See the tutorials "[Force Film, IVTC, and Deinterlacing – what is DVD2AVI trying to tell you and what can you do about it.](#)" or "[the analog capture guide](#)" which explain how to do this.

#### 4.1.5 How do I process hybrid content?

You only run into troubles when your clip as openend in AviSynth shows combing (being partly interlaced, telecined, etc ...). I'm not sure yet what to do in that case.

#### 4.1.6 What is variable framerate video?

There are two kinds of video when considering framerate, constant framerate (cfr) video and variable framerate (vfr) video. For cfr video the frames have a constant duration, and for vfr video the frames have a non-constant duration. Many editing programs (including VirtualDub and AviSynth) assume that the video is cfr, partly because avi doesn't support vfr. Although the avi container doesn't support vfr, there are several containers (mkv, mp4 and wmv/asf for example) which do support vfr. More information can be found [here](#).

#### 4.1.7 How do I import variable framerate video into AviSynth and how do I process it?

There are two ways to import variable framerate video into AviSynth:

1. Open the video in AviSynth using for example DirectShowSource(..., convertfps=false) or FFmpegSource. The problem is that in those cases no frames are added or removed to convert it to constant framerate video to ensure sync.  
Generate a timecode file using some external program or using the AviSynth plugin you use for importing the video into AviSynth (if possible). Many non-AVI files contain video with a variable framerate, and in that case you need to make sure of the following two things:
  1. *Don't change the framerate and the number of frames in AviSynth.* If you don't this (and you don't change the timecodes file manually) your video and audio in your final encoding will be out of sync.
  2. *Use the timecodes file again when muxing your encoded video and audio.* If you don't do this your video and audio in your final encoding will be out of sync.
2. Open the video in AviSynth using for example DirectShowSource(..., convertfps=true). In this case frames are added or removed to convert it to constant framerate video to ensure sync. You can process the video the way you want. You can even create a new timecodes file and create a new variable framerate video using it. More information can be found [here](#).

Regarding the first way. If you did change the framerate or the number of frames, you can use DeDup to recreate a new timecode file:

```
dedup_dedup(threshold=0.1, maxcopies=10, maxdrops=4, log="01.log", timesin="original.tmc", time
```

## Avisynth 2.5 Selected External Plugin Reference

The parameter "timesin" specifies the timecode file of the original video on which the output file will be based on (rather than just using the input stream's framerate). I never used it, so I'm not sure how good this is. Look [here](#) for a discussion.

| [Main Page](#) | [General Info](#) | [Loading Clips](#) | [Loading Scripts](#) | [Common Error Messages](#) | **Processing Different Content** | [Dealing with YV12](#) | [Processing with Virtualdub Plugins](#) |

\$Date: 2008/07/02 18:57:42 \$

# 5 AviSynth FAQ – Opening scripts in encoder and player applications

## 5.1 Contents

1. [What is frameserving and what is it good for?](#)
2. [How do I use AviSynth as a frameserver?](#)
3. [How do I frameserve my AVS-file to encoder/application X?](#)
  - ◆ [Direct frameserving to compatible applications](#)
  - ◆ [Direct frameserving to applications using additional plugins](#)
  - ◆ [Direct frameserving to special or modified versions of encoders](#)
  - ◆ [Frameserving to applications via fake AVI files and proxy utilities](#)
  - ◆ [Frameserving via pipe from auxiliary programs to application-encoders](#)
4. [How do I solve problems when opening/reading scripts in encoders and players?](#)
5. [How do I frameserve from Premiere/Ulead/Vegas to AviSynth?](#)

### 5.1.1 What is frameserving and what is it good for?

An excellent description is found on [Lukes homepage](#):

"Frameserving is a process by which you directly transfer video data from one program on your computer to another. No intermediate or temporary files are created. The program that opens the source file(s) and outputs the video data is called the frameserver. The program that receives the data could be any type of video application.

There are two main reasons that you would want to frameserve a video:

1. **Save Disk Space:** Depending on the frameserving application, you can usually edit/process your video as it is being frameserved. Because frameserving produces no intermediate files, you can use a frameserver to alter your videos without requiring any additional disk space. For example, if you wanted to join two video files, resize them, and feed them to another video application, frameserving would allow you to do this without creating a large intermediate file.
2. **Increased Compatibility:** To the video application that's receiving the frameserved video, the input looks like a relatively small, uncompressed video file. However, the source file that the frameserver is transferring could actually be, for example, a highly compressed MPEG-1 video. If your video application doesn't support MPEG-1 files, it's not a problem because the application is just receiving standard uncompressed video from the frameserver. This feature of frameserving enables you to open certain types of files in an application that wouldn't normally support them.

Furthermore, because the video application is being fed the source video one frame at a time, it doesn't know anything about the file size of the source video. Therefore, if your application has 2 GB or 4 GB limit on input file size, it won't have any effect on your frameserved video. You could feed 100 GB of video via a frameserver to an application limited to 2 GB and it wouldn't cause a problem."

### 5.1.2 How do I use AviSynth as a frameserver?

Write a script using a text editor. Load your clip in AviSynth (see [FAQ loading clips](#)), do the necessary



## Avisynth 2.5 Selected External Plugin Reference

filtering and load the AVS–file in encoder/application X (must be an encoder or application which can read AVI–files (see also [here](#)).

### 5.1.3 How do I frameserve my AVS–file to encoder/application X?

There is simple way for many applications, and tricky ways for many others.

#### 5.1.3.1 Direct frameserving to compatible applications

Simply open your AVS file in coder/application with menu, command line or drag–and–drop AVS file to it (working ways are dependent on the application). Some programs have "AviSynth \*.avs" in "Open" menu, for others try select "All files \*.\*" or type AVS file name instead of "AVI".

Players: Media Player Classic, Windows Media Player 6.4, 9 and others.

Encoders: QuEnc, Mencoder, HC Encoder, CCE SP 2.50 and 2.66, Canopus ProCoder 1.5 and above, MainConcept MPEG Encoder, TMPGEnc, TMPGEncXpress 3/4, Elecard Converter Studio, xvid\_encraw, FFMpeg (new versions), Nero 6, Nero 7 (drag–and–drop only) and others.

Editors: VirtualDub, AviDemux (through its avs proxy option)

But some applications work fine only with some specific video or audio formats, have a look at the next section.

#### 5.1.3.2 Direct frameserving to applications using additional plugins

- For frameserving to Premiere there exists an import plugin "IM–Avisynth.prm".

The original version was located at [Bens site](#), see [mirror](#). A much improved version can be downloaded from the [Video Editors Kit sourceforge page](#). This works for Premiere 5.x, 6.x and Pro at present. Version 1.5 also works for Premier CS3. To install the import plugin move the IM–Avisynth.prm file into your Premiere "Plug–ins" directory.

#### 5.1.3.3 Direct frameserving to special or modified versions of encoders

Some programs initially could not open AviSynth scripts, but updated or alternative programs can do it:

- Mencoder
- FFMpeg: Versions older then SVN–r6129 use the "AVSredirect.dll" for communication with Avisynth. From SVN–r6129 up, AVS redirect code is integrated in the FFMpeg executable (as option at compiling). Use builds at <http://ffdshow.faireal.net/mirror/ffmpeg/>
- Windows Media 9 Encoder: Download Nic's Windows Media 9 Encoder and make sure you also installed the Windows Media 9 codec. Both can be found [here](#).

#### 5.1.3.4 Frameserving to applications via fake AVI files and proxy utilities

Many "new" programs do NOT use the Windows functions to read the AVI-files. If they use own read functions the AviSynth-script files will not work. There are utilities that can create small fake AVI file with special type (FOURCC), and provide correspondent system codec to "decode" these dummy compressed files.

Select your AVS file in utility menu, set options and create fake AVI file with some name. Then you can open this fake AVI in your application/encoder, that will be receive frames from the codec that will be receive frames from AviSynth.

Several such utilities are different by supported modes (formats) of output video (with or without conversion) and audio (unpacked audio is most compatible but filesize is larger), by user interface (window, command line) and number of bugs.

- [VFAPI reader codec](#) with DGVfapi (as a client) from [DGDMPGDec](#).

Features – output RGB24 only, unpacked audio, multiple files support, good compatibility, but a bit slow.

- MakeAVIS is included in ffvfw and [FFDShow](#).

Features – output to any color format. Uncompressed audio works properly in old ffvfw and recent (13 november 2007) ffdshow (8 and 16 bit only, use [ConvertAudioTo16bit](#) when necessary).

- [Proxy-codec AVS2AVI](#). (Note that the same-name utility by Moitah and others is an encoder and not an AVI-wrapper.)

Features – video output same as input format, no audio.

Known programs that will not open AVS scrips without these utilities:

CCE SP v2.62–2.64, Windows Media Encoder vx.x. (older than v9), Ulead VideoStudio 5–11, MediaStudio 6–8, Pinnacle Studio, Sony Vegas, Nero 8, ImageMixer and others.

#### 5.1.3.5 Frameserving via pipe from auxiliary programs to application-encoders

[Avs2YUV](#) is a command-line program, intended for use under Wine, to interface between AviSynth and Linux-based video tools.

Programs: Mpeg2enc, Mencoder, FFMpeg.

avs2yuv out.avs -o - | mpeg2enc - options...

This way is obsolete since these programs have native AviSynth support now.

#### 5.1.4 How do I solve problems when opening/reading scripts in encoders and players?

1. TMPGEnc doesn't read my AVS files (this happens in old versions of TMPGEnc), what to do?
  - ◆ Install the VFAPI plugin for TMPGEnc.

## Avisynth 2.5 Selected External Plugin Reference

- ◆ Disable the direct show filters within TMPGEnc and turn off the VirtualDub proxy before frameserving.
  - ◆ Add "[ConvertToRGB24](#)" at the end of your AVS-file.
  - ◆ Install [Huffyuv/DivX](#) codec so that it can do the decompression for you when loading an AVI in TMPGEnc.
  - ◆ Install the [ReadAVS plugin](#). Just copy ReadAVS.dll to the VFAPI reader directory and open the reg-file ReadAVS.reg in notepad and change the corresponding path. Save it, and doubleclick on it to merge it with your registry-file.
2. CCE SP crashes when reading an AVS-file, what to do?
- ◆ If you're using Win2k then run CCE in WinNT4-SP5 compatibility mode.
  - ◆ Put addaudio.avsi in your AviSynth plugin folder and add "[AddAudio\(44100\)](#)" in your script, if you don't have any audio in your AVS-file.
  - ◆ Some versions (like CCE SP v2.62/v2.64) don't read AVS files. Get CCE SP v2.66 or a more recent version.
3. My encoder or player doesn't open AviSynth scripts, what should I do?
- ◆ In this case you may try other way, for example an AVI wrapper, like [vfapi](#) or [makeAVIS](#).
4. When opening my clip in an encoder or player, the colors are messed up, what to do?
- ◆ If you have such problems, some external (or internal) codec is messing up the used colorspace conversion. If you have such problems add "[ConvertToRGB24](#)" as the last line of your script (for ProCoder and CCE use [ConvertToYUY2\(interlaced=true\)](#) or [=false](#)) and have a look at the thread (and the suggested solutions) [colorspace conversion errors](#).
5. Windows Media Encoder 9 Series does not open AVS files, what to do?
- ◆ Use an [updated WMCmd.vbs script](#) [[discussion about the fix](#)].
  - ◆ In order to use AviSynth source with WME9, you need to set the encoder source to "Both device and file" in the Session Properties, see [discussion](#) and [WMV faq](#). Or use [Nic's WMV encoder](#).
6. WMP11 on Vista dos no play AVS, what to do?
- ◆ You may [edit registry](#) to add .avs as known extension. Copy the registry key (and subkeys) for  
**HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Multimedia\WMPlayer\Extensions\avi**  
to  
**HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Multimedia\WMPlayer\Extensions\avs**
  - ◆ On Vista x64 you have to copy the correct 32-Bit nodes:  
**HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Multimedia\WMPlayer**  
to  
**HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Multimedia\WMPlayer**
  - ◆ Do not change anything at the registry if you are not experienced!

### 5.1.5 How do I frameserve from Premiere/Ulead/Vegas to AviSynth?

Install the AviSynth compatible frameserver [PluginPace frameserver \(by Satish Kumar\)](#) for frameserving from SonicFoundry Vegas (and earlier Vegas Video/VideoFactory versions), Adobe Premiere, Ulead MediaStudio Pro or Wax to AviSynth ([discussion](#)).

[| Main Page](#) | [| General Info](#) | [| Loading Clips](#) | [| Loading Scripts](#) | [| Common Error Messages](#) | [| Processing Different Content](#) | [| Dealing with YV12](#) | [| Processing with Virtualdub Plugins](#) |

\$Date: 2009/09/12 20:57:20 \$

# 6 AviSynth FAQ – General information

## 6.1 Contents

1. [What is AviSynth?](#)
2. [Who is developing AviSynth?](#)
3. [Where can I download the latest versions of AviSynth?](#)
4. [What are the main bugs in these versions?](#)
5. [Where can I find documentation about AviSynth?](#)
6. [How do I install/uninstall AviSynth?](#)
7. [What is the main difference between v1.0x, v2.0x, v2.5x, v2.6x and v3.x?](#)
8. [How do I know which version number of AviSynth I have?](#)
9. [How do I make an AVS-file?](#)
10. [Where do I save my AVS-file?](#)
11. [Are plugins compiled for v2.5x/v2.6x compatible with v1.0x/v2.0x and vice versa?](#)
12. [How do I use a plugin compiled for v2.0x in v2.5x?](#)
13. [How do I switch between different AviSynth versions without re-install?](#)
14. [VirtualdubMod, WMP6.4, CCE and other programs crash every time on exit \(when previewing an avs file\)?](#)
15. [My computer seems to crash at random during a second pass in any encoder?](#)
16. [Is there a command line utility for encoding to DivX/XviD using AviSynth?](#)
17. [Does AviSynth have a GUI \(graphical user interface\)?](#)

### 6.1.1 What is AviSynth?

AviSynth (AVI SYNTHesizer) is a frameserver. An excellent description is given on [Lukes homepage](#):

"AviSynth is a very useful utility created by Ben Rudiak-Gould. It provides many options for joining and filtering videos. What makes AviSynth unique is the fact that it is not a stand-alone program that produces output files. Instead, AviSynth acts as the "middle man" between your videos and video applications.

Basically, AviSynth works like this: First, you create a simple text document with special commands, called a script. These commands make references to one or more videos and the filters you wish to run on them. Then, you run a video application, such as Virtualdub, and open the script file. This is when AviSynth takes action. It opens the videos you referenced in the script, runs the specified filters, and feeds the output to video application. The application, however, is not aware that AviSynth is working in the background. Instead, the application thinks that it is directly opening a filtered AVI file that resides on your hard drive.

There are five main reasons why you would want to use AviSynth:

1. Join Videos: AviSynth lets you join together any number of videos, including segmented AVIs. You can even selectively join certain portions of a video or dub soundtracks.
2. Filter Videos: Many video processing filters are built in to AviSynth. For example, filters for resizing, cropping, and sharpening your videos.
3. Break the 2 GB Barrier: AviSynth feeds a video to a program rather than letting the program directly open the video itself. Because of this, you can usually use AviSynth to open files larger than 2 GB in programs that don't natively support files of that size.
4. Open Unsupported Formats: AviSynth can open almost any type of video, including MPEGs and

## Avisynth 2.5 Selected External Plugin Reference

certain Quicktime MOVs. However, when AviSynth feeds video to a program, it looks just like a standard AVI to that program. This allows you to open certain video formats in programs that normally wouldn't support them.

5. Save Disk Space: AviSynth generates the video that it feeds to a program on the fly. Therefore, no temporary or intermediate videos are created. Because of this, you save disk space."

### 6.1.2 Who is developing AviSynth?

Originally AviSynth (up to v1.0b) was developed by Ben Rudiak-Gould. See [mirror of his homepage](#). Currently it is developed by Sh0dan, IanB, d'Oursse (AviSynth v3), Bidoche (AviSynth v3) and [others](#).

### 6.1.3 Where can I download the latest versions of AviSynth?

The most recent stable version is v2.57, which can be found [here](#) (just as more recent builds).

### 6.1.4 What are the main bugs in these versions?

Current bugs can be found in the documentation on the [AviSynth project page](#). Fixed bugs can be found in the [Changelog](#).

### 6.1.5 Where can I find documentation about AviSynth?

Documentation about the filters of AviSynth can be found on this site [Main Page](#), and in particular here: [Internal filters](#). You should read these documents before posting to the forum, but it's OK to post if you have trouble understanding them.

### 6.1.6 How do I install/uninstall AviSynth?

Starting from v2.06 AviSynth comes with an auto installer. Also make sure you have no other versions of AviSynth floating around on your harddisk, because there is a chance that one of those versions will be registered. Remove them if necessary. For uninstalling AviSynth go to "program", "AviSynth 2.5" and select "Uninstall AviSynth".

Installing AviSynth v2.05 or older versions: move avisynth.dll to your system/system32 directory and run install.reg. For uninstalling run uninstall.reg and delete avisynth.dll.

### 6.1.7 What is the main difference between v1.0x, v2.0x, v2.5x, v2.6x and v3.x?

The versions v1.0x and v2.0x are compatible and outdated. The main difference with v2.5x is that the internal structure of AviSynth has changed (YV12 and multichannel support) with the consequence that external plugins compiled for v1.0x/v2.0x will not work for v2.5x/v2.6x and vice versa. In v2.6x other planar formats like YV24 and Y8 are added. v2.5x plugins will work in v2.6x but not vice-versa. All versions are incompatible with v3.x, which will also work under Linux/MacOSX (see [AviSynth v3](#)) and rely on the GStreamer API.

### 6.1.8 How do I know which version number of AviSynth I have?

Open a text-editor, for example notepad. Add the following line

```
Version()
```

and save the file with the extension "avs". Save for example as "version.avs" (make sure that the extension is "avs" and not "txt"). Open the file in an application which can read AVI-files, for example WMP 6.4 or Media Player Classic. The version number will be displayed.

### 6.1.9 How do I make an AVS-file?

Use your preferred text editor (e.g. Notepad). See also [this](#).

Although AviSynth doesn't need them, there are several GUIs (graphical user interface) which may help you writing your AVS files. You can read a description for each one of them [here](#).

### 6.1.10 Where do I save my AVS-file?

Anywhere on your hard-disk.

### 6.1.11 Are plugins compiled for v2.5x/v2.6x compatible with v1.0x/v2.0x and vice versa?

As explained [here](#) that is not the case. However it is possible to use a v1.0x/v2.0x plugin in v2.5x/v2.6x, as explained [here](#).

### 6.1.12 How do I use a plugin compiled for v2.0x in v2.5x?

In plugin collection [warpssharp\\_2003\\_1103.cab](#) you will find a plugin called "LoadPluginEx.dll". (When using an older version of LoadPluginEx.dll, don't move this plugin to your plugin dir. But move it to a separate folder, otherwise VirtualDubMod and WMP6.4 will crash on exit.) This will enable you using v2.0x plugins in v2.5x. An example script (using the v2.0x plugin Dust by Steady):

```
LoadPlugin("C:\Program Files\avisynth2_temp\plugins\LoadPluginEx.dll")
LoadPlugin("C:\Program Files\avisynth2_temp\plugins\dustv5.dll")
```

```
AviSource("D:\clip.avi")
ConvertToYUY2()
PixieDust(5)
```

If you want to automate this process, have a look at [LoadOldPlugins](#).

### 6.1.13 How do I switch between different AviSynth versions without re-install?

## Avisynth 2.5 Selected External Plugin Reference

- You can use AvisynthSwitcher available [here](#). Versions v2.08 and v2.50 are provided, but you can easily add a new one under AvisynthSwitcher\versions\Avisynth 2.x.x.
- Some other ways are described [here](#).

### 6.1.14 VirtualDubMod, WMP6.4, CCE and other programs crash every time on exit (when previewing an avs file)?

This problem can be caused by certain plugins in your (autoloading) plugin folder. The solution is to move the problematic plugins outside your plugin folder and load them manually.

### 6.1.15 My computer seems to crash at random during a second pass in any encoder?

AviSynth is highly optimized. As a consequence it is possible that your computer seems to crash at random during a second pass. Try running the [Prime95](#) stress test for an hour, to check if your system is stable. If this test fails (or your computer locks up) make sure that your computer is not overclocked and lower your bus speed of your processor in steps of (say) five MHz till the crashes are gone.

### 6.1.16 Is there a command line utility for encoding to DivX/XviD using AviSynth?

- There is a command line utility called [AVS2AVI](#) (and AVS2AVI GUI) for encoding to DivX / XviD using AviSynth. [[discussion](#)]
- [xvid\\_encraw](#) for encoding to XviD in M4V. Use [mp4box](#) or [YAMB](#) to mux it into MP4.

### 6.1.17 Does AviSynth have a GUI (graphical user interface)?

AviSynth doesn't have a full fledged gui, but several tools are available:

- [VirtualDubMod](#): The following AviSynth related utilities are present:
  - ◆ 'Open via AVISynth' command: This allows you to open any AviSynth compatible video file by automatically generating a suitable script by a selectable template.
  - ◆ AVS Editor (Hotkey Ctrl+E): Just open your AVS and under tools select "script editor". Change something and press F5 to preview the video.
- AvisynthEditor: This is an advanced AviSynth script editor featuring syntax highlighting, auto-complete code and per version plugin definition files. [Here is a screenshot](#). It can be found [here](#). Discussion can be found on [Doom9.org forum](#).
- [AVSGenie](#): AVSGenie allows the user to select a filter from a drop down list or from a popup menu. An editable page of parameters will then be brought into view, with a guide to the filter and it's parameters. A video preview window opens, showing "source" and "target" views. The source window, in simple cases, shows output of the first line of the script, generally an opened video file. The target window shows the output of the whole script. In this way, effects of filters can easily be seen. The line which represents the source window can be changed. Discussion can be found [here](#).
- [SwiftAVS \(by Snollygoster\)](#): Another nice gui, formerly known as AviSynthesizer. [[discussion](#)]

## Avisynth 2.5 Selected External Plugin Reference

- [AvsP](#): It's a tabbed script editor for Avisynth. It has many features common to programming editors, such as syntax highlighting, autocompletion, call tips. It also has an integrated video preview, which when coupled with tabs for each script make video comparisons a snap. What really makes AvsP unique is the ability to create graphical sliders and other elements for any filter's arguments, essentially giving Avisynth a gui without losing any of its powerful features. Discussion can be found [here](#).

[|Main Page](#) | **General Info** | [Loading Clips](#) | [Loading Scripts](#) | [Common Error Messages](#) | [Processing Different Content](#) | [Dealing with YV12](#) | [Processing with Virtualdub Plugins](#) |

\$Date: 2008/10/26 14:18:53 \$



# 7 AviSynth FAQ – Loading clips (video, audio and images) into AviSynth

## 7.1 Contents

1. [Which media formats can be loaded in AviSynth?](#)
2. [Which possibilities exist to load my clip into AviSynth?](#)
3. [What are the advantages and disadvantages of using DirectShowSource to load your media files?](#)
4. [Has AviSynth a direct stream copy mode like VirtualDub?](#)
5. [How do I load AVI files into AviSynth?](#)
6. [Can I load video with audio from AVI into AviSynth?](#)
7. [How do I load MPEG-1/MPEG-2/DVD VOB/TS/PVA into AviSynth?](#)
8. [How do I load QuickTime files into AviSynth?](#)
9. [How do I load raw source video files into AviSynth?](#)
10. [How do I load RealMedia files into AviSynth?](#)
11. [How do I load Windows Media Video files into AviSynth?](#)
12. [How do I load MP4/MKV/M2TS/EVO into AviSynth?](#)
13. [How do I load WAVE PCM files into AviSynth?](#)
14. [How do I load MP1/MP2/MP3/MPA/AC3/DTS/LPCM into AviSynth?](#)
15. [How do I load aac/flac/ogg files into AviSynth?](#)
16. [How do I load pictures into AviSynth?](#)

### 7.1.1 Which media formats can be loaded into AviSynth?

Most video/audio formats can be loaded into AviSynth, but there are some exceptions like flv4 (VP6) and dvr-ms. If it is not possible to load a clip into AviSynth, you will have to convert it into some other format which can be loaded. Remember to choose a format for which you will have a minimal downgrade in quality as a result of the conversion.

### 7.1.2 Which possibilities exist to load my clip into AviSynth?

In general there are two ways to load your video into AviSynth:

1. using an AviSynth internal filter or plugin which is designed to open some specific format.
2. using the [DirectShowSource](#) plugin.

Make sure that your clip contains maximal one video and/or one audio stream (thus remove the subtitles and remove other video/audio streams). If you want to load a clip which contains both video and audio, you have two options:

- Demux the audio stream and load the streams separately in AviSynth.
- Try to load the clip in AviSynth. This might or might not work. For AVIs, make sure you have a good AVI splitter installed, e.g. [Gabest's AVI splitter](#). (Yes, Windows comes with an own AVI splitter, which will work in most cases.)

When loading a clip into AviSynth it is advised to follow the following guidelines:

## Avisynth 2.5 Selected External Plugin Reference

- When it is possible to load your clip into AviSynth using either AviSource or a specific plugin then do so, since this is more reliable than the alternatives which are listed below.
- If the above fails, load your clip using the DirectShowSource plugin.
- If the above fails, convert your clip into a different format (into one which is supported by AviSynth).

### 7.1.3 What are the advantages and disadvantages of using DirectShowSource to load your media files?

*advantages of DirectShowSource:*

- Many video and audio formats are supported through DirectShowSource (have a look at ffdshow for example).

*disadvantages of DirectShowSource:*

- It's less reliable than AviSource and specific video/audio input plugins.
- Seeking problems.
- It might be much trouble to get specific DirectShow filter doing the decoding for you. In many cases you will have multiple decoders that can decode the same specific format. The one which will be used is the one with the highest merit. It might be difficult to ensure that a specific decoder is doing the decoding. The document "[Importing media into AviSynth](#)" contains some more information about this.

### 7.1.4 Has AviSynth a direct stream copy mode like VirtualDub?

No, the video and the audio are decompressed when opening them into AviSynth.

There is a modification of AviSynth v2.55 which supports 'direct stream copy' for both video and audio. This modification is called DSynth and can be downloaded [here](#). Perhaps it will be updated and merged into the official AviSynth builds one day.

### 7.1.5 How do I load AVI files into AviSynth?

Use [AviSource](#) to load your AVI files in AviSynth. Example:

```
AviSource("d:\filename.avi")
```

or without the audio:

```
AviSource("d:\filename.avi", false)
```

If AviSynth is complaining about not being able to load your avi (couldn't decompress ...) you need to install an appropriate codec. [GSpot](#), for example, will tell you what codec you need to install in order to be able to open your avi.

Forcing a decoder being used for loading the clip into AviSynth:

```
# load your avi using the XviD codec:
```

7.1.3 What are the advantages and disadvantages of using DirectShowSource to load your media files?

## Avisynth 2.5 Selected External Plugin Reference

```
AviSource("d:\filename.avi", fourCC="XVID") # opens an avi (for example encoded with DivX3) using  
  
# load your dv-avi using the Canopus DV Codec:  
AviSource("d:\filename.avi", fourCC="CDVC")
```

### 7.1.6 Can I load video with audio from AVI into AviSynth?

It is always possible to demux your audio from the AVI file and load it separately in AviSynth using an audio decoder, but in some cases (for example: AVI with MP2/MP3/AC3/DTS audio) it is possible to load it directly in AviSynth.

For loading your AVI with audio you need (1) a Vfw (Video for Windows) codec to open (that is decode) your video in AviSynth and an ACM (Audio Compression Manager) codec to open your audio in AviSynth. For many video and audio format such codecs are available, but certainly not for all of them.

In the document "[Importing media into AviSynth](#)" you can find those codecs.

### 7.1.7 How do I load MPEG-1/MPEG-2/DVD VOB/TS/PVA into AviSynth?

DGDecode is an external plugin and supports MPEG-1, MPEG-2 / VOB, TS (with MPEG-4 ASP video) and PVA streams. Open them into DGIndex first and create a d2v script which can be opened in AviSynth (note that it will only open the video into AviSynth):

A few examples:

```
# DGDecode:  
LoadPlugin("d:\dgdecode.dll")  
mpeg2source("d:\filename.d2v")
```

If your transport stream (\*.TS) contains MPEG-4 AVC video you need to demux the raw video stream from it and use [DGAVCDec](#) to open it in AviSynth. See [here](#) for its usage.

### 7.1.8 How do I load QuickTime files into AviSynth?

There are two ways to load your quicktime movies into AviSynth (and also RawSource for uncompressed movs): QTSource and QTReader. The former one is very recent and able to open many quicktime formats (with the possibility to open them as YUY2), but you need to install QuickTime player in order to be able to use this plugin. The latter one is very old, no installation of a player is required in order to be able to open quicktime formats in AviSynth.

QTSource:

You will need Quicktime 6 for getting video only or Quicktime 7 for getting audio and video.

```
# YUY2 (default):  
QTInput("FileName.mov", color=2)  
  
# with audio (in many cases possible with QuickTime 7)  
QTInput("FileName.mov", color=2, audio=true)
```

## Avisynth 2.5 Selected External Plugin Reference

```
# raw (with for example a YUYV format):
QTInput("FileName.mov", color=2, mode=1, raw="yuyv")

# dither = 1; converts raw 10bit to 8bit video (v210 = 10bit uyvy):
QTInput("FileName.mov", color=2, dither=1, raw="v210")
```

QTReader:

If that doesn't work, or you don't have QuickTime, download the QTReader plugin (can be found in Dooms download section):

```
LoadVFAPIPlugin("C:\QTReader\QTReader.vfp", "QTReader")
QTReader("C:\quicktime.mov")
```

### 7.1.9 How do I load raw source video files into AviSynth?

The external plugin RawSource supports all kinds of raw video files with the YUV4MPEG2 header and without header (video files which contains YUV2, YV16, YV12, RGB or Y8 video data).

Examples:

```
# This assumes there is a valid YUV4MPEG2-header inside:
RawSource("d:\yuv4mpeg.yuv")

# A raw file with RGBA data:
RawSource("d:\src6_625.raw", 720, 576, "BGRA")

# You can enter the byte positions of the video frames directly (which can be found with yuvscal)
# This is useful if it's not really raw video, but e.g. uncompressed MOV files or a file with s
RawSource("d:\yuv.mov", 720, 576, "UYVY", index="0:192512 1:1021952 25:21120512 50:42048512 75:
```

### 7.1.10 How do I load RealMedia files into AviSynth?

RM/RMVB (RealMedia / RealMedia Variable Bitrate usually containing Real Video/Audio): install the [rmvb splitter](#) and the Real codecs by installing RealPlayer/[RealAlternative](#). Create the script:

```
DirectShowSource("d:\clip.rmvb", fps=25, convertfps=true) # adjust fps if necessary
```

### 7.1.11 How do I load Windows Media Video files into AviSynth?

WMV/ASF (Windows Media Video / Advanced Systems Format; usually containing WMV/WMA) is not fully supported by ffdshow, so you will have to install wmv codecs. Get [WMF SDK v9 for W2K or later for XP/Vista](#) which contains the codecs (and the DMO wrappers necessary to use DMO filters in DirectShow). You can also get these codecs from Windows Media Player 9 Series or later, Windows Media Format runtime (WMFDist.exe), Codec Installation Package (WM9Codecs.exe) from Microsoft site or other place. (Note that Microsoft's own VC1 codec is not supported in W2K since you need WMF SDK v11.) Create the script:

```
DirectShowSource("d:\clip.wmv", fps=25, convertfps=true) # adjust fps if necessary
```

## 7.1.12 How do I load MP4/MKV/M2TS/EVO into AviSynth?

If your media file contains MPEG-4 ASP video, then there are two possibilities of opening them in AviSynth:

1) Using the plugin [FFmpegSource](#). All included dlls except ffmpegsource.dll should be copied to your system folder. Some examples:

```
# loading the video from MKV and returning a timecodes file:
FFmpegSource("D:\file.mkv", vtrack = -1, atrack = -2, timecodes="timecodes_file.txt")

# loading the video and audio from a MP4 and returning a timecodes file:
FFmpegSource("D:\file_aac.mp4", vtrack = -1, atrack = -1, timecodes="timecodes_file.txt")
```

It's important to generate a timecode file to check whether the video has a constant framerate. If this the case you don't need to use the timecode file and you can process the video in any way you want. However, many non-AVI files contain video with a variable framerate (AVI files always have a constant framerate though), and in that case you need to make sure of the following two things:

1. *Don't change the framerate and the number of frames in AviSynth.* If you don't this (and you don't change the timecodes file manually) your video and audio in your final encoding will be out of sync.
2. *Use the timecodes file again when muxing your encoded video and audio.* If you don't do this your video and audio in your final encoding will be out of sync.

The main reason for this is that FFmpegSource opens the video as it is. It doesn't add or remove frames to convert it to constant framerate video to ensure sync.

2) Get [ffdshow](#) and open the MP4/MKV file with DirectShowSource, thus for example

```
DirectShowSource("D:\file.mkv", convertfps=true) # convertfps=true ensures sync if your video h
```

If your media file contains MPEG-4 AVC video, then there are two possibilities of opening them in AviSynth:

1) Using the plugin [FFmpegSource](#). See above for its usage. At the moment, the supported containers are: AVI, MKV and MP4.

2) Get [DGAVCDec](#). At the moment you need to extract the raw stream (\*.264) from the container first (using MKVExtract, MPlayer, TSRemux or whatever program can extract those streams). Open the raw stream file in DGAVCIndex to create an index file (say track1.dga). Open the index file in AviSynth:

```
# raw video demuxed from M2TS (Blu-ray BDAV MPEG-2 transport streams)
LoadPlugin("C:\Program Files\AviSynth\plugins\DGAVCDecode.dll")
AVCSorce("D:\track1.dga")
```

## 7.1.13 How do I load WAVE PCM files into AviSynth?

Use WavSource to open your WAVE PCM files (assuming that they are smaller than 4GB):

```
WavSource("D:\file.wav")
```

## Avisynth 2.5 Selected External Plugin Reference

Use the plugin RaWav to open your WAVE PCM files that are larger than 4GB ([Sonic Foundry Video Editor Wave64 Files or W64](#)):

```
RaWavSource("D:\file.w64", SampleRate=96000, SampleBits=24, Channels=6)
```

```
# or when a W64 header is present
```

```
RaWavSource("D:\file.w64", SampleRate=6) # assumes the presence of a W64 header and reads the m
```

### 7.1.14 How do I load MP1/MP2/MP3/MPA/AC3/DTS/LPCM into AviSynth?

Use NicAudio for loading your MP1/MP2/MP3/MPA/AC3/DTS/LPCM in AviSynth:

Some examples:

```
LoadPlugin("C:\Program Files\AviSynth25\plugins\NicAudio.dll")
```

```
# AC3 audio:
```

```
V = BlankClip(height=576, width=720, fps=25)
```

```
A = NicAC3Source("D:\audio.AC3")
```

```
# A = NicAC3Source("D:\audio.AC3", downmix=2) # downmix to stereo
```

```
AudioDub(V, A)
```

```
# LPCM audio (48 kHz, 16 bit and stereo):
```

```
V = BlankClip(height=576, width=720, fps=25)
```

```
A = NicLPCMSource("D:\audio.lpcm", 48000, 16, 2)
```

```
AudioDub(V, A)
```

### 7.1.15 How do I load aac/flac/ogg files into AviSynth?

Use ffdshow (set AAC to libfaad or realaac), and use

```
DirectShowSource("d:\audio.aac")
```

For WAVE\_FORMAT\_EXTENSIBLE, ogg, flac, wma, and other formats, [BassAudio and the corresponding libraries and Add-Ons](#) can be used. Note that BassAudioSource can decode stereo aac/mp4, but it can't decode multichannel aac.

Some examples:

```
bassAudioSource("C:\ab\Dido\001 Here With Me.m4a")
```

```
bassAudioSource("C:\ab\Dido\001 Here With Me.aac")
```

### 7.1.16 How do I load pictures into AviSynth?

1) Use [ImageReader](#) or [ImageSource](#) to load your pictures into AviSynth (can load the most popular formats, except GIF and animated formats). See internal documentation for information.

2) Use the Immaavs plugin for GIF, animated formats and other type of pictures.

## Avisynth 2.5 Selected External Plugin Reference

```
# single picture:  
immareadpic("x:\path\pic.bmp")  
  
# animation:  
immareadanim("x:\path\anim.gif")  
  
# image sequence:  
immareadseq("x:\path\seq%3.3d.png", start=5, stop=89, fps=25, textmode=2, posx=50, posy=50)
```

[| Main Page](#) | [| General Info](#) | **[| Loading Clips](#)** | [| Loading Scripts](#) | [| Common Error Messages](#) | [| Processing Different Content](#) | [| Dealing with YV12](#) | [| Processing with Virtualdub Plugins](#) |

**\$Date: 2009/09/12 20:57:20 \$**

# 8 AviSynth FAQ – Using VirtualDub plugins in AviSynth

## 8.1 Contents

1. [Where can I download the latest version of scripts which import filters from VirtualDub?](#)
2. [Which filters can be imported?](#)
3. [Do these scripts work in RGB-space or in YUV-space?](#)
4. [How do I make such a script?](#)

### 8.1.1 Where can I download the latest version of scripts which import filters from VirtualDub?

The AviSynth scripts are on the [Shared functions](#) page, or you can download a package called vdub\_filtersv15.zip from [\[1\]](#).

### 8.1.2 Which filters can be imported?

Most filters. Read the corresponding documentation.

### 8.1.3 Do these scripts work in RGB-space or in YUV-space?

You need to convert your clip to RGB32 before applying these scripts.

### 8.1.4 How do I make such a script?

Take a look at the following example script (this VirtualDub filter can be downloaded from [Donald's homepage](#)):

Smart Bob by Donald Graft:

```
function VD_SmartBob(clip clip, bool show_motion, int threshold, bool motion_map_denoising)
{
  LoadVirtualdubPlugin("d:\bob.vdf", "_VD_SmartBob", 1)
  return clip.SeparateFields._VD_SmartBob(clip.GetParity?1:0,
    \ default(show_motion,false)?1:0, default(threshold,10),
    \ default(motion_map_denoising,true)?1:0)
}
```

The VirtualDub plugin is imported with the command "LoadVirtualdubPlugin". The first argument gives the path of the plugin, the second argument the name for the plugin that will be used in the script and the third argument is called the preroll.

The preroll should be set to at least the number of frames the filter needs to pre-process to fill its buffers and/or updates its internal variables. This last argument is used in some filters like: SmartBob,



## Avisynth 2.5 Selected External Plugin Reference

SmartDeinterlace, TemporalCleaner and others. The reason is that due to filtering architecture of VirtualDub the future frames can't be accessed by a filter. Divdee reports: "In the "Add filter" dialog of VirtualDub, some filters have a "Lag:" value in their description. I think this is the value that must be used as preroll. Unfortunately, this indication is not always present. In those cases you have to guess." Of course you can always ask the creator of the filter.

The first step is to find out the sequence of the arguments in the last line where the clip is returned. Configure the script in VirtualDub and select "Save processing Settings" in the File Menu or press Ctrl+S. Open the created .vcf file with a text editor and you should see lines like this:

```
VirtualDub.video.filters.Add("smart bob (1.1 beta 2)");
VirtualDub.video.filters.instance[0].Config(1, 0, 10, 1);
```

The order of the arguments is the one that has to be used in AviSynth. To find the role of the arguments, play with them in VirtualDub and examine the resulting lines.

The second step is to test the filter and to compare it with the VirtualDub filter itself. For the programming itself you can learn a lot by looking at the script which are already contained in vdub\_filters.avs.

Example script which uses the function VD\_SmartBob:

```
Import("d:\vdub_filters.avs")
AviSource("d:\filename.avi")
ConvertToRGB32() # only when necessary (but doesn't hurt)
VD_SmartBob(false, 10, true)
ConvertBackToYUY2() # only when necessary
```

The package vdub\_filtersv15.zip is a bit outdated since many new VirtualDub filters are not in it. If that's the case for your VirtualDub filter and you don't want to create a function yourself (such as VD\_SmartBob), could also use the following script:

```
LoadVirtualdubplugin("d:\bob.vdf", "VD_SmartBob", 1)
VD_SmartBob(1, 0, 10, 1) # parameters taken from the .vcf file
```

[| Main Page](#) | [| General Info](#) | [| Loading Clips](#) | [| Loading Scripts](#) | [| Common Error Messages](#) | [| Processing Different Content](#) | [| Dealing with YV12](#) | **Processing with Virtualdub Plugins** |

**\$Date: 2009/09/12 20:57:20 \$**

# 9 AviSynth FAQ – The color format YV12 and related processing and encoding issues

## 9.1 Contents

1. [What is YV12?](#)
2. [Where can I download the latest stable AviSynth version which supports YV12?](#)
3. [Where can I download the DGIndex/DGDecode plugin, which supports YV12, to import MPEG–1/MPEG–2/TS/PVA in AviSynth ?](#)
4. [Which encoding programs support YV12?](#)
5. [How do I use v2.5x if the encoding programs can't handle YV12 \(like TMPGEnc or CCE SP\)?](#)
6. [What will be the main advantages of processing in YV12?](#)
7. [How do I use VirtualDub/VirtualDubMod such that there are no unnecessary color conversions?](#)
8. [Which internal filters support YV12?](#)
9. [Which external plugins support YV12?](#)
10. [Are there any disadvantages of processing in YV12?](#)
11. [How do I know which colorspace I'm using at a given place in my script?](#)
12. [The colors are swapped when my I load a DivX file in AviSynth v2.5?](#)
13. [I got a green \(or colored line\) at the left or at the right of the clip, how do I get rid of it?](#)
14. [I installed AviSynth v2.5 and get the following error message: "Couldn't locate decompressor for format 'YV12' \(unknown\)."?](#)
15. [I installed AviSynth v2.5 and DivX5 \(or one of the latest Xvid builds of Koepi\), all I got is a black screen when opening my avs in VirtualDub/VirtualDubMod/MPEG–2 encoder?](#)
16. [Are there any lossless YV12 codecs, which I can use for capturing for example?](#)
17. [Some important links](#)

### 9.1.1 What is YV12?

These are several different ways to represent colors. For example: YUV and RGB colorspace. In YUV colorspace there is one component that represent lightness (luma) and two other components that represent color (chroma). As long as the luma is conveyed with full detail, detail in the chroma components can be reduced by subsampling (filtering, or averaging) which can be done in several ways (thus there are multiple formats for storing a picture in YUV colorspace). YV12 is such a format (where chroma is shared in every 2x2 pixel block) that is supported by AviSynth. Many important codecs stored the video in YV12: MPEG–4 (x264, XviD, DivX and many others), MPEG–2 on DVDs, MPEG–1 and MJPEG.

### 9.1.2 Where can I download the latest stable AviSynth version which supports YV12?

"AviSynth v2.57" (and more recent versions) can be downloaded [here](#).

### 9.1.3 Where can I download the DGIndex/DGDecode plugin, which supports YV12, to import MPEG–1/MPEG–2/TS/PVA in AviSynth ?

The latest DGIndex/DGDecode combo can be downloaded [here](#).

### 9.1.4 Which encoding programs support YV12?

The regular builds of Virtualdub (by Avery Lee) have YV12 support in fast recompress mode since v1.5.6. There are also two other options for encoding to DivX/XviD:

There is a modified version (called VirtualdubMod) which has YV12 support: This modification (by pulco-citron, Suiryc and Belgabor) has OGM and AVS-preview support. It can be downloaded from [here](#). In order to use the YV12 support (without doing any color conversions) you have to load your AVI in VirtualdubMod and select "fast recompress".

For easy (and fast) YV12 support, you can also try out the command line utility [AVS2AVI](#) – compresses video from an AviSynth script using any VFW codec, see also [here](#).

The MPEG-1/MPEG-2 encoders [HC](#) and [OuEnc](#) also support (and even require) YV12.

### 9.1.5 How do I use v2.5x if the encoding programs can't handle YV12 (like TMPGEnc or CCE SP)?

Using TMPGEnc you have to add the line "[ConvertToRGB24](#)" (with proper "interlaced" option) to your script, and for CCE SP you need to add the line "[ConvertToYUY2](#)" to your script, since Windows has no internal YV12 decompressor.

You can also install some [YV12 decompressor \(codec\)](#) which will decompress the YV12-AVI for you when loading the avi in TMPGEnc or CCE SP.

### 9.1.6 What will be the main advantages of processing in YV12?

- speed increase:  
That depends entirely on the external plugins whether they will have YV12 support or not. Speed increases like 25–35 percent are expected. Of course there will only be a large speed increase if both your source and target are in YV12, for example in DVD to DivX/Xvid conversions.
- no color conversions:  
The colour information doesn't get interpolated (so often) and thus stays more realistic.

MPEG-2 encoders such as CCE, ProCoder and TMPGEnc can't handle YV12 input directly. CCE and ProCoder needs YUY2, and TMPGEnc RGB24. This only means that the last line of AviSynth must be a [ConvertToYUY2](#) (for CCE/ProCoder, or [ConvertToRGB24](#) for TMPGEnc) call, and that you will not be able to take full advantage of YV12 colorspace. Still there are two advantages:

1. All internal filtering in AviSynth will be faster though (less data to filter, better structure to filter, and a very fast conversion from YV12 to YUY2), and you will definitely be able to tell the difference between v2.06 and v2.5.
2. If you are making a progressive clip there is another advantage. Putting off the YV12->YUY2 conversion until the end of the script allows you to first IVTC or Deinterlace to create progressive frames. But the YV12 to YUY2 conversion for progressive frames maintains more chroma detail than it does for interlaced or field-based frames.

The color conversions:

CCE: YV12 -> YUY2 -> YV12

TMPEGEnc: YV12 -> RGB24 -> YV12

### 9.1.7 How do I use VirtualDub/VirtualDubMod such that there are no unnecessary color conversions?

Just load your avs file in VirtualDub/VirtualDubMod and set the video on "Fast recompress". In this mode the process will stay in YV12 (all the necessary filtering has to be done in AviSynth itself). Under compression select a codec which support YV12, like Xvid, DivX5, RealVideo (provided you download the latest binaries) or 3ivx D4 (provided you download the latest binaries). Note that DivX3/4 also supports YV12, except that PIV users could experience crashes when encoding to DivX4 in YV12.

If you want to preview the video you also need a [YV12 decompressor](#).

### 9.1.8 Which internal filters support YV12?

In principal all internal filters support YV12 natively. Which color formats the filters support is specified in the documentation.

### 9.1.9 Which external plugins support YV12?

The plugins which are compiled for AviSynth v2.5 are given in [External plugins](#). New plugins are listed in this [sticky](#). Most of them support YV12 (see documentation).

### 9.1.10 Are there any disadvantages of processing in YV12?

- If source format is not YV12 (analog capture, DV) or final encoding format is not YV12, then color format conversion will result in chroma interpolation with some quality decreasing.
- Filtering of subsampled chroma can result in some chroma broadening relatively luminosity pixels, especially for interlaced video.
- Because the chroma in interlaced YV12 video occurs on alternating lines, it is necessary to use a different upsampling/downsampling method when converting between YV12 and YUV 4:2:2 or RGB. This can lead to chroma upsampling/downsampling errors if the wrong color space conversion method is used on the video.
- If YV12 video is stored in an AVI container, there is no metadata to indicate whether the video is interlaced or progressive. This means that an application or component doing color space conversion has no easy way of choosing the correct conversion method (interlaced or progressive). Most color space converters assume progressive which can lead to chroma upsampling/downsampling errors when interlaced video is processed in such an environment.

### 9.1.11 How do I know which colorspace I'm using at a given place in my script?

To see which colorspace you are using at a given place in your script, add:

```
Info()
```

... and AviSynth will give you much information about colorspace amongst other things!

### 9.1.12 The colors are swapped when I load a DivX file in AviSynth v2.5?

This happens due to a bug in old versions of DivX5. Download the latest binaries or use "[SwapUV\(clip\)](#)".

### 9.1.13 I got a green (or colored line) at the left or at the right of the clip, how do I get rid of it?

Your decoder is probably borked, try a `ConvertToRGB()` at the end of your script just to be sure and check whether the line has disappeared. Some application have trouble displaying YV12 clips where the width or height is not a multiple of 16.

There are several solutions to this problem:

- Try having the codec decode to RGB or YUY2 (using `pixel_type="..."` argument in [AviSource](#) or [DirectShowSource](#)).
- Use a codec that correctly decodes YV12 clips where the width or height is not a multiple of 16.

### 9.1.14 I installed AviSynth v2.5 and get the following error message: "Couldn't locate decompressor for format 'YV12' (unknown)."

Install a codec which supports YV12. DivX5 or one of the recent [Xvid builds](#) or [Helix YUV codec](#) or some other (ffvfw, ffdshow). If that still doesn't work, modify your registry as explained in the next question.

### 9.1.15 I installed AviSynth v2.5 and DivX5 (or one of the latest Xvid builds of Koepi), all I got is a black screen when opening my avs in VirtualDub/VirtualDubMod/MPEG-2 encoder?

Ok, apparently your video is not decompressed by DivX 5.02 (or Xvid). Try to use [VCSwap utility](#) for hot swapping video codecs.

Advanced user can also do it by hand. Go to your windows-dir and rename a file called MSYUV.DLL, or add the following to your registry file:

```
REGEDIT4
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Drivers32]
"VIDC.YV12"="divx.dll"
```

Replace "divx.dll" by "xvid.dll" for xvid v0.9 or "xvidvfw.dll" for xvid v1.0.

### 9.1.16 Are there any lossless YV12 codecs, which I can use for capturing for example?

Capturing in YV12 is not recommended due to issues of interlacing and chroma; YUY2 will generally pose fewer problems. A lossless YV12 codec is more useful for saving intermediate files before a multi-pass encode, to avoid having to run a CPU-intensive script several times. There are several lossless YV12 codecs:

- [VBLE Codec \(by MarcFD\)](#): A huffyuv based encoder [[discussion](#)].
- [LocoCodec \(by TheRealMoh\)](#): see also [here](#).
- [ffvfw codec](#) – has various modes, in particular HuffYUV yv12.
- [Lagarith codec \(by Ben Greenwood\)](#) – better compression than Huffyuv but slower.

### 9.1.17 Some important links:

- [Technical explanation of YV12 \(and similar formats\)](#)
- [Good Microsoft page on YUV](#)
- [4:2:0 Video Pixel Formats](#)

[| Main Page](#) | [| General Info](#) | [| Loading Clips](#) | [| Loading Scripts](#) | [| Common Error Messages](#) | [| Processing Different Content](#) | [| Dealing with YV12](#) | [| Processing with Virtualdub Plugins](#) |

\$Date: 2009/09/12 20:57:20 \$

## 9.2 Filters with multiple input clips

There are some functions which combine two or more clips in different ways. How the video content is calculated is described for each function, but here is a summary which properties the result clip will have.

The input clips must always have the same color format and – with the exception of [Layer](#) and [Overlay](#) – the same dimensions.

filter	framerate	framecount	audio content	audio sampling rate
<a href="#">AlignedSplice</a> , <a href="#">UnalignedSplice</a>	first clip	sum of all clips	see filter description	first clip
<a href="#">Dissolve</a>	first clip	sum of all clips minus the overlap	see filter description	first clip
<a href="#">Merge</a> , <a href="#">MergeLuma</a> , <a href="#">MergeChroma</a> , <a href="#">Merge(A)RGB</a>	first clip	first clip (the last frame of the shorter clip is repeated until the end of the clip)	first clip	first clip
<a href="#">Layer</a>	first clip	first clip (the last frame of the shorter clip is repeated until the end of the clip)	first clip	first clip
<a href="#">Subtract</a>	first clip	longer clip (the last frame of the shorter clip is repeated until the end of the clip)	first clip	first clip
<a href="#">StackHorizontal</a> , <a href="#">StackVertical</a>	first clip	longer clip (the last frame of the shorter clip is repeated until the end of the clip)	first clip	first clip
<a href="#">Interleave</a>	(fps of first clip) x (number of clips)	N x frame-count of longer clip (the last frame of the shorter clip is repeated until the end of the clip)	first clip	first clip

As you can see the functions are not completely symmetric but take some attributes from the FIRST clip.

`$Date: 2008/07/19 15:17:14 $`

### 9.3 Getting started

Basically, AviSynth works like this: First, you create a simple text document with special commands, called a script. These commands make references to one or more videos and the filters you wish to run on them. Then, you run a video application, such as VirtualDub, and open the script file. This is when AviSynth takes action. It opens the videos you referenced in the script, runs the specified filters, and feeds the output to video application. The application, however, is not aware that AviSynth is working in the background. Instead, the application thinks that it is directly opening a filtered AVI file that resides on your hard drive.

#### 9.3.0.1 Linear Editing:

The simplest thing you can do with AviSynth is the sort of editing you can do in VirtualDub. The scripts for this are easy to write because you don't have to worry about variables and complicated expressions if you don't want.

For testing create a file called test.avs and put the following single line of text in it:

```
Version
```

Now open this file with e.g. Windows Media Player and you should see a ten-second video clip showing AviSynth's version number and a copyright notice.

`Version` is what's called a "source filter", meaning that it generates a clip instead of modifying one. The first command in an AviSynth script will always be a source filter.

Now add a second line to the script file, so that it reads like this:

```
Version  
ReduceBy2
```

Reopen the file in Media Player. You should see the copyright notice again, but now half as large as before. [ReduceBy2](#) is a "transformation filter," meaning that it takes the previous clip and modifies it in some way. You can chain together lots of transformation filters, just as in VirtualDub.

Let's add another one to make the video fade to black at the end. Add another line to the script file so that it reads:

```
Version  
ReduceBy2  
FadeOut(10)
```

Now reopen the file. The clip should be the same for the first 9 seconds, and then in the last second it should fade smoothly to black.

The [FadeOut](#) filter takes a numerical argument, which indicates the number of frames to fade.

It takes a long time before the fade starts, so let's trim the beginning of the clip to reduce the wait, and fade out after that.

## Avisynth 2.5 Selected External Plugin Reference

Let's discard the first 120 of them, and keep the frames 120–150:

```
Version
ReduceBy2
# Chop off the first 119 frames, and keep the frames 120–150
# (Avisynth starts numbering frames from 0)
Trim(120,150)
FadeOut(10)
```

In this example we used a comment for the first time.

Comments start with the # character and continue to the end of the line, and are ignored completely by Avisynth.

The [Trim](#) filter takes two arguments, separated by a comma: the first and the last frame to keep from the clip. If you put 0 for the last frame, it's the same as "end of clip," so if you only want to remove the first 119 frames you should use Trim(120,0).

Keeping track of frame numbers this way is a chore. It's much easier to open a partially–completed script in an application like VirtualDub which will display the frame numbers for you. You can also use the [ShowFrameNumber](#) filter, which prints each frame's number onto the frame itself.

In practice a much more useful source filter than [Version](#) is [AVISource](#), which reads in an AVI file (or one of several other types of files) from disk. If you have an AVI file handy, you can try applying these same filters to your file:

```
AVISource("d:\capture.avi") # or whatever the actual pathname is
ReduceBy2
FadeOut(15)
Trim(120,0)
```

Even a single–line script containing only the AVISource command can be useful for adding support for >2GB AVI files to applications which only support <2GB ones.

---

### 9.3.0.2 Non–Linear Editing:

Now we're getting to the fun part. Make an AVS file with the following script in it:

```
StackVertical(Version, Version)
```

Now open it. Result: An output video with two identical lines of version information, one on top of the other. Instead of taking numbers or strings as arguments, [StackVertical](#) takes video clips as arguments. In this script, the Version filter is being called twice. Each time, it returns a copy of the version clip. These two clips are then given to [StackVertical](#), which joins them together (without knowing where they came from).

One of the most useful filters of this type is [UnalignedSplice](#), which joins video clips end–to–end. Here's a script which loads three AVI files (such as might be produced by AVI\_IO) and concatenates them together.

```
UnalignedSplice(AVISource("d:\capture.00.avi"), \
  AVISource("d:\capture.01.avi"), \
  AVISource("d:\capture.02.avi"))
```



## Avisynth 2.5 Selected External Plugin Reference

Both [StackVertical](#) and [UnalignedSplice](#) can take as few as two arguments or as many as sixty. You can use the + operator as a shorthand for [UnalignedSplice](#).

For example, this script does the same thing as the previous example:

```
AVISource("d:\capture.00.avi") + \  
  AVISource("d:\capture.01.avi") + \  
  AVISource("d:\capture.02.avi")
```

Now let's suppose you're capturing with an application that also saves the video in multiple AVI segments, but puts the audio in a separate WAV file.

Can we recombine everything? You bet:

```
AudioDub(AVISource("d:\capture.00.avi") + \  
  AVISource("d:\capture.01.avi") + \  
  AVISource("d:\capture.02.avi"), \  
  WAVSource("d:\audio.wav"))
```

**\$Date: 2008/07/18 17:38:49 \$**

## 9.4 AviSynth Internet Links

[Official AviSynth site www.avisynth.org \(maintained by Richard Berg\)](http://www.avisynth.org)

[Official AviSynth project at sourceforge \(source codes and binaries download\)](#)

[AviSynth Usage forum at doom9 site](#)

[AviSynth Development forum at doom9 site](#)

[VirtualDub video editor \(by Avery Lee\)](#)

[VirtualDubMod video editor with AviSynth script editor](#)

[AVSEdit Editor description](#)

[AvsP Editor homepage \(by qwerpoi\)](#)

[AviSynth filter collection, some utilities too \(maintained by WarpEnterprises\)](#)

[Command-line and batch utilities for AviSynth](#)

[New plugins and utilities for AviSynth](#)

[Postprocessing video using AviSynth](#) (the section from Analog TV capture guide)

**\$Date: 2006/12/17 10:28:23 \$**

## 9.5 Scripting reference

This section contains information that goes beyond scripting basics. It presents the internals of AviSynth script processing, their influence on script performance as well as advanced techniques for using productively the features of the AviSynth script language. Before reading further it is recommended that you first become familiar with basic concepts of the [AviSynth syntax](#).

- [The script execution model](#)

The steps behind the scenes from the script to the final video clip output. The filter graph. Scope and lifetime of variables. Evaluation of runtime scripts.

- [User functions](#)

How to effectively write and invoke user functions; common pitfalls to avoid; ways to organise your function collection and create libraries of functions, and many more.

- [Block statements](#)

Techniques and coding idioms for creating blocks of AviSynth script statements.

- [Arrays](#)

Using arrays and array operators for manipulating collections of data in a single step.

- [Runtime environment](#)

How to unravel the power of runtime filters and create complex runtime scripts that can perform interesting (and memory/speed efficient) editing/processing operations and effects.

\$Date: 2008/04/20 19:07:33 \$

## 9.6 Arrays

As everybody using Avisynth knows, arrays are not supported natively by the scripting language.

However, a library named [\[AVSLib\]](#) exists that provides a functional interface for creating and manipulating arrays. Coupled with Avisynth's OOP style for calling functions, one can treat arrays as objects with methods, which is a familiar and easy to understand and code scripting concept.

Therefore, two preparatory steps are needed before being able to create and manipulate process arrays into your script:

- [\[Download\]](#) and install the most current version of AVSLib into your system.
- Import the needed AVSLib files in your script as follows (see the instructions inside the library's documentation to fill-in the gaps):

- ◇ AVSLib 1.1.x versions: Enter `LoadPackage("avslib", "array")` to load the array implementation files, or `LoadLibrary("avslib", CONFIG_AVSLIB_FULL)` to load entire AVSLib.

## Avisynth 2.5 Selected External Plugin Reference

◇ AVSLib 1.0.x versions: Enter an appropriate `Import({path to AVSLib header})` statement as the first line of your script.

Now you are ready to create your first array! In order to provide an almost real case example let's assume the following (which are commonplace in many situations) about the script you want to create:

- The script selects a distinct range of frames from each video clip.
- Some of the input clips may have different size, fps, audio and/or colorspace; thus they need to be converted.
- Some of the filtering parameters are distinct for each clip.

Having done that, let's proceed to the actual code:

First, we create the array; ..1.., ..2.., etc. are actual filename strings. Clip loading is made by [AviSource](#) in the example but [DirectShowSource](#) may also be specified.

```
inp = ArrayCreate( \  
  AviSource\(..1..\), \  
  AviSource\(..2..\), \  
  ... \  
  AviSource\(..n..\) )
```

Then we convert to same fps, audio, colorspace and size by using [AssumeFPS](#), [ConvertAudioTo16bit](#), [ConvertToYV12](#) and [BilinearResize](#) respectively (or any resizer that you find fit). We use OOP + chaining to make compact expressions.

Note that since Avisynth does not provide a way for in-place variable modification we must reassign to an array variable after each array operation (usually the same).

```
inp = inp.ArrayOpFunc("AssumeFPS", "24").ArrayOpFunc("ConvertAudioTo16bit" \  
  ).ArrayOpFunc("ConvertToYV12").ArrayOpFunc("BilinearResize", "640,480")
```

To perform trimming we will use arrays of other types also. Below *ts* stands for first frame to trim, *te* for last; each number corresponds to a clip in *inp* variable.

```
ts = ArrayCreate(12, 24, ..., 33) # n numbers in total  
te = ArrayCreate(8540, 7834, ..., 5712) # n numbers in total
```

We also need a counter to make things easier; we will use `ArrayRange` to create an array of 0,1,2,...

```
cnt = ArrayRange(0, inp.ArrayLen()-1)
```

In addition we must define a user function that will accept *inp*, *ts*, *te* and *cnt* and do the trimming.

Since `ArrayOpArrayFunc` only accepts two arrays for per-element processing, it is easier to pass 'inp' and *cnt* as array elements and *ts*, *te* as entire arrays.

```
Function MyTrim(clip c, int count, string fs, string fe) {  
  return c.Trim\(fs.ArrayGet\(count\), fe.ArrayGet\(count\)\)  
}
```

Now we are ready to do the trim (line below).

## Avisynth 2.5 Selected External Plugin Reference

```
inp = ArrayOpArrayFunc(inp, cnt, "MyTrim", StrQuote(ts)+", "+StrQuote(te))
```

We will finish the processing with a final tweak on brightness with different settings on each clip and on hue with same settings for all clips.

```
bright = ArrayCreate(2.0, 1.5, ..., 3.1) # n numbers in total
```

```
Function MyTweak(clip c, float br) {  
    return c.Tweak(bright=br, hue=12.3)  
}
```

```
inp = ArrayOpArrayFunc(inp, bright, "MyTweak")
```

And now we are ready to combine the results and return them as script's output. We will use [Dissolve](#) for a smoother transition.

```
return inp.ArraySum(sum_func="Dissolve", sum_args="5")
```

This is it; the n input clips have been converted to a common video and audio format, trimmed and tweaked with individual settings and returned as a single video stream with only 11 lines of code (excluding comments).

Other types of array processing are also possible (slicing ie operation on a subset of elements, joining, multiplexing, etc.) but these are topics to be discussed in other pages. Those that are interested can browse the [AVSLib documentation](#). One can also take a closer look at the [examples section](#) of the AVSLib documentation.

---

Back to [scripting reference](#).

\$Date: 2008/04/20 19:07:33 \$

## 9.7 Block statements

- [1 Background](#)
  - ◆ [1.1 Features enabling construction of block statements](#)
- [2 Implementation Guide](#)
  - ◆ [2.1 The if..else block statement](#)
    - ◇ [2.1.1 Using Eval\(\) and three–double–quotes quoted strings](#)
    - ◇ [2.1.2 Using separate scripts as blocks and the Import\(\) function](#)
    - ◇ [2.1.3 Using functions \(one function for each block\)](#)
  - ◆ [2.2 The if..elif..else block statement](#)
    - ◇ [2.2.1 Using Eval\(\) and three–double–quotes quoted strings](#)
    - ◇ [2.2.2 Using separate scripts as blocks and the Import\(\) function](#)
    - ◇ [2.2.3 Using functions \(one function for each block\)](#)
  - ◆ [2.3 The for..next block statement](#)
    - ◇ [2.3.1 For..Next loop with access to variables in local scope](#)
    - ◇ [2.3.2 For..Next loop without access to variables in local scope](#)
  - ◆ [2.4 The do..while and do..until block statements](#)
- [3 Deciding which implementation to use](#)
  - ◆ [3.1 The if..else and if..elif..else block statements](#)
  - ◆ [3.2 The for..next block statement](#)
  - ◆ [3.3 The do..while and do..until block statements](#)

- [4 References](#)

## 9.8 Background

A first glance at Avisynth documentation leaves the impression that aside from function definitions, block statements are not possible in Avisynth script. However, there are specific features of the language allowing the construction of block statements that have remained unaltered to date and probably will remain so in the future since block statements are very useful in extending the capabilities of the script language.

Indeed, in most programming and scripting languages, block statements are very useful tools for grouping together a set of operations that should be applied together under certain conditions. They are also useful in Avisynth scripts.

Assume, for example, that after an initial processing of your input video file, you want to further process your input differently (for example, apply a different series of [filters](#) or apply the same set of filters with different order) based on a certain condition calculated during the initial processing, which is coded at the value of Boolean variable *cond*.

Instead of making an ugly series of successive conditional assignments using the conditional (ternary) [operator](#), `? :`, as in **Example 1** below (items in brackets are not needed if you use the implicit *last* variable to hold the result):

### Example 1

```
[result_1 = ]cond ? filter1_1 : filter2_1
[result_2 = ]cond ? filter1_2 : filter2_2
...
[result_n = ]cond ? filter1_n : filter2_n
```

It would be nice to be able to construct two blocks of filter operations and branch in a single step, as in the (ideal) **Example 2** below:

### Example 2

```
[result = ] cond ? {
    filter1_1
    filter1_2
    ...
    filter1_n
} : {
    filter2_1
    filter2_2
    ...
    filter2_n
}
```

Something approaching this construction (and others) **is** possible; perhaps some constraints may apply, but you will nevertheless be capable of providing more powerful flow control to your scripts. The rest of this section will show you how to implement them.

### 9.8.1 Features enabling construction of block statements

The list below briefly presents the features making possible the creation of block statements in your script. Listed first are the more obvious ones, followed by those that are somewhat more esoteric and require a little digging inside the Avisynth documentation and experimenting with test cases to discover them.

- globals (in particular, variables preceded by the "global" keyword) allow the communication of information between code blocks executing in different context.
- The conditional operator (condition ? expr\_if\_true : expr\_if\_false) can contain an arbitrary number of nested expressions, if grouped by parentheses.
- Strings can contain double quote (") characters inside them if they are surrounded by three-double-quotes (""").
  - Thus, the following strings are valid in Avisynth script (note that the 2nd and 3rd ones could be lines in a script):
    - ◆ """"this is a string with " inside it""""
    - ◆ """"var = "a string value" """"
    - ◆ """"var = "a string value" # this is a comment""""
- There is a script function, [Eval\(\)](#), that allows the evaluation of strings containing arbitrary script expressions.
  - Thus, every expression that you can write in a script can be, if stored in a string, passed to Eval.
  - Eval returns the result of the evaluated expression, ie *anything* that can be constructed by such an expression (a clip, a number, a bool, a string).
  - The evaluation of the string is done in the same context as the call to Eval. Thus, if Eval is called at the script level, the expression is assumed to reference script-level variables or / and globals (globals are allowed everywhere). But if Eval is called inside a user-defined function then the expression is assumed to reference variables local to the function (ie arguments and any locally declared variable
- There is a script function, [Import\(\)](#), that allows the evaluation of arbitrary Avisynth scripts.
  - Thus, any script written in Avisynth script language can be evaluated by Import. Import returns the return value of the script.
  - Despite the common misbelief that this can only be a clip, it can actually be *any* type of variable (a clip, a number, a bool, a string).
  - Like Eval, the evaluation of the script is done in the same context as the call to Import.
  - Hence, as a side-effect of the script evaluation any functions and variables declared inside the imported script are accessible from the caller script, from the point of the Import call and afterwards.
- Recursion (ie calling a function from inside that function) can be used for traversing elements of a collection.
  - Thus, for..next, do..while, do..until loops can be constructed by using recursion.
- Multiline strings, ie strings that contain newlines (the CR/LF pair) inside them, are allowed by the script language.
- Multiline strings are parsed by [Eval\(\)](#) as if they were scripts.
  - Thus, each line of a multiline string will be evaluated as if it was a line in a script.

## Avisynth 2.5 Selected External Plugin Reference

Also, return statements inside the string are allowed (the value of their expression will be the return value of Eval(), as well as comments, function calls and in general every feature of the script language.

Consider the following **Example 3**, of a (useless) script that returns some black frames followed by some white frames:

### Example 3

```
c = BlankClip().Trim(0,23)
d = BlankClip(color=$ffffff).Trim(0,23)
b = true
dummy = b ? Eval( """
    k = c      # here comments are allowed!
    l = d
    return k   # this will be stored in dummy
    """) : Eval( """
    k = d
    l = c
    return k   # this will be stored in dummy
    """)
# variables declared inside a multiline string
# are available to the script after calling Eval
return k + l
```

Variables *k*, *l* are not declared anywhere before the evaluation of the if..else block. However, since Eval evaluates the string at the script-level context, it is as if the statements inside the string were written at the script level. Therefore, after Eval() they are available to the script. A few other interesting things to note are the following:

- The return statement at the end of the selected (by the value of *b*) string for evaluation is the value that will be returned to the *dummy* variable.
- Contrary to the case of line continuation by backslashes, a multiline string allows comments everywhere that they would be allowed in a script.

## 9.9 Implementation Guide

The features above can be used to construct block statements in various ways. The most common implementation cases are presented in this section, grouped by block statement type.

### 9.9.1 The if..else block statement

#### 9.9.1.1 Using Eval() and three-double-quotes quoted strings

This is by far the more flexible implementation, since the flow of text approaches most the "natural" (ie the commonly used in other languages) way of branching code execution.

Using the rather common case illustrated by **Example 1**, the solution would be (again items in square brackets are optional):

**Example 4**

```
[result = ] cond ? Eval("""
    filter1_1
    filter1_2
    ...
    filter1_n
[ return {result of last filter} ]
""") : Eval("""
    filter2_1
    filter2_2
    ...
    filter2_n
[ return {result of last filter} ]
""")
```

In short, you write the code blocks as if Avisynth script would support block statements and then enclose the blocks in three–double–quotes to make them multiline strings, wrap a call to [Eval](#) around each string and finally assemble Eval calls into a conditional operator statement.

The return statements at the end of each block are needed only if you want to assign a useful value to the *result* variable. If you simply want to execute the statements without returning a result, then you can omit the *return* statement at the end of each block.

One important thing to note is that the implicit setting of *last* continues to work as normal inside the Eval block. If the result of Eval is assigned to a variable, *last* will not be updated for the final expression in the block (with or without *return*), but it will be (where appropriate) for other statements in the block.

If the block statement produces a result you intend to use, it is clearer to enter a *return {result}* line as the last line of each block, but the keyword *return* is not strictly necessary.

The following real–case examples illustrate the above:

**Example 5** In this example, all results are assigned to script variables, so *last* is unchanged.

```
c = AviSource(...)
...
cond = {expr}
...
cond ? Eval("""
    text = "single double quotes are allowed inside three-double-quotes"
    pos = FindStr(text, "llo") # comments also
    d = c.Subtitle(LeftStr(text, pos - 1))
""") : Eval("""
    text = "thus by using three-double-quotes you can write expressions like you do in a script
    pos = FindStr(text, "tes")
    d = c.SubTitle(MidStr(text, pos + StrLen("tes")))
""")
return d
```

**Example 6** This example assigns a different clip to d depending on the [Framecount](#) of a source clip.

```
a = AviSource(...)
c = BlankClip().Subtitle("a test case for an if..else block statement")
d = a.Framecount >= c.Framecount ? Eval(""
    a = a.BilinearResize(c.Width, c.Height)
```



## Avisynth 2.5 Selected External Plugin Reference

```
c = c.Tweak(hue=120)
return Overlay(a, c, opacity=0.5)
""" : Eval("""
c = c.BilinearResize(a.Width, a.Height)
a = a.Tweak(hue=120)
return Overlay(c, a, opacity=0.5)
""")
return d
```

**Example 7** This example is a recode of Example 6 using implicit assignment to the *last* special variable. Since the result of the entire Eval() is not assigned to another variable, the implicit assignments to *last* on each line of the string (including the *last line* of the string) are preserved and thus the desired result is obtained.

```
c = BlankClip().SubTitle("a test case for an if..else block statement")
AviSource(...)
last.Framecount >= c.Framecount ? Eval("""
    BilinearResize(c.Width, c.Height)
    c = c.Tweak(hue=120)
    Overlay(last, c, opacity=0.5)
""") : Eval("""
    c = c.BilinearResize(last.Width, last.Height)
    Tweak(hue=120)
    Overlay(c, last, opacity=0.5)
""")
```

The only disadvantage of the Eval approach is that coding errors inside the string blocks are masked by the [Eval\(\)](#) call, since the parser actually parses a **single line** of code:

```
[result = ] cond ? Eval("""block 1""") : Eval("""block 2""")
```

Thus, any error(s) inside the blocks will be reported as a single error happening on the above line. You will not be pointed to the exact line of error as in normal script flow. Therefore, you will have to figure out where exactly the error occurred, which can be a great debugging pain, especially if you write big blocks.

### 9.9.1.2 Using separate scripts as blocks and the Import() function

Using **Example 1** as above, the solution would be (again items in square brackets are optional):

**Example 8** Code of script file *block1 avs*:

```
filter1_1
filter1_2
...
filter1_n
```

Code of script file *block2 avs*:

```
filter2_1
filter2_2
...
filter2_n
```

Code of main script where the conditional branch is desired:

```
...
```

### 9.9.1.2 Using separate scripts as blocks and the Import() function

## Avisynth 2.5 Selected External Plugin Reference

```
[result = ]cond ? Import("block1.avs") : Import("block2.avs")
...
```

In short, you create separate scripts for each block and then conditionally import them at the main script.

If you need to pass [variables](#) as "parameters" to the blocks, declare them in your main script and just reference them into the block scripts. The following example demonstrates this:

**Example 9** Code of script file *block1.avs*:

```
filter1_1(..., param1, ...)
filter1_2(..., param2, ...)
...
filter1_n(..., param3, ...)
```

Code of script file *block2.avs*:

```
filter2_1(..., param1, ...)
filter2_2(..., param2, ...)
...
filter2_n(..., param3, ...)
```

Code of main script where the conditional branch is desired:

```
# variables must be defined *before* importing the block script
param1 = ...
param2 = ...
param3 = ...
...
[result = ]cond ? Import("block1.avs") : Import("block2.avs")
...
```

Using [Import\(\)](#) instead of [Eval\(\)](#) and three–double–quoted multiline strings has some disadvantages:

- There is an administration overhead because instead of one file  $2k + 1$  files have to be maintained ( $k$  = the number of conditional branches in your script).
- The code has less clarity, in the sense that it does not visually appear as a block statement, neither the communication of parameters is apparent by inspection of the main script.

On the other hand:

- Debugging is not an issue; every error will be reported with accurate line information.
- You can reuse scripts that you frequently use and build more complex ones by simply importing ready–made components.
- For large–scale operations where few parameters have to be communicated it is usually a better approach.

One useful general purpose application of this implementation is to prototype, test and debug a block conditional branch and then recode it (by adding the `Eval()` and three–double–quotes wrapper code and removing the [global](#) keyword before the parameter's declarations) so that a single script using multiline strings as blocks is created. This workaround compensates for the main disadvantage of the `Eval()` and three–double–quotes implementation.

### 9.9.1.3 Using functions (one function for each block)

This is the most "loyal" to the Avisynth script's [syntax](#) approach. Using **Example 1** as above, the solution would be (again items in square brackets are optional):

#### Example 10

```
Function block_if_1()
{
    filter1_1
    filter1_2
    ...
    filter1_n
}

Function block_else_1()
{
    filter2_1
    filter2_2
    ...
    filter2_n
}
...
[result = ]cond ? block_if_1() : block_else_1()
...
```

In short, you create separate functions for each block and then conditionally call them at the branch point.

If you need to pass variables as "parameters" to the blocks, either declare them *global* in your main script and just reference them into the functions or – better – use argument lists at the functions. The following example demonstrates this:

#### Example 11

```
Function block_if_1(arg1, arg2, arg3, ...)
{
    filter1_1(..., arg1, ...)
    filter1_2(..., arg2, ...)
    ...
    filter1_n(..., arg3, ...)
}

Function block_else_1(arg1, arg2, arg3, ...)
{
    filter2_1(..., arg1, ...)
    filter2_2(..., arg2, ...)
    ...
    filter2_n(..., arg3, ...)
}
...
[result = ]cond \
    ? block_if_1(arg1, arg2, arg3, ...) \
    : block_else_1(arg1, arg2, arg3, ...)
...
```

Compared to the other two implementations this one has the following disadvantages:

## Avisynth 2.5 Selected External Plugin Reference

- There is an extra overhead due to the need for supplying function headers and (typically) argument lists.
- It tends to "pollute" the global namespace, thus having the potential of strange errors due to name conflicts; use a clear naming scheme, as the suggested above.

On the other hand:

- It is **portable**; it does not depend on any type of hack or specific behavior to work. It is thus guaranteed to continue working in the long term.
- It does not raise any special debugging difficulties.
- It has coding clarity.

### 9.9.2 The if..elif..else block statement

By nesting If..Else block expressions inside the conditional operator, you can create entire if..elseif...else conditional constructs of any level desired to accomodate more complex needs.

A generic example for each if..else implementation presented above is following. Of course, any combination of the three above pure cases is possible.

#### 9.9.2.1 Using Eval() and three-double-quotes quoted strings

The solution would be (again items in square brackets are optional):

##### Example 12

```
[result = \  
  cond_1 ? Eval("""  
    statement 1_1  
    ...  
    statement 1_n  
  """) : [(] \  
  cond_2 ? Eval(""" # inner a?b:c enclosed in parentheses for clarity (optional)  
    statement 2_1  
    ... # since backslash line continuation is used between Eval blocks  
    statement 2_n # place comments only inside the strings  
  """) : [(] \  
  ...  
  cond_n ? Eval("""  
    statement n_1  
    ...  
    statement n_n  
  """) \  
  : Eval("""  
    statement n+1_1  
    ...  
    statement n+1_n  
  """)[...)))] # 1 closing parenthesis for Eval() + n-1 to balance the opening ones (if used)
```

### 9.9.2.2 Using separate scripts as blocks and the Import() function

The solution would be (again items in square brackets are optional):

#### Example 13

```
# here no comments are allowed; every line but the last must end with a \
[result = \
  cond_1 ? \
    Import("block1.avs") : [() \
  cond_2 ? \
    Import("block2.avs") : [() \
  ...
  cond_n ? \
    Import("blockn.avs") \
  : \
    Import("block-else.avs") \
  ...) # n-1 closing parentheses to balance the opening ones
```

### 9.9.2.3 Using functions (one function for each block)

The solution would be (again items in square brackets are optional):

#### Example 14

```
# here no comments are allowed; every line but the last must end with a \
[result = \
  cond_1 ? \
    function_block_1({arguments}) : [() \
  cond_2 ? \
    function_block_2({arguments}) : [() \
  ...
  cond_n ? \
    function_block_n({arguments}) \
  : \
    function_block_else({arguments}) \
  [...)] # n-1 closing parentheses to balance the opening ones (if used)
```

## 9.9.3 The for..next block statement

The problem here is to implement the `for . . next` loop in a way that allows accessing variables in the local scope, so that changes made in local scope variables inside the loop can be accessible by the caller when it is finished. This is the way that the `for . . next` loop works in most programming languages that provide it. In addition, a means for getting out of the loop before is finished (ie breaking out of the loop) should be available.

There is of course the alternative to implement the `for . . next` loop in a way that does not allow access to local variables. This is easier in AviSynth, since then it can be implemented by a function; but it is also less useful. However in many cases it would be appropriate to use such a construct and thus it will be presented here.

**9.9.3.1 For..Next loop with access to variables in local scope**

1. Use a `ForNext(start, end, step, blocktext)` function to create a multiline string (a script) that will unroll the loop in a series of statements and then
2. use `Eval()` to execute the script in the current scope.

The `blocktext` is a script text, typically a multiline string in triple double quotes, that contains the instructions to be executed in each loop, along with special variables (say `#{i}` for the loop counter) that are textually replaced by the `ForNext` function with the current value(s) in each loop. The [StrReplace\(\)](#) function is particularly suited for the replacement task.

A little tweak is needed in order to implement the `break` statement; the unrolled string must be constructed in such a way that when the break flag is set the rest of the code is skipped.

The following proof-of-concept example demonstrates the procedure:

```
a = Avisource("c:\some.avi")
cnt = 12
b = a.Trim(0,-4)
cond = false

# here we would like to do the following
# for (i = 0; i < 6; i++) {
#   b = b + a.Trim(i*cnt, -4)
#   cond = b.Framecount() > 20 ? true : false
#   if (cond)
#     break
# }

return b
```

In order to make this happen in Avisynth, our script with `ForNext` would look like that:

```
a = Avisource("c:\some.avi")
cnt = 12
b = a.Trim(0,-4)
cond = false
block = ForNext(0, 5, 1, """
    b = b + a.Trim(#{i}*cnt, -4)
    cond = b.Framecount() > 20 ? true : false
    ${break(cond)}
    """)
void = Eval(block)
return b
```

and the output of `ForNext` with the above arguments should be something like this (the only problem is that string literals cannot be typed inside the block text):

```
"""
__break = false
dummy = __break ? NOP : Eval("
    b = b + a.Trim(0*cnt, -4)
    cond = b.Framecount() > 20 ? true : false
    __break = cond ? true : false
")
dummy = __break ? NOP : Eval("
    b = b + a.Trim(1*cnt, -4)
```

## Avisynth 2.5 Selected External Plugin Reference

```
cond = b.Framecount() > 20 ? true : false
__break = cond ? true : false
")
dummy = __break ? NOP : Eval("
b = b + a.Trim(2*cnt, -4)
cond = b.Framecount() > 20 ? true : false
__break = cond ? true : false
")
dummy = __break ? NOP : Eval("
b = b + a.Trim(3*cnt, -4)
cond = b.Framecount() > 20 ? true : false
__break = cond ? true : false
")
dummy = __break ? NOP : Eval("
b = b + a.Trim(4*cnt, -4)
cond = b.Framecount() > 20 ? true : false
__break = cond ? true : false
")
dummy = __break ? NOP : Eval("
b = b + a.Trim(5*cnt, -4)
cond = b.Framecount() > 20 ? true : false
__break = cond ? true : false
")
""
```

TO BE CONTINUED...

### 9.9.3.2 For..Next loop without access to variables in local scope

If we don't care for accessing variables in the local scope, then the implementation is straightforward:

1. Create an [AVSLib array](#) with the appropriate loop values.
2. Define needed globals (for example a bool flag to return immediately from the block if true).
3. Pack the block's code inside a function.
4. Use an [array operator](#) to execute the block for every loop value.

TO BE CONTINUED...

### 9.9.4 The do..while and do..until block statements

TODO...

## 9.10 Deciding which implementation to use

To be frank, there is no clear-cut answer to this question; it depends on the purpose that the script will serve, your coding abilities and habits, whether there are ready-made components available and what type are they (scripts, function libraries, etc.) and similar factors.

Thus, only some generic guidelines will be presented here, grouped on the type of block statement

### 9.10.1 The if..else and if..elif..else block statements

- For short (up to say 10 lines) blocks, using Eval() and three–double–quotes quoted strings is generally the best solution; it is fast to code and presents a "natural" text flow to the reader (thus it is easy to comprehend).
- For long blocks, using any of the other two implementations is generally better because it is easier to debug.
- If the blocks pre–exist as independent scripts, using [Import\(\)](#) is, obviously, preferred.
- If building a function library, usually an implementation with functions will be easier to maintain and debug. However using [Eval\(\)](#) for small blocks is still an option to consider, to minimise the risk of namespace clashing with user's own functions.

### 9.10.2 The for..next block statement

TODO...

### 9.10.3 The do..while and do..until block statements

TODO...

## 9.11 References

[1] [http://www.avisynth.org/stickboy/ternary\\_eval.html](http://www.avisynth.org/stickboy/ternary_eval.html)

[2] <http://forum.doom9.org/showthread.php?t=102929>

[3] <http://forum.doom9.org/showthread.php?p=732882#post732882>

---

Back to [scripting reference](#).

⌘Date: 2008/12/21 09:23:02 ⌘

## 9.12 The script execution model

This section is a walkthrough to the internals of the AviSynth script engine. Its aim is to provide a better understanding of how AviSynth transforms script commands to actual video frames and help a user that has already grasped the basics of AviSynth scripting to start writing better and optimised scripts.

The following subsections present the various parts that when combined together form what can be called the AviSynth's "script execution model":

- [Sequence of events](#)



## Avisynth 2.5 Selected External Plugin Reference

A detailed description of the sequence of events that occur when you execute (ie load and render to your favorite encoder) an AviSynth script.

- [The \(implicit\) filter graph](#)

A glance at the basic internal data structure that holds the representation of a parsed AviSynth script.

- [The fetching of frames \(from bottom to top\)](#)

How the AviSynth engine requests frames from filters.

- [Scope and lifetime of variables](#)

The interplay of variables' scope and lifetime with the other features of AviSynth [syntax](#).

- [Evaluation of runtime scripts](#)

The details of runtime scripts' evaluation.

- [Performance considerations](#)

Various performance-related issues and advice on how to optimise your AviSynth scripts and configuration.

---

Back to [scripting reference](#).

⌘Date: 2008/04/20 19:07:33 ⌘

### 9.13 The script execution model – Evaluation of runtime scripts

Evaluation of runtime scripts starts, as already stated, at the frame serving phase of the main script's execution. At that point frames of the final output clip are requested by the host video application. This triggers a sequence of successive calls to the GetFrame / GetAudio methods of all filters along the filter graph. Whenever one of those filters is a runtime filter, the following three-phase sequence of events happens *in every frame*:

- Runtime environment initialisation.
- Runtime script parsing and evaluation.
- Runtime environment cleanup and delivery of the resulting frame.

The following paragraphs examine each phase in more detail.

### 9.14 Contents

- [1 Runtime environment initialisation](#)
- [2 Runtime script parsing and evaluation](#)
- [3 Runtime environment cleanup and delivery of the resulting frame](#)
- [4 The runtime environment in detail](#)
  - ◆ [4.1 Elements of the runtime environment](#)
  - ◆ [4.2 Runtime functions and current frame](#)

◆ [4.3 Checklist for developing correct runtime scripts](#)

### 9.14.1 Runtime environment initialisation

The runtime filter code sets (at the top-level script local scope) its [special variables](#) for the runtime script. These at the minimum include `last`, which is set to the filter's source clip and `current_frame`, which is set to the frame number requested by the filter from the AviSynth code.

As a consequence, those special variables *cannot* be passed between runtime scripts; whatever value the passing script will set, it will be overwritten by the receiving filter's frame initialisation code.

### 9.14.2 Runtime script parsing and evaluation

The runtime script is parsed, as a regular script would be parsed if loaded in AviSynth. The parsing mechanism is the same. Thus *everything* allowed to a regular script is allowed to a runtime script; *what changes is the context of execution*. For example, you can:

- Use **multi-line scripts**; they just have to be contained inside a three-double-quotes pair (this is a requirement only if string literals are used inside the script, else single double quotes can be used also).
- Define / assign variables, both local and global.
- [Import](#) other scripts and/or load plugins and/or define functions.
- Call functions and filters.
- Use [arrays](#) and [block statements](#).
- Use [control structures](#).

Of course, some of the above are **not advisable**, because the different execution context poses different constraints regarding performance and resource usage. The main rule of thumb here is: **Parsing occurs in every frame requested. Therefore, computationally expensive actions should be avoided.** More on the [performance considerations](#) section.

### 9.14.3 Runtime environment cleanup and delivery of the resulting frame

The runtime filter code receives the result of script parsing and evaluation. If all went well, the result will be a valid filter graph (the runtime filter graph) from which the runtime filter requests to fetch the needed frame. If not, the filter will propagate the error to the caller. When the filter's code will return the final video frame, the runtime filter graph will be destroyed. As part of the cleanup the runtime filter code also restores the `last` special variable to its previous value.

### 9.14.4 The runtime environment in detail

Despite the very thin layer of added features (just a handful of variables and functions) the runtime environment is much more dynamic than the normal (main) script environment. The *key-difference* is the event-driven model of runtime script execution as opposed to the linear flow of the main script's execution. Execution of a runtime script occurs only in the event of a frame request. In addition, since intermediate filters in the chain may shuffle and combine frames in an arbitrary fashion, the requested frame's number may be

different than the final clip's frame number.

#### 9.14.4.1 Elements of the runtime environment

At any time during the frame serving phase, the elements of the runtime environment are the following:

- The environment inherited by the main script's parsing phase, that is the main script's top-level local variables, the global variables and all imported script and plugin functions.
- The [special variables](#) set on every frame by the runtime filter initialisation code (`last`, `current_frame`, etc.)
- A set of [runtime functions](#) to assist common information extraction operations.
- The environment created by the successive evaluation of runtime scripts triggered by all the final output clip's frames that have been requested so far by the host video application. This may include modifications to the environment inherited by the parsing phase such as change of variables' values, as well as addition of new locals and globals.

#### 9.14.4.2 Runtime functions and `current_frame`

An interesting feature of [runtime functions](#) is that they consult the value of the `current_frame` special variable in order to determine what frame of their input clip(s) to inspect for extracting information. This provides the ability inside a runtime script to easily request information for *any* frame of a clip by changing before the call to the function the `current_frame` variable.

As explained above, setting the `current_frame` variable has no effect on other runtime scripts in the filter chain because the runtime filters' initialisation code resets `current_frame` to the proper value before executing the runtime script. It also has no effect on subsequent filter calls in the runtime script. But it does have on runtime functions and anywhere the value of `current_frame` is used. Therefore, after such a usage it is good practice to restore the variable to its initial value before issuing other script commands.

A skeleton example of a runtime script that computes a weighted second order luma interpolation value (the actions after the computation are omitted) follows:

```
...previous processing omitted...
ScriptClip(""" # this is a multiline string
n = current_frame
lm_k = AverageLuma()
current_frame = n - 1
lm_kml = AverageLuma()
current_frame = n + 1
lm_kpl = AverageLuma()
dvg = (lm_kml - 2 * lm_k + lm_kpl) / 2
lm_ipl = lm_k + Sign(dvg) * Sqrt(Abs(dvg))
current_frame = n # remember to reset current_frame
...rest of script omitted...
""")
...subsequent processing omitted...
```

### 9.14.4.3 Checklist for developing correct runtime scripts

Despite the dynamic nature of the runtime environment, creating runtime scripts is relatively easy if you follow a simple set of rules:

- Remember that your input (source) clip is stored upon start of script execution in the `last` special variable.
- If you assign temporary clips to variables, remember to set `last` at the end or issue a `return` statement.
- Do not change (with respect to the source clip of the filter) the dimensions, colorspace or framerate of the final result.
- Do not assume – unless there is a **very** compelling performance–related reason– a particular ordering of frame requests; try to build ordering–neutral scripts, that depend only on `current_frame`.
- Inspect the names of all input variables (ie those variables that are *not initialised* to a value inside the runtime script) of your runtime scripts to ensure that they are not overridden accidentally by a normal, not used for inter–script communication variable in any runtime script along the chain.
- In particular, avoid putting inside [functions](#) calls to runtime filters that share state between invocations or with other filters through variables; it is easy to forget that *you may only call the function once*, else you will end up with multiple filters that share *the same* variables, thus with a bug in your script.

In view of the above, runtime filters should be used in functions only if they either:

- ◊ do not share state between invocations or with other filters through variables, or
- ◊ the function code takes care to create *unique names* of all the runtime script's shared variables on each function invocation.

A way to avoid variables is to dynamically build the runtime script using string concatenation and assign the related arguments' values to local variables in the runtime script. See the example code of the [bracket\\_luma](#) function.

Back to the [script execution model](#).

\$Date: 2008/04/20 19:07:33 \$

## 9.15 The script execution model – The fetching of frames

Despite that the script writer writes the script from top to bottom, AviSynth requests frames from the filters included in the script in the reverse order (**from bottom to top**). This is a consequence of the fact that AviSynth fakes the existence of an AVI file to the host video application through its AVI handler code.

The host video application does not know that AviSynth is behind the scenes; it just requests a finished and ready to render video frame, that is the final result of the script's processing from the AVI handler. The AVI handler in turn requests the frame from the root of the filter graph, ie from the final clip returned by the script.

In order to create the frame the final clip requests video frame(s) from its input clip(s). This process continues until a source filter, such as [AviSource](#), is reached which directly produces a video frame without requesting input from other filters. Therefore, the request process traverses the filter chain from bottom to top. That frame is then subsequently processed by all filters in the request chain (now from top to bottom, as in the script source code), resulting in the finished video frame that is served to the host video application.

This backward searching implementation has been chosen for effectiveness reasons and it doesn't need to bother you except for one important case: when you use runtime filters (cf. next sections).

Back to the [script execution model](#).

\$Date: 2008/04/20 19:07:33 \$

## 9.16 The script execution model – The filter graph

The final purpose of every AviSynth script is to create a filter graph that can be used by the top-level AVI stream object (see previous section) to call the filters needed for the creation of each specific frame that is requested by the host video application.

This filter graph is **implicit** in the sense that it is not directly available to script writers for querying or modification. However it is created and built-up on each filter call that the script source code makes, using the following rules:

- Each filter –actually its resulting clip– links to *all* its source clips (those that it uses directly or indirectly for input).
- Source filters (such as [AviSource](#)) do not link to other clips. Their resulting clip is a leaf-node in the filter graph.
- Only filters –ie their resulting clips– that are linked by the final clip returned by the script are part of the filter graph. That is a clip is part of the filter graph only if there is a path that connects it with the final clip returned by the script.

### 9.16.1 An example of a filter graph

Consider the following script, which loads a clip, makes a simple levels processing and muxes it with a custom sound track; then it overlays on top of it another clip, along with its inverted image, as separate windows.

```

AviSource("clip1.avi")
Levels(10, 1, 248, 0, 255)
aud = WavSource("mysoundtrack.wav")
AudioDub(last, aud)
ov = AviSource("clip2.avi")
ov1 = Lanczos4Resize(ov, 280, 210)
ov2 = ov1.Invert()
w = last.Width
h = last.Height
Overlay(ov1, x=w-280-40, y=40) # first clip is last
Overlay(ov2, x=w-280-40, y=h-210-40)
    
```

The filter graph that results from the script code above is the following.

```

AviSource(clip1) <-- Levels <--+
                        |
WavSource <-----+-----+ AudioDub <--+
                        |
AviSource(clip2) <-- Lanczos4Resize <--+-----+ Overlay <--+
                        |                               |
                        +-- Invert <-----+-----+ Overlay (filter graph's root)
    
```

The root of the filter graph, that is the filter from which the host video application requests video frames is the second [Overlay](#) (the last line of the script).

---

Back to the [script execution model](#).

\$Date: 2008/04/20 19:07:33 \$

## 9.17 The script execution model – Scope and lifetime of variables

There are essentially two scope types in the AviSynth script language:

### 9.18 Contents

- [1 Global scope](#)
- [2 Local scope](#)
- [3 Lifetime of variables](#)
- [4 A variables scope and lifetime example](#)
- [5 Code-injecting language facilities and scopes](#)
  - ◆ [5.1 Import and Eval](#)
  - ◆ [5.2 Runtime scripts](#)
  - ◆ [5.3 The try...catch block](#)

#### 9.18.1 Global scope

Every [variable](#) placed in this scope can be freely accessed from any nested block of script code at any level of nesting. That is, you can access a global from the top-level script block, from inside a user [function](#) at any level of recursion, as well as from inside [runtime scripts](#).

**Note:** One important precondition for the above rule to apply is that *you must not have a local variable with the same name as a global one*. If you define a local variable with the same name as a global one, then you can no longer get (read) the value of the global variable in that specific local scope. This is because AviSynth when given a variable's name it first searches in the current local scope; only if the search fails the global scope is searched. You can however set (write) the global's value, since in that case the use of the keyword `global` distinguishes between a global and a local variable.

#### 9.18.2 Local scope

Variables placed in this scope can be accessed (read or written) *only* from script code within that scope. This allows the isolation of nested local scopes and makes the creation and usage of script functions possible.

The most important local scope is the top-level script scope (the one that is created just before the executing script is parsed and evaluated). All non-global variables defined inside the script-level source code reside there. All the other local scopes are created due to function calls and are nested inside this one.

Nested local scopes result from function or filter calls (plugin writers can create nested scopes through `env->PushContext ( )` and `env->PopContext ( )`) and can be created at an arbitrary depth. For

example, a recursive user function such as the one below:

```
function strfill(string s, int count) {
    return count > 0 ? s + strfill(s, count - 1) : ""
}
```

will result during its evaluation in the creation and subsequent destruction of eleven nested local scopes if called with a value ten for its count argument.

### 9.18.3 Lifetime of variables

The lifetime of variables defined at the global and top-level script (local) scope spans from the time of the definition (that is the first statement that assigns a value to them) to the end of frame serving and the unload of avisynth.dll.

The lifetime of variables defined at nested local scopes spans from the time of the definition in the nested local scope to the end of the nested local scope's lifetime. Since nested local scopes result from function / filter calls, the nested scope's lifetime is the lifetime of the function / filter call.

### 9.18.4 A variables scope and lifetime example

To clarify the statements of the previous section, the following example demonstrates the scope and lifetime of variables in a moderately complex AviSynth script that also includes runtime scripts.

What the script does is to divide a clip (after some processing tweaks) in 4 equal-sized regions and evaluate the average luma of each region per frame. If this is outside a range defined by two thresholds, the corresponding region is turned to all black (if below) or white (if above) for that frame.

```
function Quartile(clip c, int quartile) {
    Assert(quartile >= 0 && quartile <= 3, "Invalid Quartile!")
    hw = Int(c.Width() / 2)
    hh = Int(c.Height() / 2)
    return Select(quartile, \
        Crop(c, 0, 0, hw, hh), Crop(c, hw, 0, hw, hh), \
        Crop(c, 0, hh, hw, hh), Crop(c, hw, hh, hw, hh))
}

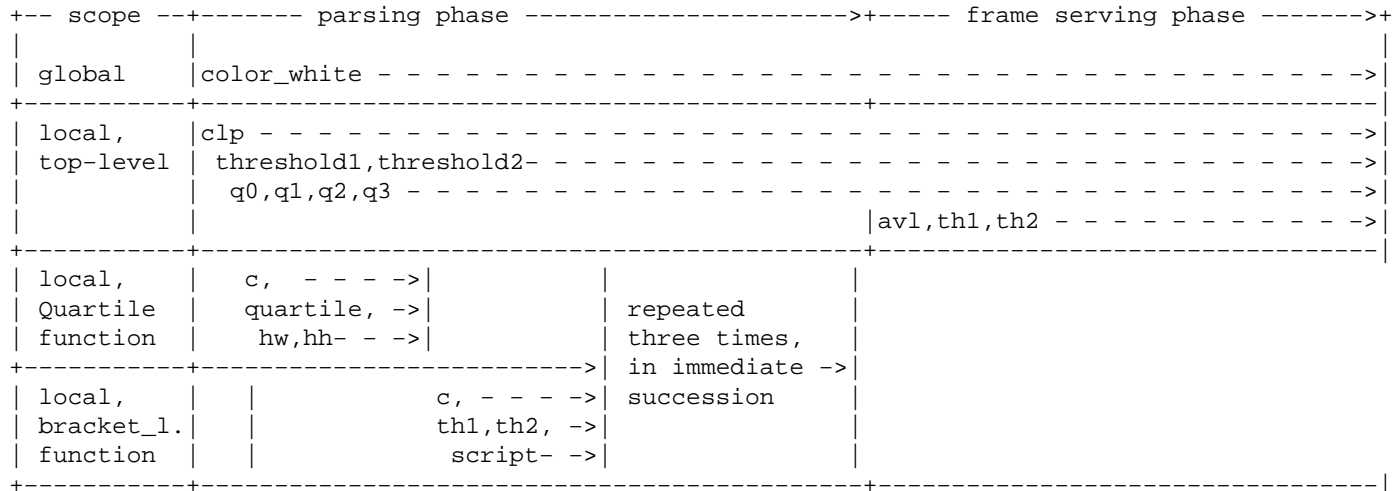
function bracket_luma(clip c, float th1, float th2) {
    Assert(0 <= th1 && th1 < th2 && th2 <= 255, "Invalid thresholds!")
    script = "th1 = " + String(th1) + Chr(13) + Chr(10) + \
        "th2 = " + String(th2) + ""
    avl = AverageLuma()
    return avl <= th1 ? last.BlankClip() : (avl >= th2 ? \
        last.BlankClip(color=color_white) : last)
    ""
    return ScriptClip(c, script)
}

clp = AviSource("myclip.avi")
clp = Tweak(clp, hue=20, sat=1.1)
threshold1 = 12.0
threshold2 = 78.0
q0 = Quartile(clp, 0).bracket_luma(threshold1, threshold2)
```

## Avisynth 2.5 Selected External Plugin Reference

```
q1 = Quartile(clp, 1).bracket_luma(threshold1, threshold2)
q2 = Quartile(clp, 2).bracket_luma(threshold1, threshold2)
q3 = Quartile(clp, 3).bracket_luma(threshold1, threshold2)
StackVertical(StackHorizontal(q0, q1), StackHorizontal(q2, q3))
```

The scope and lifetime of all variables in the example script is presented in the following timeline (the `color_white` global is from the autoloaded `.avsi` that ships with AviSynth):



### 9.18.5 Code-injecting language facilities and scopes

There are certain language constructs (functions, filters and control structures) that allow the injection of code in the script, ie the execution of arbitrary sequences of AviSynth script language [statements](#).

This is a *very* useful functionality that allows among other things dynamic code evaluation, the creation of [block statements](#) and [arrays](#), the organisation of AviSynth code in libraries, etc. However, there are some subtle issues regarding variables' scope and visibility that can lead to surprises if not fully understood.

#### 9.18.5.1 Import and Eval

[Import\(\)](#) and [Eval\(\)](#) evaluate the passed-in script source code in the context of the current local scope.

This means that variables contained in the top-level scope of the imported script or in the code string passed to `Eval()` are created inside the current local scope and become available for read/write to the following script source code. For example:

##### 1. File "a.avs"

```
x = 12
y = 24
c = BlankClip(pixel_type="YV12", color=color_orange, width=240, height=180)
```

##### 2. File "b.avs"

```
Import("a.avs")
AviSource("myvideo.avi")
```



## Avisynth 2.5 Selected External Plugin Reference

```
Levels(x, 1.0, 255, y, 242)  
Overlay(c, x=last.Width-320, y=last.Height-240, mode="chroma")
```

In addition, the imported script or the code string passed to `Eval()` can use previously defined in that scope local variables (as well as globals, of course). For example (the use of multiline triply quoted strings makes easier the writing of [block statements](#)):

```
x = 5  
AviSource("aclip.avi")  
f = Framecount()  
f < 100 ? Eval(""  
    Trim(x, f-2)  
    x = 0  
    """) : Eval(""  
    Trim(x, 15*x + 30)  
    x = 1  
    """)  
x == 0 ? Invert() : Subtitle(String(last.Framecount))
```

Especially the later is something that you must always keep in mind –mostly for [Import\(\)](#) since the code is not immediately visible; only the filename shows up in the script– because it has the potential to introduce bugs by unexpected overriding of a variable's value.

Consider, the following example:

### 1. File "mylib.avsi"

```
function preset(int num) { # 0 to 3  
    return Select(num, AviSource("..."), AviSource("..."), AviSource("..."), AviSource("..."))  
}  
global def_preset = preset(0)
```

### 2. File "myscript.avs"

```
global def_preset = AviSource("myfav.avi")  
Import("mylib.avsi")  
Tweak(def_preset, hue=20) # oops, using clip from mylib.avsi instead of myscript.avs!  
...
```

The imported script changed a previously defined variable and the results will now be suprising (until of course the bug is discovered).

However, this same feature has a number of interesting possibilities, for example:

- You can define sub–scripts that communicate with the parent script through a defined set of variables.
- You can create libraries (Avisynth include files) that perform initialisation code based on "environment" variables (the ones you set in the parent script before importing) and / or return status information (through a variable that they set at the global or top–script level code)
- You can implement [block statements](#).

**Note:** To test for the existence of input/output variables in the above scenarios try to read their value in a `try..catch` block; else your script will die hard if for any reason they do not exist.

### 9.18.5.2 Runtime scripts

Local variables inside runtime filters' scripts are **always** binded to the top-level script local scope; even if the filter calls were made inside a user function. This is because the parsing of runtime scripts is done *after* the parsing of the script, at the frame serving phase. At that point in script execution, nested local scopes have already vanished and only the global and the top-level script local scopes survive.

The same is true for the [special variables](#) set by the runtime filters (such as for example `current_frame`); they are defined at the top-level script local scope.

Some consequences of the above setup are the following:

- You can use top-level script local variables inside the runtime scripts to pass information, just as is customary to do with global ones.
- You must be careful if you define local variables in your runtime scripts to not clash with local variables in other runtime scripts in the filter chain. This is also true for globals, but globals are typically used for inter-filter communication; use of locals is not so common and thus may be overlooked by script writers.
- Overriding a variable (either local or global) does not have an effect at the main script, because the evaluation of the main script is done at the parsing phase, before the execution of any runtime script.
- When examining the way that a variable will be modified by a chain of runtime scripts, you must remember that the evaluation of scripts is done from bottom to top, just like the fetching of frames.

Consider the following example:

```

AviSource("myclip.avi")
x = 5
fc = Framecount()
fc > 2x ? Trim(x, fc - x) : Trim(0, fc - x)
fc = Framecount()
ScriptClip("""Subtitle("and the value of x is : " + String(x))""")
FrameEvaluate("x = (x % 3 == (fc - x - 1) % 3) ? x + 2 : x - 1")
FrameEvaluate("x = current_frame")

```

The assignment `x = 5` at the main script is used to control trimming of the source clip. `x` is passed as argument in the [Trim](#) filter during the script's parsing phase. Thus the modifications by the runtime scripts that start at the frame serving phase has no effect on the values passed to [Trim](#).

By the time the first frame will be fetched, `x` will have been overridden by the `x = current_frame` assignment in the last [FrameEvaluate](#) filter's runtime script. Thus its value in the script has no effect (in this particular case) to the results of the runtime filters processing.

Here, using `x` in all runtime filter scripts does not pose a naming clash problem. `x` is the variable used to communicate state information along the runtime filter chain. However, if we have needed a conditional assignment by frame number and we have accidentally used the following runtime script in place of the last [FrameEvaluate](#) line,

```

FrameEvaluate("""
    fc = 12
    x = current_frame < fc ? current_frame : fc
""")

```

then there would be a clash with the use of `fc` in the previous line (the clips framecount would have been overwritten with an unrelated value) and the logic of our processing would be in error.

### 9.18.5.3 The try...catch block

This may seem surprising at first, but the `try . . . catch` block does inject code in the script (at the scope that contains it). If this code defines new variables, then those variables are available to the code in the section that follows the `try . . . catch` block. More specifically, there are two possibilities:

- *No error* occurs inside the `try{ . . . }` section.
  1. All statements of the code contained in the `try{ . . . }` section are evaluated and affect the script code that follows.
- An error *does* occur inside the `try{ . . . }` section.
  1. Statements of the code contained in the `try{ . . . }` section up to the point of error are evaluated and affect the script code that follows.
  2. All statements of the code contained in the `catch{ . . . }` section are evaluated and affect the script code that follows.
  3. The variable that is used the `catch{ . . . }` section to store the error message becomes available to the script code that follows.

The following example code excerpt clarifies the above:

```
a = ... # it is assumed that the (missing) code may result in a being either 1 or 0
try {
    y = 3
    x = 6 / a # if a == 0 this will lead to an error
    z = 12
}
catch (msg) {
    NOP
}
...code that follows...
```

Now, if *a* is *not* zero at the point the `try . . . catch` block is evaluated, then three new local variables in the current scope will be created (*x*, *y* and *z*) and be available for use by the code that follows.

If however, *a* is zero, then from the three variables in the `try` section only *y* will be created; in addition, since the `catch` section will be evaluated, *msg* will be created. Thus the variables available for use by the code that follows will be *x* and *msg*.

---

Back to the [script execution model](#).

\$Date: 2008/04/20 19:07:33 \$

## 9.19 The script execution model – Performance considerations

This section presents some performance-related issues that originate from the way AviSynth scripts are executed; it also provides advice on how to optimise your scripts and AviSynth configuration so that your

scripts are parsed and/or encoded faster.

### 9.19.1 Plugin auto-loading

An important thing to note is that auto-loading although is a convenient method to have all your favorite filters on hands, it *does* incur a speed penalty. The penalty is twofold:

1. The loading, registering and unloading of plugins takes some time. The parsing of .avsi scripts also takes some time. This time, although small, is payed *in every* AviSynth script invocation.
2. The registering of many functions and globals increases the size of internal AviSynth data structures, which on turn increases the seek time to locate a filter / variable during the parsing phase as well as during runtime script parsing.

For small scripts and / or small number of auto-loading plugins the ease of use outweighs the above speed penalty (since there is also a speed penalty in writing a lot of [LoadPlugin](#) calls in every script that needs them). However if you regularly write large and complex scripts and have a large number of plugins / include scripts in your AviSynth plugin folder, you should consider a more granular approach to increase overall script parsing / encoding performance.

For example, you could group [LoadPlugin](#) calls for related plugins in separate .avsi scripts and have a central .avsi script with a config function that loads different .avsi scripts depending on its arguments. Then place in the plugin folder only the central .avsi script and the bare-essential plugins that you use almost every time.

### 9.19.2 Frame caching and the effect on splitting filter graph's paths

In order to improve performance AviSynth places, transparently to script writers, a specialised Cache filter just after each filter. The purpose of the cache is to avoid the computationally expensive generation of a video frame that has recently been created; if the frame is in the cache then it is returned immediately, avoiding a possibly long chain of filter calls.

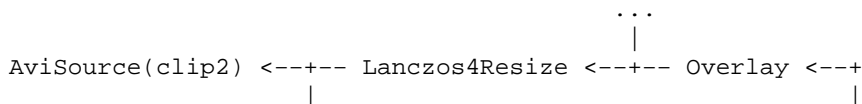
The presence of the cache gives a speed and memory advantage to filter graphs that split processing paths *as late as possible*. In our filter graph example above, if instead of:

```
ov = AviSource("clip2.avi")
ov1 = Lanczos4Resize(ov, 280, 210)
ov2 = ov1.Invert()
```

we have used the following code:

```
ov = AviSource("clip2.avi")
ov2 = ov
ov1 = Lanczos4Resize(ov, 280, 210)
ov2 = ov2.Lanczos4Resize(280, 210).Invert()
```

then the respective part of the filter graph would have been:



## Avisynth 2.5 Selected External Plugin Reference

```
+-- Lanczos4Resize <-- Invert <-----+-- Overlay (filter graph's root)
```

In the later case we would have one more filter (and cache) in the filter chain and –more importantly– we would have to generate two resized frames for each call by the host application to get a frame instead of one.

Therefore, always try to split processing paths as late as possible; it will make your scripts faster.

### 9.19.3 What *not* to include in runtime scripts

Although as said above, runtime scripts are parsed as regular scripts do and thus every statement allowed to a regular script is allowed in a runtime script, some statements are not advisable from a performance point of view.

The principal reason is that runtime script parsing occurs in *every* frame requested. Therefore, as a rule of thumb, computationally expensive actions should in general be placed outside the runtime environment (at the main script) in order to be executed only once. This practice trades some start–up overhead with savings during frame serving, which in general dominates the overall clip rendering / encoding time; thus it is justified as an optimisation. This is of course to be taken with a grain of salt because there are circumstances where the application needs force the (balanced) use of such statements.

Having said all that, let's see our not–to–do–in–runtime–scripts list (and some interesting counter–examples):

- The following actions should most of the time be avoided:
  - ◆ Importing a script.
  - ◆ Loading a plugin.
  - ◆ Defining a user function.

Issuing them on every frame will slow down (maybe significantly) encoding speed and (subject to implementation details) eat valuable memory. Moreover, this overhead will be beared without returning significant gains. It is in general much better to place them at the main script. See however an example of acceptable use: [Subtitles from a changing text–file](#).

- Calling a lot of filters / functions inside the runtime script will slow down your encoding speed. Those filters will be created and destroyed on every frame; thus you pay initialisation/cleanup costs at every frame.

If you can, put not–essential for the runtime processing filter calls outside the runtime environment; break the runtime script in more scripts if you have to. For example, instead of doing this:

```
AviSource("myclip.avi")
total_frames = Framecount()
ScriptClip(""
  Levels(0, 0.9, 255, 5, 250)
  total_frames % current_frame < 2 ? FlipHorizontal : last
  Tweak(hue=18)
  Subtitle("frame: " + String(current_frame), y=320)
  """)
```

do this:

```
AviSource("myclip.avi")
```

## Avisynth 2.5 Selected External Plugin Reference

```
total_frames = Framecount()  
Levels(0, 0.9, 255, 5, 250)  
ScriptClip(""total_frames % current_frame < 2 ? FlipHorizontal : last"")  
Tweak(hue=18)  
ScriptClip("" Subtitle("frame: " + String(current_frame), y=320)"")
```

- [Arrays](#), due to their recursive, script-based implementation can be expensive to parse, especially if they host a large number of elements. Using them without paying attention to minimize operations will slow down your encoding speed.

See however an example of acceptable use: [Per frame filtering, exporting specific frame\(s\)](#) (note that `FrameFilter` is a wrapper around [ScriptClip](#)).

---

Back to the [script execution model](#).

\$Date: 2008/04/20 19:07:34 \$

## 9.20 The script execution model – Sequence of events

The sequence of events that occur when you execute (ie load and render to your favorite encoder) an Avisynth script is presented below. The sequence is conceptually divided in three major phases:

1. [Initialisation phase](#)
2. [Script loading and parsing phase](#)
3. [The video frames serving phase](#)

### 9.20.1 The initialisation phase

During this phase the following events occur:

- An AVI stream object is created by the operating system, using the handler code provided by Avisynth.
- Avisynth dll is loaded and performs (among other initialization actions) auto-loading of the following files that are placed in the Avisynth autoload folder:
  - ◆ All Avisynth plugins (files with `.dll` extension).
  - ◆ All [VirtualDub](#) plugins (files with `.vdf` extension).
  - ◆ All Avisynth include scripts (files with `.avsi` extension).

During auto-loading:

- ◇ The plugin and script functions are registered in the internal data structures of Avisynth; all plugins are unloaded after this phase. Only if a function is actually used inside the script, the plugin is loaded again.
- ◇ Avisynth include scripts are parsed (executed), as if they were all included in the top of the script source code with [Import](#) statements.

## 9.20.2 The script loading and parsing phase

During this phase the following events occur:

- The host video application (the editor or the encoder) requests a frame from the AVI stream object (usually frame 0, but you should not count on this been always true).
- The script is loaded with a call to the Import [internal function](#) and its contents are passed to the Eval() [control function](#) for parsing.
- The entire parse process takes place. [Expressions](#) are evaluated, [variables](#) are set and clips are created and chained together to form the script's filter graph (see next section).
- At this point, just after the end of the parse process:
  - ◆ The filter graph has been formed.
  - ◆ Global variables have attained their final (regarding the parsing phase) values and, along with all functions defined in the script or imported in the global scope, are available for use and possibly **modification** by the runtime scripts.

## 9.20.3 The video frames serving phase

During this phase the following events occur:

- The initially requested frame is delivered to the host application. This forces runtime scripts that are possibly included in the source code to be parsed (executed) for that frame.
- The host video application requests additional frames. Each frame is delivered, possibly forcing runtime scripts that are in the filter chain for that frame to be parsed (executed).

**Note:** since runtime scripts may be anywhere in the filter chain and frames may be shuffled (AviSynth is an NLE after all!) the statement "for that frame" should be interpreted as "for that frame of the final clip and any linked frames of intermediate clips".

---

Back to the [script execution model](#).

⌘Date: 2008/04/20 19:07:34 ⌘

## 9.21 User functions

Having read the basics about [user-defined script functions](#), we can now step forward to examine in detail each function building block and identify rules for effective code development.

- [1 Manipulating arguments](#)
  - ◆ [1.1 Optional arguments](#)
  - ◆ [1.2 var arguments](#)
- [2 Manipulating globals](#)
- [3 Recursion](#)
- [4 Tuning performance](#)
- [5 Design and coding-style considerations](#)
- [6 Organising user defined functions](#)

## 9.21.1 Manipulating arguments

### 9.21.1.1 Optional arguments

### 9.21.1.2 `var` arguments

## 9.21.2 Manipulating globals

how to use effectively and safely

## 9.21.3 Recursion

the only tool to act upon collections

## 9.21.4 Tuning performance

## 9.21.5 Design and coding–style considerations

## 9.21.6 Organising user defined functions

---

Back to [scripting reference](#).

\$Date: 2008/04/20 19:07:34 \$

## 9.22 AviSynth Syntax

- [AviSynth Syntax](#) – The official reference documentation.
  - ◆ [Plugins](#) – How to load plugins (AviSynth, VirtualDub, VFAPI and C–plugins), autoloading and name–precedence.
  - ◆ [Script variables](#) – How to declare and use them in scripts.
  - ◆ [Operators](#) – Available operators and relative precedence.
  - ◆ [User defined script functions](#) – How to define and use them in scripts.
  - ◆ [Control structures](#) – Language constructs for script flow control.
  - ◆ [Internal functions](#) – Ready–made non–clip functions to use in scripts.
  - ◆ [Clip properties](#) – Functions that return a property of a clip.
  - ◆ [Runtime environment](#) – Scripting on a per clip frame basis.
- [Scripting reference](#) – Beyond scripting basics.
  - ◆ [The script execution model](#) – The steps behind the scenes from the script to the final video clip output. The filter graph. Scope and lifetime of variables. Evaluation of runtime scripts.
  - ◆ [User functions](#) – How to effectively write and invoke user defined script functions; common pitfalls to avoid; ways to organise your function collection and create libraries of functions, and many more.



- ◆ [Block statements](#) – Techniques and coding idioms for creating blocks of AviSynth script statements.
- ◆ [Arrays](#) – Using arrays (and array operators) for manipulating collections of data in a single step.
- ◆ [Scripting at runtime](#) – How to unravel the power of runtime filters and create complex runtime scripts that can perform interesting (and memory/speed efficient) editing/processing operations and effects.

\$Date: 2008/06/24 20:47:59 \$

### 9.23 AviSynth Clip properties

You can access clip properties in AVS scripts. For example, if the variable *clip* holds a video clip, then *clip.height* is its height in pixels, *clip.framecount* is its length in frames, and so on. Clip properties can be manipulated just like [script variables](#) (see the [AviSynth Syntax](#) for more), except that they cannot be l-values in C-terminology.

The full list of properties:

- Width (clip)

Returns the width of the clip in pixels (type: int).

- Height (clip)

Returns the height of the clip in pixels (type: int).

- FrameCount (clip)

Returns the number of frames of the clip (type: int).

- FrameRate (clip)

Returns the number of frames per seconds of the clip (type: float). The framerate is internally stored as a ratio though and more about it can be read [here](#).

- FrameRateNumerator (clip) (v2.55)

Returns the numerator of the number of frames per seconds of the clip (type: int).

- FrameRateDenominator (clip) (v2.55)

Returns the denominator of the number of frames per seconds of the clip (type: int).

- AudioRate (clip)

Returns the sample rate of the audio of the clip (type: int).

- AudioLength (clip) (v2.51)

## Avisynth 2.5 Selected External Plugin Reference

Returns the number of samples of the audio of the clip (type: int). Be aware of possible overflow on very long clips ( $2^{31}$  samples limit).

- **AudioLengthF (clip) (v2.55)**

Returns the number of samples of the audio of the clip (type: float).

- **AudioChannels (clip)**

Returns the number of audio channels of the clip (type: int).

- **AudioBits (clip)**

Returns the audio bit depth of the clip (type: int).

- **IsAudioFloat (clip) (v2.55)**

Returns true if the bit depth of the audio of the clip is float (type: bool).

- **IsAudioInt (clip) (v2.55)**

Returns true if the bit depth of the audio of the clip an integer (type: bool).

- **IsPlanar (clip) (v2.52)**

Returns true if the clip is [planar](#), false otherwise (type: bool).

- **IsRGB (clip)**

Returns true if the clip is [RGB](#), false otherwise (type: bool).

- **IsRGB24 (clip) (v2.07)**

Returns true if the clip is [RGB24](#), false otherwise (type: bool).

- **IsRGB32 (clip) (v2.07)**

Returns true if the clip is [RGB32](#), false otherwise (type: bool).

- **IsYUV (clip) (v2.54)**

Returns true if the clip is [YUV](#), false otherwise (type: bool).

- **IsYUY2 (clip)**

Returns true if the clip is [YUY2](#), false otherwise (type: bool).

- **IsYV12 (clip) (v2.52)**

Returns true if the clip is [YV12](#), false otherwise (type: bool).

## Avisynth 2.5 Selected External Plugin Reference

- IsPlanar (clip) (v2.51)

Returns true if the clip color format is [Planar](#), false otherwise (type: bool).

- IsFieldBased (clip)

Returns true if the clip is field-based (type: bool). What this means is explained [here](#).

- IsFrameBased (clip)

Returns true if the clip is frame-based (type: bool). What this means is explained [here](#).

- IsInterleaved (clip) (v2.52)

Returns true if the clip color format is Interleaved, false otherwise (type: bool).

- GetParity (clip, int n)

Returns true if frame n (default 0) is top field of field-based clip, or it is full frame with top field first of frame-based clip (type: bool).

- HasAudio (clip) (v2.56)

Returns true if the clip has audio, false otherwise (type: bool).

- HasVideo (clip) (v2.56)

Returns true if the clip has video, false otherwise (type: bool).

---

Back to [AviSynth Syntax](#).

\$Date: 2009/09/12 20:57:20 \$

## 9.24 AviSynth – Colors

In some filters ([BlankClip](#), [Letterbox](#), [AddBorders](#) and [FadeXXX](#)) a color argument can be specified. In all cases the color should be specified in [RGB](#) format even if the colorformat of the input clip is [YUV](#). This can be done in hexadecimal or decimal notation.

In **hexadecimal notation** the number is composed as follows: the first two digits denote the red channel, the next two the green channel and the last two the blue channel. The hexadecimal number must be preceded with a \$.

In **decimal notation** the number is as follows: the red channel value is multiplied by 65536, the green channel value is multiplied by 256 and the two resulting products are both added to the blue channel value.

Let's consider an example. Brown is given by R=\$A5 (165), G=\$2A (42), B=\$2A (42). Thus

```
BlankClip(color=$A52A2A)
```

## Avisynth 2.5 Selected External Plugin Reference

gives a brown frame. Converting each channel to decimal (remember that A=10, B=11, C=12, D=14, E=14, F=15) gives

$$R = \$A5 = 10*16^1 + 5*16^0 = 165$$

$$G = \$2A = 2*16^1 + 10*16^0 = 42$$

$$B = \$2A = 2*16^1 + 10*16^0 = 42$$

$$165*65536 + 42*256 + 42 = 10824234$$

Thus creating a brown frame specifying the color in decimal notation gives

```
BlankClip(color=10824234)
```

Common color presets can be found in the file `colors_rgb.avsi`, which should be present in your plugin autoload folder (look into the file for list of presets). Thus `BlankClip(color=color_brown)` gives the same brown frames.

Note that black `RGB=$000000` will be converted to `Y=16, U=V=128` if the `colorformat` of the input clip is `YUV`, since the default color conversion `RGB [0,255] -> YUV [16,235]` is used.

```
$Date: 2008/04/20 19:07:34 $
```

## 9.25 AviSynth Syntax – Control structures

In the strict sense, [AviSynth Syntax](#) provides only one control structure (actually two, the other being the conditional [operator](#), `?:`, presented elsewhere), the `try..catch` statement.

## 9.26 Contents

- [1 The try..catch statement](#)
- [2 Other control structures \(in the broad sense\)](#)
  - ◆ [2.1 Example 1: Create a function that returns a n-times repeated character sequence](#)
  - ◆ [2.2 Example 2: Create a function that selects frames of a clip in arbitrary intervals](#)

### 9.26.1 The try..catch statement

The `try..catch` statement permits the execution of code that **may** generate a run-time error and the handling of the error, if it actually arises.

The full syntax of the `try..catch` statement is:

```
try {
    ...
    statements
    ...
}
catch(err_msg) {
    ...
    statements
    ...
}
```

## Avisynth 2.5 Selected External Plugin Reference

The `err_msg` string in the `catch` block contains the text generated by AviSynth when the error inside the `try` block was encountered. This text is the same that would appear in the familiar `MessageBox` that shows up when a fatal script error occurs.

You can query the text (it is a normal string [variable](#)) to find specific substrings inside that indicate the error that has been encountered. This is a technique that can produce many useful results (for an example, see [here](#)).

### 9.26.2 Other control structures (in the broad sense)

In the broad sense, there are many elements inside [AviSynth Syntax](#) that although not control structures by themselves, together they allow the creation of language constructs equivalent to a control structure. Those constructs in turn allow the performance of complex programming tasks.

The elements under consideration are the following:

1. The `Eval()` statement that allows execution of arbitrary script language statements (and its cousin `Apply` that simplifies calling functions by name).
2. Multiline strings and in particular multiline strings surrounded by triple double quotes (the `"""` sequence of chars), since they allow to write string literals inside them naturally, as one would do in a normal AviSynth script.
3. The `Import()` statement that allows execution of arbitrary scripts, which can return a value (not necessarily a clip; a script can return a value of any type, which can be assigned to a [variable](#) of the calling script).
4. Recursion (the ability to create recursive functions).
5. Control functions, in particular `Assert`, `Select`, `Default`, `NOP`.

The first three allow one to create simple [Block statements](#), such as branching blocks (the analogous of the `if..elseif..else` control structure commonly encountered in programming and scripting languages). A basic example is presented below (see the link above for more):

```
# define different filtering based on this flag
heavy_filtering = true
AviSource("c:\sources\mysource.avi")
# assign result of Eval() to a variable to preserve the value of the last special variable
dummy = flag ? Eval("""
    Levels(0, 1.2, 255, 20, 235)
    stext = "heavily filtered"
    Spline36Resize(720, 400)
""") : Eval("""
    stext = "lightly filtered"
    BicubicResize(720, 400)
""")
AddBorders(0, 40, 0, 40)
Subtitle(stext)
```

The fourth is the general tool provided by the [syntax](#) for operating on collections and calculating expressions of any complexity. It is also, currently, **the only** tool.

This does not mean that there is something that you can't do inside the AviSynth script language; in fact recursion together with assignment can achieve everything an imperative language with looping constructs can do. It just does it in a way that most people without special programming skills are not familiar, since functional programming is not a major ICT course but rather a specialised topic.

## Avisynth 2.5 Selected External Plugin Reference

The fifth are more or less necessary tools inside the above constructs for controlling input, setting values and facilitating proper execution of the construct.

Lets look at some examples of recursion, to grasp the general pattern that one must follow to master it for its purposes.

### 9.26.2.1 Example 1: Create a function that returns a n–times repeated character sequence

We will use an existing implementation, from [AVSLib](#) as our example:

```
Function StrFill(string s, int count, bool "strict") {
    strict = Default(strict, true)
    Assert((strict ? count >= 0 : true), "StrFill: 'count' cannot be negative")
    return count > 0 ? s + StrFill(s, count - 1) : ""
}
```

The recursion is the call that the function makes to itself at the `return` statement. In order to be done properly, the sequence of recursive calls must eventually end to a single return value. Thus a recursive function's return statement will always use the conditional [operator](#), `?:`.

This is all that is about recursion, the other two lines (where the fifth element, control functions are used) are simply for ensuring that proper arguments are passed in. The "strict" argument is just an add–on for using the functions in case where it should quietly (without throwing an error) return an empty string.

### 9.26.2.2 Example 2: Create a function that selects frames of a clip in arbitrary intervals

Filters like [SelectEvery](#) allow the efficient selection of arbitrary sets of frames. They require though that each set of frames has a constant frame separation with its successor and predecessor set (in other words, the sets are periodic on frame number). In order to select frames with varying separation (that is non–periodic) we have to resort to script functions that use recursion.

The function below is a generic frame selection filter, which in order to select arbitrary frames uses a user–defined function (the `func` argument must contain its name) that maps the interval `[s_idx..e_idx)` to the set of frames that will be selected. `func` must accept a single integer as argument and return the corresponding mapped frame number.

```
Function FSelectEvery(clip c, string func, int s_idx, int e_idx) {
    Assert(s_idx >= 0, "FSelectEvery: start frame index (s_idx) is negative")
    f = Apply(func, s_idx)
    return (s_idx < e_idx && f >= 0 && f < c.Framecount) \
        ? c.Trim(f, -1) + FSelectEvery(c, func, s_idx + 1, e_idx) \
        : c.BlankClip(length=0)
}
```

The recursive step (first conditional branch in the `return` statement) is again an expression that involves the function as a subpart. This is not necessary in the general case (depending on the specific task, it could also be just the function call) but it is the most usual case when building complex constructs.

`Apply` calls the user function to calculate the frame number (a more robust implementation would enclose this call in a `try...catch` block). If the frame number is within the clip's frames then the associated frame is appended to the result else recursion ends.

The following example will clarify the design:

```
# my custom selector (x^2)
Function CalcFrame(int idx) { return Int(Pow(idx, 2)) }

AviSource("my_200_frames.avi")
# select up to 20 frames, mapped by CalcFrame
# in this case: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196
FSelectEvery(last, "CalcFrame", 0, 20)
```

\$Date: 2008/04/21 20:31:23 \$

### 9.27 AviSynth Syntax – Internal functions

In addition to [internal filters](#) AviSynth has a fairly large number of other (non-clip) internal functions. The input or/and output of these functions are not clips, but some other variables which can be used in a script. They are roughly classified as follows:

- [Numeric functions](#)

They provide common mathematical operations on numeric variables.

- [String functions](#)

They provide common operations on string variables.

- [Boolean functions](#)

They return true or false, if the condition that they test holds or not, respectively.

- [Conversion functions](#)

They convert between different types.

- [Control functions](#)

They facilitate flow of control (loading of scripts, arguments checks, global settings adjustment, etc.).

- [Version functions](#)

They provide AviSynth version information.

- [Runtime functions](#)

These are internal functions which are evaluated at every frame. They can be used inside the scripts passed to runtime filters ([ConditionalFilter](#), [ScriptClip](#), [FrameEvaluate](#)) to return information for a frame.

\$Date: 2008/04/20 19:15:05 \$

## 9.28 AviSynth Syntax – Boolean functions

Boolean functions return true or false, if the condition that they test holds or not, respectively.

- `IsBool` | | `IsBool(var)`

Tests if *var* is of the bool type. *var* can be any expression allowed by the [AviSynth Syntax](#).

*Examples:*

```
b = false
IsBool(b) = true
IsBool(1 < 2 && 0 == 1) = true
IsBool(123) = false
```

- `IsClip` | | `IsClip(var)`

Tests if *var* is of the clip type. *var* can be any expression allowed by the [AviSynth Syntax](#).

*Examples:*

```
c = AviSource\(...\)
IsClip(c) = true
IsClip("c") = false
```

- `IsFloat` | | `IsFloat(var)`

Tests if *var* is of the float type. *var* can be any expression allowed by the [AviSynth Syntax](#).

*Examples:*

```
f = Sqrt\(2\)
IsFloat(f) = true
IsFloat(2) = true # ints are considered to be floats by this function
IsFloat(true) = false
```

- `IsInt` | | `IsInt(var)`

Tests if *var* is of the int type. *var* can be any expression allowed by the [AviSynth Syntax](#).

*Examples:*

```
IsInt(2) = true
IsInt(2.1) = false
IsInt(true) = false
```

- `IsString` | | `IsString(var)`

Tests if *var* is of the string type. *var* can be any expression allowed by the [AviSynth Syntax](#).

*Examples:*

```
IsString("test") = true
IsString(2.3) = false
IsString(String\(2.3\)) = true
```

- `Exist` | v2.07 | `Exist(filename)`



Tests if the file specified by *filename* exists.

*Examples:*

```
filename = ...
clp = Exist(filename) ? AviSource(filename) : Assert(false, "file: " + filename + " does not exist")
```

- `Defined` | | `Defined(var)`

Tests if *var* is defined. Can be used inside [Script functions](#) to test if an optional argument has been given an explicit value.

More formally, the function returns false if its argument (normally a function argument or variable) has the void ('undefined') type, otherwise it returns true.

*Examples:*

```
b_arg_supplied = Defined(arg)
myvar = b_arg_supplied ? ... : ...
```

Back to [Internal functions](#).

\$Date: 2008/12/07 15:46:17 \$

## 9.29 AviSynth Syntax – Control functions

They facilitate flow of control (loading of scripts, arguments checks, global settings adjustment, etc.).

- `Apply` | | `Apply(string func_string [, arg1 [, arg2 [, ... [, argn]]])`

`Apply` calls the function or filter `func_string` with arguments `arg1`, `arg2`, ..., `argn` (as many as supplied). Thus, it provides a way to call a function or filter **by name** providing arguments in the usual way as in a typical function call.

Consequently, `Apply("f", x)` is equivalent to `f(x)` which in turn is equivalent to `Eval("f(" + String(x) + ")")`.

*Examples:*

```
# here the same call to BicubicResize as in the Eval() example is shown
Apply("BicubicResize", 352, 288)
```

- `Eval` | | `Eval(expression)`

`Eval` evaluates an arbitrary *expression* as if it was placed inside the script at the point of the call to `Eval` and returns the result of evaluation (either to the [variable](#) that is explicitly assigned to or to the last special variable).

You can use `Eval` to construct and evaluate expressions dynamically inside your scripts, based on variable input data. Below some specific examples are shown but you get the general idea.

*Examples:*

```
# this calls BicubicResize(last, 352, 288)
settings = "352, 288"
Eval("BicubicResize(" + settings + ")")
...
# this will result in Defined(u) == false
u = Eval("#")
```

## Avisynth 2.5 Selected External Plugin Reference

```
...
# this increments a global based on a variable's value
dummy = Eval("global my_counter = my_counter + " + String\(increment\))
```

- `Import | | Import(filename)`

`Import` evaluates the contents of another AviSynth script and returns the imported script's return value. Typically it is used to make available to the calling script library functions and the return value is not used. However this is simply a convention; it is not enforced by the [AviSynth Syntax](#). See also the dedicated [Import](#) page in [Internal filters](#) for other possible uses.

Possible scenarios (an indicative list) where the return value could be of use is for the library script to:

- ◇ indicate whether it successfully initialised itself (a bool return value),
- ◇ inform for the number of presets found on disk (an int return value);

the value then could be tested by the calling script to decide what action to take next.

*Examples:*

```
Import("mylib.avsi") # here we do not care about the value (mylib.avsi contains only functions
...
okflag = Import("mysources.avsi") # mysources loads predetermined filenames from a folder into
source = okflag ? global1 + global2 + global3 : BlankClip()
```

- `Select | | Select(index, item0 [, item1 [, ... [, itemn]]])`

Returns the item selected by the index argument, which must be of int type (0 returns item0, 1 returns item1, ..., etc). Items can be any [script variable](#) or expression of any type and can even be mixed.

*Examples:*

```
# select a clip-brush from a set of presets
idx = 2
brush = Select(idx, \
AviSource("round.avi"), \
rectangle, \
diagonal, \
diagonal.FlipHorizontal)
```

- `Default | | Default(x, d)`

Returns  $x$  if `Defined(x)` is true,  $d$  otherwise.  $x$  must either be a function's argument or an already declared script variable (ie a variable which has been assigned a value) else an error will occur.

*Examples:*

```
function myfunc(clip c, ..., int "strength") {
...
strength = Default(strength, 4) # if not supplied make it 4
...
}
```

- `Assert | | Assert(condition [, err_msg])`

Does nothing if condition is true; throws an error, immediately terminating script execution, if *condition* is false. In the later case `err_msg`, if supplied, is presented to the user; else the standard message "Assert: assertion failed". shows up.

*Examples:*

## Avisynth 2.5 Selected External Plugin Reference

```
function myfunc(clip c, ..., int "strength") {
    ...
    strength = Default(strength, 4) # if not supplied make it 4
    Assert(strength > 0, "'strength' must be positive")
    ...
}
```

- **NOP** | | **NOP()**

This is a no-operation function provided mainly for conditional execution with non-return value items such as [Import](#), when no "else" condition is desired. That is, use it whenever the [Avisynth Syntax](#) requires an operation (such as with the ?: operator) but your script does not need one. Return value: 0 (int type).

*Examples:*

```
preset = want_presets ? Avisource("c:\presets\any.avi") : NOP
...
loadlib ? Import("my_useful_functions.avs") : NOP
```

- **SetMemoryMax** | v2 | **SetMemoryMax(amount)**

Sets the maximum memory (in MB) that Avisynth uses for its internal Video Frame cache to the value of *amount*. From v2.5.8, setting to zero just returns the current Memory Max value. In the 2.5 series the default Memory Max value is 25% of the free physical memory, with a minimum of 16MB. From rev 2.5.8 RC4, the default Memory Max is also limited to 512MB.

Free	<64	128	256	512	1024	2048	3072
Default Max v2.57 and older	16	32	64	128	256	512	768
Default Max since v2.58 RC4	16	32	64	128	256	512	512

In some versions there is a default setting of 5MB, which is quite low. If you encounter problems (e.g. low speed) try to set this values to at least 32MB. Too high values can result in crashes because of 2GB address space limit.

Return value: Actual MemoryMax value set.

*Examples:*

```
SetMemoryMax(128)
```

- **SetWorkingDir** | v2 | **SetWorkingDir(path)**

Sets the default directory for Avisynth to the *path* argument.

This is primarily for easy loading of source clips, [importing](#) scripts, etc. It does not affect plugins' autoloading.

Return value is 0 if successful, -1 otherwise.

*Examples:*

## Avisynth 2.5 Selected External Plugin Reference

```
SetWorkingDir("c:\my_presets")  
AviSource("border_mask.avi") # this loads c:\my_presets\border_mask.avi
```

- [SetPlanarLegacyAlignment](#) | v2.56 | [SetPlanarLegacyAlignment\(mode\)](#)

Set alignment mode for [planar](#) frames. *mode* can either be true or false.

Some older [plugins](#) illegally assume the layout of video frames in memory. This special filter forces the memory layout of planar frames to be compatible with prior versions of Avisynth. The filter works on the [GetFrame\(\)](#) call stack, so it effects filters **before** it in the script.

*Examples:*

Example : Using an older version of [Mpeg2Source\(\)](#) (1.10 or older):

```
LoadPlugin("...\Mpeg2Decode.dll")  
Mpeg2Source("test.d2v") # A plugin that illegally assumes the layout of memory  
SetPlanarLegacyAlignment(true) # Set legacy memory alignment for prior statements  
ConvertToYUY2\(\) # Statements through to the end of the script have  
... # advanced memory alignment.
```

- [OPT\\_AllowFloatAudio](#) | v2.57 | `global OPT_AllowFloatAudio = True`}

This option enables `WAVE_FORMAT_IEEE_FLOAT` audio output. The default is to autoconvert Float audio to 16 bit.

- [OPT\\_UseWaveExtensible](#) | v2.58 | `global OPT_UseWaveExtensible = True`}

This option enables `WAVE_FORMAT_EXTENSIBLE` audio output. The default is `WAVE_FORMAT_EX`.

**Note:** The default DirectShow component for .AVS files, "AVI/WAV File Source", does not correctly implement `WAVE_FORMAT_EXTENSIBLE` processing, so many application may not be able to detect the audio track. There are third party DirectShow readers that do work correctly. Intermediate work files written using the AVIFile interface for later DirectShow processing will work correctly if they use the DirectShow "File Source (async)" component or equivalent.

---

Back to [Internal functions](#).

\$Date: 2008/12/21 21:42:29 \$

### 9.30 Avisynth Syntax – Conversion functions

Conversion functions convert between different types. There are also some [numeric functions](#) that can be classified in this category, namely: `Ceil`, `Floor`, `Float`, `Int` and `Round`.

- [Value](#) | v2.07 | [Value\(string\)](#)

Converts a decimal string to its associated numeric value.

*Examples:*

```
Value ("-2.7") = -2.7
```

- [HexValue](#) | v2.07 | [HexValue\(string\)](#)

## Avisynth 2.5 Selected External Plugin Reference

Converts a hexadecimal string to its associated numeric value.

*Examples:*

```
HexValue ("FF00") = 65280
```

- `String` | v2.07 | `String(float / int [, string format_string])`

Converts a variable to a string.

If the variable is float or integer, it first converts it to a float and then uses `format_string` to convert the float to a string. The syntax of `format_string` is as follows:

```
%[flags][width][.precision]f
```

*width*: the minimum width (the string is never truncated)

*precision*: the number of digits printed

*flags*:

- left align (instead right align)
- + always print the +/- sign
- 0 padding with leading zeros
- ' ' print a blank instead of a "+"
- # always print the decimal point

*You can also put arbitrary text around the format\_string as defined above, similar to the C-language printf function.*

*Examples:*

```
Subtitle( "Clip height is " + String(last.height) )
Subtitle( String(x, "Value of x is %.3f after AR calc") )
Subtitle( "Value of x is " + String(x, "%.3f") + " after AR calc" ) # same as above
String(1.23, "%f") = '1.23'
String(1.23, "%5.1f") = ' 1.2'
String(1.23, "%1.3f") = '1.230'
String(24, "%05.0f") = '00024'
```

---

Back to [Internal functions](#).

**\$Date: 2008/09/07 17:43:58 \$**

### 9.31 AviSynth Syntax – Multithreading functions

- `GetMTMode` | v2.6 | `GetMTMode(threads)`
- `SetMTMode` | v2.6 | `SetMTMode(mode, threads)`

These functions enable AviSynth to use more than one thread when processing filters. This is useful if you have more than one cpu/core or hyperthreading. This feature is still experimental.

`GetMTMode(bool threads)`:

If *threads* is set to true `GetMTMode` returns the number of threads used else the current mode is returned (see below). Default value false.

`SetMTMode(int mode, int threads)`:

## Avisynth 2.5 Selected External Plugin Reference

Place this at the first line in the avs file to enable temporal (that is more than one frame is processed at the same time) multithreading. Use it later in the script to change the mode for the filters below it.

*mode*: There are 6 modes 1 to 6. Default value 2.

- Mode 1 is the fastest but only works with a few filter.
- Mode 2 should work with most filters but uses more memory.
- Mode 3 should work with some of the filters that doesn't work with mode 2 but is slower.
- Mode 4 is a combination of mode 2 and 3 and should work with even more filter but is both slower and uses more memory
- Mode 5 is slowest but should work with all filters that doesn't require linear frameserving (that is the frames come in order (frame 0,1,2 ... last)).
- Mode 6 is a modified mode 5 that might be slightly faster.

*threads*: Number of threads to use. Set to 0 to set it to the number of processors available. It is not possible to change the number of threads other than in the first SetMTMode. Default value 0.

### Example:

```
SetMTMode(2,0) # enables multithreading using thread = to
                # the number of available processors and mode 2
LoadPlugin("...\LoadPluginEX.dll") # needed to load avisynth 2.0 plugins
LoadPlugin("...\DustV5.dll") # Loads Pixiedust
Import("limitedsharpen.avs")
src = AviSource("test.avi")
SetMTMode(5) # change the mode to 5 for the lines below
src = src.ConvertToYUY2.PixieDust() # Pixiedust needs mode 5 to function.
SetMTMode(2) # change the mode back to 2
src.LimitedSharpen() # because LimitedSharpen works well with mode 2
# display mode and number of threads in use
Subtitle("Number of threads used: " + String(GetMTMode(true))
\ + " Current MT Mode: " + String(GetMTMode()))
```

---

Back to [Internal functions](#).

\$Date: 2009/09/12 20:57:20 \$

## 9.32 AviSynth Syntax – Runtime functions

These are the internal functions which are evaluated at every frame. They can be used inside the scripts passed to runtime filters ([ConditionalFilter](#), [ScriptClip](#), [FrameEvaluate](#)) to return information for a frame (usually the current one). When using these functions there is an implicit **last** clip (the one that is passed to the runtime filter). Thus, first parameter doesn't have to be specified; it is replaced by the last clip.

- AverageLuma | v2.x | AverageLuma(clip)
- AverageChromaU | v2.x | AverageChromaU(clip)
- AverageChromaV | v2.x | AverageChromaV(clip)

This group of functions return a float value with the average pixel value of a plane (Luma, U–chroma and V–chroma, respectively). They require that clip is in [YV12 colorspace](#) and [ISSE](#).

*Examples:*

```
threshold = 55
```

## Avisynth 2.5 Selected External Plugin Reference

```
luma = AverageLuma()  
luma < threshold ? Levels(0, 1.0 + 0.5*(threshold - luma)/threshold, 255, 10, 255) : last
```

- [RGBDifference](#) | v2.x | [RGBDifference](#)(clip1, clip2)
- [LumaDifference](#) | v2.x | [LumaDifference](#)(clip1, clip2)
- [ChromaUDifference](#) | v2.x | [ChromaUDifference](#)(clip1, clip2)
- [ChromaVDifference](#) | v2.x | [ChromaVDifference](#)(clip1, clip2)

This group of functions return a float value between 0 and 255 of the absolute difference between two planes (that is two frames from two different clips). Either the combined RGB difference or the Luma, U–chroma or V–chroma differences, respectively. They require that `clip` is in [YV12 colorspace](#) and [ISSE](#).

*Examples:*

```
ovl = Overlay(last, mov_star, x=some_xvalue, y=some_yvalue, mask=mov_mask)  
ldif = LumaDifference(ovl) # implicit last for clip1  
udif = ChromaUDifference(Tweak(hue=24), ovl)  
...
```

The next two groups of functions should be quite handy for detecting scene change transitions:

- [RGBDifferenceFromPrevious](#) | v2.x | [RGBDifferenceFromPrevious](#)(clip)
- [YDifferenceFromPrevious](#) | v2.x | [YDifferenceFromPrevious](#)(clip)
- [UDifferenceFromPrevious](#) | v2.x | [UDifferenceFromPrevious](#)(clip)
- [VDifferenceFromPrevious](#) | v2.x | [VDifferenceFromPrevious](#)(clip)

This group of functions return the absolute difference of pixel value between the current and previous frame of `clip`. Either the combined RGB difference or the Luma, U–chroma or V–chroma differences, respectively.

*Examples:*

```
scene_change = YDifferenceFromPrevious() > threshold ? true : false  
scene_change ? some_filter(...) : another_filter(...)
```

- [RGBDifferenceToNext](#) | v2.x | [RGBDifferenceToNext](#)(clip)
- [YDifferenceToNext](#) | v2.x | [YDifferenceToNext](#)(clip)
- [UDifferenceToNext](#) | v2.x | [UDifferenceToNext](#)(clip)
- [VDifferenceToNext](#) | v2.x | [VDifferenceToNext](#)(clip)

This group of functions return the absolute difference of pixel value between the current and next frame of `clip`. Either the combined RGB difference or the Luma, U–chroma or V–chroma differences, respectively.

*Examples:*

```
# both th1, th2 are positive thresholds; th1 is larger enough than th2  
scene_change = YDifferenceFromPrevious() > th1 && YDifferenceToNext() < th2  
scene_change ? some_filter(...) : another_filter(...)
```

- [YPlaneMax](#) | v2.x | [YPlaneMax](#)(clip, float threshold)
- [UPlaneMax](#) | v2.x | [UPlaneMax](#)(clip, float threshold)
- [VPlaneMax](#) | v2.x | [VPlaneMax](#)(clip, float threshold)
- [YPlaneMin](#) | v2.x | [YPlaneMin](#)(clip, float threshold)

## Avisynth 2.5 Selected External Plugin Reference

- UPlaneMin | v2.x | UPlaneMin(clip, float threshold)
- VPlaneMin | v2.x | VPlaneMin(clip, float threshold)
- YPlaneMedian | v2.x | YPlaneMedian(clip)
- UPlaneMedian | v2.x | UPlaneMedian(clip)
- VPlaneMedian | v2.x | VPlaneMedian(clip)
- YPlaneMinMaxDifference | v2.x | YPlaneMinMaxDifference(clip, float threshold)
- UPlaneMinMaxDifference | v2.x | UPlaneMinMaxDifference(clip, float threshold)
- VPlaneMinMaxDifference | v2.x | VPlaneMinMaxDifference(clip, float threshold)

This group of functions return statistics about the distribution of pixel values on a plane (Luma, U–chroma and V–chroma, respectively). The statistics are, in order of presentation: maximum, minimum, median and range (maximum – minimum difference).

threshold is a percentage, stating how many percent of the pixels are allowed above or below minimum. The threshold is optional and defaults to 0.

*Examples:*

```
# median and average are close only on even distributions; this can be a useful diagnostic
have_intense_brights = YPlaneMedian() - AverageLuma() < threshold
...
# a simple per-frame normalizer to [16..235], CCIR, range
Levels(YPlaneMin(), 1.0, YPlaneMax(), 16, 235)
```

---

Back to [Internal functions](#).

\$Date: 2008/04/20 19:07:34 \$

### 9.33 AviSynth Syntax – String functions

String functions provide common operations on string variables.

- LCase | v2.07 | LCase(string)

Returns lower case of string.

*Examples:*

```
LCase("AviSynth") = "avisynth"
```

- UCase | v2.07 | UCase(string)

Returns upper case of string.

*Examples:*

```
UCase("AviSynth") = "AVISYNTH"
```

- StrLen | v2.07 | StrLen(string)

Returns length of string.

*Examples:*

```
StrLen("AviSynth") = 8
```



## Avisynth 2.5 Selected External Plugin Reference

- RevStr | v2.07 | RevStr(string)

Returns string backwards.

*Examples:*

```
RevStr("Avisynth") = "htnySivA"
```

- LeftStr | v2.07 | LeftStr(string, int)

Returns first int number of characters.

*Examples:*

```
LeftStr("Avisynth", 3) = "Avi"
```

- RightStr | v2.07 | RightStr(string, int)

Returns last int number of characters.

*Examples:*

```
RightStr("Avisynth", 5) = "Synth"
```

- MidStr | v2.07 | MidStr(string, int pos [, int length])

Returns substring starting at *pos* for optional *length* or to end. *pos*=1 specifies start.

*Examples:*

```
MidStr("Avisynth", 3, 2) = "is"
```

- FindStr | v2.07 | FindStr(string, substring)

Returns position of substring within string. Returns 0 if substring is not found.

*Examples:*

```
Findstr("Avisynth", "syn") = 4
```

- Chr | v2.51 | Chr(int)

Returns the ASCII character.

Note that characters above the ASCII character set (ie above 127) are code page dependent and may render different (visual) results in different systems. This has an importance only for user-supplied localised text messages.

*Examples:*

```
Chr(34) returns the quote character  
Chr(9) returns the tab character
```

- Time | v2.51 | Time(string)

Returns a string with the current system time formatted as defined by the string.

The string may contain any of the codes for output formatting presented below:

Code	Description
------	-------------

## Avisynth 2.5 Selected External Plugin Reference

%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%d	Day of month as decimal number (01 ? 31)
%H	Hour in 24-hour format (00 ? 23)
%I	Hour in 12-hour format (01 ? 12)
%j	Day of year as decimal number (001 ? 366)
%m	Month as decimal number (01 ? 12)
%M	Minute as decimal number (00 ? 59)
%p	Current locale's A.M./P.M. indicator for 12-hour clock
%S	Second as decimal number (00 ? 59)
%U	Week of year as decimal number, with Sunday as first day of week (00 ? 53)
%w	Weekday as decimal number (0 ? 6; Sunday is 0)
%W	Week of year as decimal number, with Monday as first day of week (00 ? 53)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without century, as decimal number (00 ? 99)
%Y	Year <i>with</i> century, as decimal number
%z, %Z	Time-zone name or abbreviation; no characters if time zone is unknown
%%	Percent sign

The # flag may prefix any formatting code. In that case, the meaning of the format code is changed as follows:

Code with # flag	Change in meaning
%#a, %#A, %#b, %#B, %#p, %#X, %#z, %#Z, %#%	No change; # flag is ignored.
%#c	Long date and time representation, appropriate for current locale. For example:  ?Tuesday, March 14, 1995, 12:41:29?.
%#x	Long date representation, appropriate to current locale. For example:  ?Tuesday, March 14, 1995?.

<pre> %#d, %#H, %#I, %#j, %#m, %#M,  %#S, %#U, %#w, %#W, %#y, %#Y         </pre>	Remove leading zeros (if any).
--	--------------------------------

Back to [Internal functions](#).

\$Date: 2008/04/20 19:07:34 \$

## 9.34 AviSynth Syntax – Version functions

Version functions provide AviSynth version information.

- `VersionNumber` | v2.07 | `VersionNumber()`

Returns AviSynth version number as a float.

*Examples:*

```
ver = VersionNumber() # ver == 2.57
```

- `VersionString` | v2.07 | `VersionString()`

Returns AviSynth version info as a string (first line used in [Version\(\)](#) command).

*Examples:*

```
VersionString() = "AviSynth 2.08 (avisynth.org) 22 nov. 2002"
```

Back to [Internal functions](#).

\$Date: 2008/04/20 19:07:34 \$

## 9.35 AviSynth Syntax – Operators

As in all programming and scripting languages, operators in AviSynth script language allow the performance of actions (operations) onto [variables](#). Operators form the basis for building up expressions, the building blocks of AviSynth scripts.

AviSynth operators follow loosely the same rules as C operators, regarding meaning, precedence and associativity. By loosely we mean that there are some exceptions, indicated at the text below.

### 9.35.1 Available Operators per Type

For **all types** of operands (clip, int, float, string, bool) you can use the following operators:

<code>==</code>	is equal
<code>!=</code>	not equal
<code>&lt;&gt;</code>	not equal (alternative to <code>!=</code> , v2.07)

## Avisynth 2.5 Selected External Plugin Reference

For **numeric** types (int, float) you can use the following int/float-specific operators:

+	add
-	subtract
*	multiply
/	divide
%	mod
>=	greater or equal than
<=	less or equal than
<	less than
>	greater than

*AviSynth in former versions parsed expressions from right to left, which gave unexpected results. For example:*

- $a = 10 - 5 - 5$  resulted in  $10 - (5 - 5) = 10$  instead of  $(10 - 5) - 5 = 0$  !
- $b = 100. / 2. / 4.$  resulted in  $100. / (2. / 4.) = 200$  instead of  $(100. / 2.) / 4. = 12.5$  !

These "bugs" have been corrected in v2.53!

For **string** type you can use the following string-specific operators:

+	concatenate
>=	greater or equal than (v2.07)
<=	less or equal than (v2.07)
<	less than (v2.07)
>	greater than (v2.07)

For **clip** type you can use the following clip-specific operators:

+	the same as the function <a href="#">UnalignedSplice</a>
++	the same as the function <a href="#">AlignedSplice</a>

For **bool** type (true/false) you can use the following bool-specific operators:

	or
&&	and
?:	execute code conditionally

The conditional execution operator is used as in the following example:

```
b = (a==true) ? 1 : 2
```

This means in pseudo-basic:

```
if (a=true) then b=1 else b=2
```

## Avisynth 2.5 Selected External Plugin Reference

From version v2.07, AviSynth provides a NOP() function which can be used inside a conditional execution block in cases where "else" may not otherwise be desirable (such as a conditional [Import](#) or [LoadPlugin](#)).

### 9.35.2 Operator Precedence

The precedence of AviSynth operators is presented at the table below. Operators higher to the top of the table have higher precedence. Operators inside the same row have the same order of precedence.

*	/	%				
+	++	-				
<	>	<=	>=	!=	<>	==
&&						
?:						

\$Date: 2008/04/21 20:31:23 \$

## 9.36 AviSynth Syntax – Plugins

With these functions you can add external functions to AviSynth.

`LoadPlugin ("filename" [, ...])`

Loads one or more external avisynth plugins (DLLs).

---

`LoadVirtualDubPlugin ("filename", "filtername", preroll)`

This loads a plugin written for VirtualDub. "filename" is the name of the .vdf file. After calling this function, the filter will be known as "filtername" in avisynth. VirtualDub filters only supports RGB32. If the video happens to be in RGB24-format, then you must use `ConvertToRGB32` (`ConvertToRGB` won't suffice).

Some filters output depends on previous frames; for those preroll should be set to at least the number of frames the filter needs to pre-process to fill its buffers and/or updates its internal variables.

---

`LoadVFAPIPlugin ("filename", "filtername")`

This allows you to use VFAPI plugins (TMPGEnc import plugins).

---

`LoadCPlugin ("filename" [, ...])`

`Load_Stdcall_Plugin ("filename" [, ...])`

Loads so called Avisynth C-plugins (DLLs).

`Load_Stdcall_Plugin()` is an alias for `LoadCPlugin()`.

C-plugins are created on pure C language and use special "AviSynth C API" (unlike ordinary Avisynth plugins which are created with MS C++). C-plugins must be loaded with `LoadCPlugin()` or `Load_Stdcall_Plugin()`.

Kevin provides a `LoadCPlugin.dll` that overloads the `LoadCPlugin()` verb to support plugins compiled using

the C subroutine calling sequence, use `Load_Stdcall_Plugin()` to load stdcall calling sequence plugins when using Kevins version. Advice: keep these plugins outside your auto plugin loading directory to prevent crashes. [\[discussion\]](#) [\[AVISynth C API \(by kevin20723\)\]](#)

## 9.37 Plugin autoload and name precedence v2

It is possible to put all plugins and script files with user-defined functions or (global) variables in a directory from where all files with the extension `.AVSI` (*v2.08, v2.5*, the type was `.AVS` in *v2.05–2.07*) and `.DLL` are loaded at startup, unloaded and then loaded dynamically as the script needs them.

`.AVSI` scripts in this directory should only contain function definitions and global variables, no main processing section (else strange errors may occur), it also is not recommended to put other files in that directory.

The directory is stored in the registry (the registry key has changed for *v2.5*). You can use double-clicking a `.REG`-file with the following lines to set the path (of course inserting your actual path):

```
REGEDIT4
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Avisynth]
"plugindir2_5"="c:\program files\avisynth 2.5\plugins"
```

The order in which function names take precedence is as follows:

```
user-defined function (always have the highest priority)
  plugin-function (have higher priority than built-in
  functions, they will override a built-in function)
    built-in function
```

Inside those groups the function loaded at last takes precedence, there is no error in a namespace conflict.

## 9.38 Plugin autoload and conflicting function names v2.55

Starting from *v2.55* there is `DLLName_function()` support. The problem is that two plugins can have different functions which are named the same. To call the needed one, `DLLName_function()` support is added. It auto-generates the additional names both for auto-loaded plugins and for plugins loaded with `LoadPlugin`.

**Some examples:**

```
# using fielddeinterlace from decomb510.dll
AviSource("D:\captures\jewel.avi")
decomb510_fielddeinterlace(blend=false)
```

Suppose you have the plugins `mpeg2dec.dll` and `mpeg2dec3.dll` in your auto plugin dir, and you want to load a `d2v` file with `mpeg2dec.dll` (which outputs `YUY2`):

```
# using mpeg2source from mpeg2dec.dll
mpeg2dec_mpeg2source("F:\From_hell\from_hell.d2v")
```

or with `mpeg2dec3.dll` (which outputs `YV12`):

```
# using mpeg2source from mpeg2dec3.dll
mpeg2dec3_mpeg2source( "F:\From_hell\from_hell.d2v" )
```

\$Date: 2008/04/20 19:07:34 \$

## 9.39 AviSynth Syntax

All basic AviSynth scripting statements have one of these forms:

1. *variable\_name* = *expression*
2. *expression*
3. **return** *expression*

(Two higher-level constructs also exist – the [function declaration](#) and the [try..catch statement](#).)

In the first case, *expression* is evaluated and the result is assigned to *variable\_name*. In the second case, *expression* is evaluated and the result, if a clip, is assigned to the special variable **last**. In the third case, *expression* is evaluated and is used as the "return value" of the active script block—that is either a function or the entire script. In the latter case, the return value is typically the video clip that will be seen by the application which opens the AVS file. As a shorthand, a bare expression as the final statement in a script (or script block) is treated as if the keyword **return** was present.

Most of the time the result of an expression will be a video clip; however an expression's result can be of any type supported by the scripting language (clip, int, float, bool, string) and this is how utility functions such as [internal script functions](#) operate.

An *expression* can have one of these forms:

1. *numeric\_constant*, *string\_constant* or *boolean\_constant*
2. *variable\_name* or *clip\_property*
3. *Function(args)*
4. *expression.Function(args)*
5. *expression1* **operator** *expression2*
6. *bool\_expression* ? *expression1* : *expression2*

In the first case, the value of the *expression* is the value of the constant. In the second case, the values correspond to [clip properties](#) or [script variables](#) (which must have been previously initialized). In the third case, the value is the return value of an AVS function (see below). The fourth case is an alternate syntax (called "OOP notation") which is equivalent to *Function(expression, args)*.

The final two cases show that one can manipulate expressions using all of the usual arithmetic and logical [operators](#) (from C) as you'd expect on ints, floats and bools, as well as execute code conditionally with the ternary operator. Strings can be compared with relational operators and concatenated with '+'. The following operators are also defined on video clips: **a + b** is equivalent to [UnalignedSplice\(a, b\)](#), and **a ++ b** is equivalent to [AlignedSplice\(a, b\)](#).

The functions in AviSynth's scripting language are, by and large, video filters. Although a function can return any type it chooses (this is a useful feature for creating utility code to reuse in scripts; you can define your own [script functions](#)) functions which do **not** return a **clip** are always limited to intermediate processing of variables to pass as arguments to filters (functions that *do* return a clip). The script should always return a clip as its final value. After all, AviSynth is a video processing application.

## Avisynth 2.5 Selected External Plugin Reference

Functions can take up to sixty arguments (hope that's enough), and the return value can be of any type supported by the scripting language (clip, int, float, bool, string). Functions always produce a new value and never modify an existing one. What that means is that all arguments to a function are passed "by value" and not "by reference"; in order to alter a variable's value in AviSynth script language you must assign to it a new value.

To see the syntax of the function call for each built-in filter, view the [internal filters](#). There are also built-in [internal functions](#) that perform common operations on non-clip variables.

*Args* is a list of function arguments separated by commas. The list can be empty. Each argument must be an expression whose type (eg text string, integer, floating-point number, boolean value or video clip) matches the one expected by the function. If the function expects a video clip as its first argument, and that argument is not supplied, then the clip in the special variable *last* will be used.

AviSynth functions can take named arguments. The named arguments can be specified in any order, and the filter will choose default values for any that you leave off. This makes certain filters much easier to use. For example, you can now write [Subtitle](#)("Hello, World!", text\_color=\$00FF00, x=100, y=200) instead of [Subtitle](#)("Hello, World!", 100, 200, 0, 999999, "Arial", 24, \$00FF00). [Colors](#) can be specified in hexadecimal as in the example above or in decimal. In both cases it should be specified as RGB value, even if the clip itself is YUV.

If no arguments are being passed to the function, you can also make the function call without parentheses, e.g. **FilterName**. The primary reason for this is to retain compatibility with old scripts. However, it's sometimes convenient to leave off the parentheses when there's no possibility of confusion.

Avisynth ignores anything from a # character to the end of that line. This can be used to add **comments** to a script.

```
# comment
```

In v2.58 it is possible to add **block** and **nested block** comments in the following way:

```
# block comment:
/*
comment 1
comment 2
*/

# nested block comments:
[* [* a meaningful example will follow later :) *] *]
```

Avisynth ignores anything from an \_\_END\_\_ keyword (with double underscores) to the end of the script file. This can be used to disable some last commands of script.

```
Version\(\)
__END__
ReduceBy2\(\)
Result is not reduced and we can write any text here
```

Avisynth **ignores case**: aViSouRCe is just as good as AVISource.

Multiple Avisynth statements on a single line can only be achieved in the context of OOP notation or embedding filters as parameters of another function such as:



## Avisynth 2.5 Selected External Plugin Reference

```
AviSource("c:\video.avi").Trim(0, 499)
```

–or–

```
AudioDub(AviSource("c:\video.avi"), WavSource("c:\audio.wav"))
```

Avisynth statements can be split across multiple lines by placing a backslash ("\") either as the last non-space character of the line being extended, or as the first non-space character on the next line.

Line splitting examples (both valid and equal):

```
Subtitle("Hello, World!", 100, 200, 0, \
  999999, "Arial", 24, $00FF00)
```

–or–

```
Subtitle("Hello, World!", 100, 200, 0,
  \ 999999, "Arial", 24, $00FF00)
```

When splitting across multiple lines you may *place comments only at the end of the last line*. Mixing comments with backslashes at an intermediate line of the line-split will either produce an error message or result at hard to trace bugs.

Example of a not-signalized bug by improper mixing of comments and line separation:

```
ColorBars
```

```
ShowFrameNumber
```

```
Trim(0,9) # select some frames \
  + Trim(20,29)
```

The above example does not return frames [0..9,20..29] as intended because the "\" is masked by the "#" character before it; thus the line continuation never happens.

```
$Date: 2008/12/21 09:23:02 $
```

## 9.40 AviSynth Runtime environment

### 9.40.1 Contents

- [1 Definition](#)
- [2 Runtime filters](#)
- [3 Special runtime variables and functions](#)
- [4 How to script inside the runtime environment](#)

### 9.40.2 Definition

The runtime environment is an extension to the normal AviSynth script execution environment that is available to scripts executed by [runtime filters](#). Its basic characteristic is that the runtime filters' scripts are evaluated (compiled) **at every frame**. This allows for complex video processing that it would be difficult or impossible to perform in a normal AviSynth script.

Let's now look at the above definition in more detail. The runtime environment is:

## Avisynth 2.5 Selected External Plugin Reference

1. An environment, that is a set of available local and global variables and filters / functions to be used by the script.
2. An extension to the normal AviSynth script execution environment; that is there are additional variables and functions available to runtime scripts.
3. Available to scripts executed by runtime filters **only**, that is scripts inside it are executed only during the AviSynth "runtime" (the frame serving phase).

The last is the biggest difference. Normal script code is parsed and evaluated (compiled) at the start of the script's [execution](#) (the parsing phase) in a linear fashion, from top to bottom; the result is the creation of a [filter graph](#) that is used by AviSynth to serve frames to the host video application. Runtime script code is executed *after* the parsing phase, when frames are served. Moreover it is compiled on every frame requested and only for that specific frame, in an event-driven fashion.

**Note:** Audio is *not* handled by the runtime environment; it is passed through untouched. This also means that you cannot modify clip audio with runtime filters.

### 9.40.3 Runtime filters

The following [filters](#) are the basic set of the so-called "runtime filters":

- [ConditionalFilter](#): Selects, on every frame, from one of two (provided) filters based on the evaluation of a conditional expression.
- [ScriptClip](#): Compiles arbitrary script code on every frame and returns a clip.
- [FrameEvaluate](#): Compiles arbitrary script code on every frame but the filter's output is ignored.
- [ConditionalReader](#): Loads input from an external file to a selectable [variable](#) on every frame.

In addition, the [WriteFile](#) filter can also be considered a runtime filter, because it sets the special variables set by all runtime filters before evaluating the expressions passed to it, which can use the special [runtime functions](#).

### 9.40.4 Special runtime variables and functions

All runtime filters set and make available to runtime scripts the following special variables.

- `last`: The clip passed as argument to the filter
- `current_frame`: The frame number (ranging from zero to input clip's [Framecount](#) minus one) of the requested frame.

All the above variables are defined at the [top-level script local scope](#). That is you read and write to them in a runtime script as if they were variables that you declare at the script level.

Runtime scripts can also call a rich set of [special functions](#) that provide various pieces of information for the current frame of their input clip.

## 9.40.5 How to script inside the runtime environment

Using the runtime environment is simple: you use one of the [runtime filters](#) and supply it with the needed arguments. Among them, two are the most important:

- The input clip
- The runtime script

The latter is supplied as a string argument which contains the AviSynth script commands. Scripts can contain many statements (you can use a multiline string), local and global variables and function calls, as well as special [runtime functions](#) and [variables](#). In general all statements of the AviSynth [syntax](#) are permitted, but attention must be paid to avoid overly complex scripts because this has a penalty on speed that is paid at **every frame**. See the runtime section of [scripting reference's performance considerations](#) for details.

Let's see a few examples:

TODO . . . EXAMPLES

---

Other points:

- [Runtime functions](#) and variables, such as `current_frame`, `AverageLuma()`, etc., are available only at the runtime script's scope. To use them in a function you must pass them as arguments.
  - You can include an explicit `return` statement in your runtime script to return a clip that is not contained in the special `last` variable.
- 

Back to [AviSynth Syntax](#).

\$Date: 2008/12/07 15:46:17 \$

## 9.41 AviSynth Syntax – Script variables

This page shows how to use *variables* to store intermediate values for further processing in a script. It also describes the types of data that scripts can manipulate, and how *literals* (constants) of those types are written.

A *variable name* can be a character string of practically any length (more than 4000 characters in Avisynth 2.56 and later) that contains (English) letters, digits, and underscores (`_`), but no other characters. The name cannot start with a digit.

You may use characters from your language system codepage (locale) in strings and file names (ANSI 8 bit only, not Unicode).

A variable's placement in an *expression* is determined by the

[AviSynth Syntax](#).

Variables can have a value of one of the following *types*:

- clip

A video clip containing video and / or audio. A script must return a value of this type.

## Avisynth 2.5 Selected External Plugin Reference

- string

A sequence of characters representing text. String literals are written as text surrounded either by "quotation marks" or by ""three quotes"". The text can contain any characters except the terminating quotation mark or triple-quote sequence.

The manual used to mention *TeX-style quotes*, but it has been confirmed that AviSynth doesn't work this way since v1.03. If you need to put a quotation mark inside a string, you need to use [Python-style](#) ""three quotes"". For example:

```
Subtitle("""AVISynth is as they say "l33t".""" )
```

Alternatively, you can use Windows extended-ASCII curly-quotes inside the string instead of straight quotes to get around this limitation.

- int

An integer. An integer literal is entered as a sequence of digits, optionally with a + or - at the beginning. The value can be given in *hexadecimal* by preceding them with a "\$" character. For example \$FF as well as \$ff (case does not matter) are equal to 255.

- float

A floating-point number. Literals are entered as a sequence of digits with a decimal point (.) somewhere in it and an optional + or -. For example, +1. is treated as a floating-point number. Note that exponent-style notation is **not** supported.

- bool

Boolean values must be either *true* or *false*. In addition they can be written as "yes" or "no", but you should avoid using these in your scripts (they remain for compatibility purposes only).

- val

A generic type name. It is applicable only inside a [user defined script function's](#) argument list, in order to be able to declare an argument variable to be of *any* type (int, float, bool, string, or clip). You must then explicitly test for its type (using the [boolean functions](#)) and take appropriate actions.

There is another type which is used internally by Avisynth – the void or 'undefined' type. Its principal use is in conjunction with optional function arguments. See the [Defined\(\)](#) function.

Variables can be either local (bound to the local scope of the executing script block) or global. Global variables are bound to the global script environment's scope and can be accessed by all [Internal functions](#), [User defined script functions](#), [runtime environment](#) scripts and the main script also.

To define and / or assign a value to a global variable you must precede its name with the keyword `global` at the left side of the assignment. The keyword is not needed (actually it is not allowed) in order to read the value of a global variable. Examples:

```
global canvas = BlankClip(length=200, pixel_type="yv12")
global stroke_intensity = 0.7
...
global canvas = Overlay(canvas, pen, opacity=stroke_intensity, mask=brush)
```

To declare a variable, simply type the variable name, followed by '=' (an equals sign), followed by its initial value. The type must not be declared; it is inferred by the value assigned to it (and can actually be changed by subsequent assignments). The only place where it is allowed (though not strictly required) to declare a variable's type is in [user defined script function's](#) argument lists. Examples:

```
b = false      # this declares a variable named 'b' of type 'bool' and initializes it to 'false'
x = $100      # type int (initial value is in hexadecimal)
y = 256       # type int (initial value is in decimal)
global f = 0.0 # type float declared globally
...
function my_recolor_filter(clip c, int new_color, float amount, val "userdata") { ... }
```

\$Date: 2009/09/12 20:57:20 \$

## 9.42 AviSynth Syntax – User defined script functions

### 9.42.1 Definition and Structure

You can define and call your own functions in AviSynth scripts as shown below. The function can return any clip or variable type. An user defined script function is an independent block of script code that is executed each time a call to the function is made in the script. An example of a simple user defined script function (here a custom filter) immediately follows:

```
function MuteRange(clip c, int fstart, int fend)
{
    before = c.Trim(0, -fstart)
    current = c.Trim(fstart, fend)
    after = c.Trim(fend + 1, 0)
    audio = Dissolve(Dissolve(before, current.BlankClip, 3), after, 3)
    return AudioDub(c, audio)
}
```

User defined script functions start with the keyword `function` followed by the function name. The name of a script function follows the same naming rules as [script variables](#).

Immediately after the name, the function's argument list follows. The list (which can be empty) consists of (expected argument's type – argument's name) pairs. Each argument's [type](#) may be any of those supported by the scripting language.

```
function MuteRange(clip c, int fstart, int fend)
```

Then comes the function body, ie the code that is executed each time the function is called. The arguments are accessed within the function body by their names. The function body is contained within an opening and closing brace pair { ... }.

```
{
    before = c.Trim(0, -fstart)
    current = c.Trim(fstart, fend)
    after = c.Trim(fend + 1, 0)
    audio = Dissolve(Dissolve(before, current.BlankClip, 3), after, 3)
    return AudioDub(c, audio)
}
```

## Avisynth 2.5 Selected External Plugin Reference

At the end of the function body a `return` statement which returns the final value calculated from the arguments and the function's body code is placed.

```
return AudioDub(c, audio)
```

It should be noted that unlike other languages where multiple return statements are allowed inside the function body, in AviSynth functions contain a *single* return statement. This is because the language does not support branching (i.e. compound block statements).

### 9.42.2 Facts about user defined script functions

- Functions can take up to sixty arguments and the return value can be of any type supported by the scripting language (clip, int, float, bool, string).
- Although not recommended practice, an argument type may be omitted, and will default to `val`, the generic type.
- If the function expects a video clip as its first argument, and that argument is not supplied, then the clip in the special `last` variable will be used.
- Functions support *named arguments*. Simply enclose an argument's name inside double quotes to make it a named argument. Note that after doing so the following apply:
  1. All subsequent arguments in the argument list must be made named also.
  2. **A named argument is an optional argument**, that is, it need not be supplied by the caller.
  3. When a function is called, any optional argument which has *not* been provided is set to a value which has the void ('undefined') type. This does not mean that its value is random garbage – simply that its type is neither clip, int, float, bool or string and so it has *no* usable value.
  4. Normally, you should use the [Defined](#) function to test if an optional argument has an explicit value, or the [Default](#) function, which combines the Defined test with the delivery of a default value if appropriate.
  5. A void ('undefined') value can be passed on to another function as one of its optional arguments. This is useful when you want to write a wrapper function that calls another function, preserving the same defaults.
- Functions always produce a new value and never modify an existing one. What that means is that all arguments to a function are passed "by value" and not "by reference"; in order to alter a variable's value in AviSynth script language you must assign to it a new value.
- Functions can call other functions, *including themselves*. The latter is known as recursion and is a very useful technique for creating functions that can accomplish complex tasks.
- Local function variables mask global ones with the same name inside the function body. For example, if you define in a function a local variable `myvar` by assigning to it a value, then you cannot read the global `myvar` anymore inside this function.
- The above is also true for arguments, since from the perspective of a function arguments are initialized local variables.

### 9.42.3 Related Links

- [Shared functions](#). An ever growing collection of shared script functions created by the members of the AviSynth community.

{TEMP: <http://www.avisynth.org/ShareFunctions>}

- [Conditional filters and script functions](#). A collection of highly useful conditional filters implemented as user defined script functions.

{TEMP: <http://www.avisynth.org/ExportingSingleImage>,  
<http://www.avisynth.org/HowToApplyFilterToManySingleFrames> :: Perhaps make decent functions from the last two?}

\$Date: 2009/09/12 20:57:20 \$

## 9.43 Troubleshooting

### 9.43.1 Contents

[1 Installation problems](#)

[2 Other problems](#)

[2.1 Write Simple](#)

[2.2 Always check parameters](#)

[2.3 Test scripts using Virtualdub](#)

[2.4 Go through the script step by step](#)

[2.5 Check your autoloading plugin directory for files](#)

[2.6 Use conservative image sizes](#)

[2.7 Finally check the AviSynth O&A](#)

[2.8 Reporting bugs / Asking for help](#)

#### 9.43.1.1 Installation problems

If you got problems getting AviSynth to work at all, try the following script:

[Version\(\)](#)

and open it in Windows Media Player 6.4 (it is a file "mplayer2.exe" located in "C:\Program Files\Windows Media Player", other versions of WMP will not work). If you see a video with a message with Avisynth version and Copyright, then it is installed properly.

If that doesn't work, you can try the following:

- Empty the plugin folder of AviSynth: autoloading scripts (\*.avsi) or some filters can cause this ([see here](#)).
- Install codecs, in particular [Huffyuv](#): it can be that there is no decoder present which can decode your video.
- If you use an encoding package (like DVD2SVCD, GKnot, DVX, ...) make sure that you use the version of AviSynth that came with that package: it might be that new versions of AviSynth are not compatible with the package. Try to get support from the package developers.

## Avisynth 2.5 Selected External Plugin Reference

- Reinstall AviSynth: it might be that something went wrong with the installation. If you tried playing with new beta version, reinstall a stable release.
- If all of the above doesn't help drop a post in the [Doom9 Forums](#).

### 9.43.1.2 Other problems

Creating scripts with AviSynth is not always easy, and sometimes AviSynth produces very strange results. This is a little guide to help you figure out the most common errors.

### 9.43.1.3 Write Simple

If AviSynth produces strange results, try simplifying your script. Try splitting up your script into as many lines as possible. This will help you identify your problem. For example:

```
video = AviSource("file23.avi").ConvertToYUY2().Trim(539,8534)
return AudioDub(Blur(video,1.5).Reduceby2().Bilinearrrresize(512,384),Wavsource("file23.wav").Am
```

is not as readable as

```
AviSource("file23.avi")
ConvertToYUY2()
Trim(539, 8534)
Blur(1.5)
Reduceby2()
Bilinearrrresize(512, 384)
AudioDub(Wavsource("file23.wav"))
AmplifyDB(4.5)
```

Furthermore it has the advantage, that you more easily:

- Comment out a single command (line). This is good for testing the effect of a filter.
- Get the proper position (line) of problem command, if there is a syntax error.
- Put in a "return last" at some position in the script. This will output the video at that place in the filterchain. So by varying the place of the "return last" you can check up to which line the video is correct.
- Get an overview of the "flow" of the script. (Is it a good thing that the [Trim](#) command only affects the video in the clip above?)

### 9.43.1.4 Always check parameters

If you have a filter that gives you unexpected results, try using it with the simplest parameters. Always check the internal filters either on AviSynth homepage or in the documentation that came along with your AviSynth.

Be sure you use the same type of parameters as the ones described in the documentation. The most common error in this case is related to the first parameter in all filters, "clip". Be sure you understand how "implicit last" works. If you do not have a "last clip", most filters will fail with an "Invalid parameter" error.

"Filter does not return a clip" is reported if the output of your last filter is put into a variable, and there isn't any "last clip". For instance:

```
video = AviSource("file.avi")
audio = WavSource("file.wav")
combined = AudioDub(video, audio)
```



will fail. This can be solved by:

```
video = AviSource("file.avi")
audio = WavSource("file.wav")
AudioDub(video, audio)
```

where 'last' now contains a clip, or:

```
video = AviSource("file.avi")
audio = WavSource("file.wav")
combined = AudioDub(video, audio)
return combined
```

where the variable is returned, or even:

```
video = AviSource("file.avi")
audio = WavSource("file.wav")
return AudioDub(video, audio)
```

### 9.43.1.5 Test scripts using Virtualdub

Always use [Virtualdub](#) or even better [VirtualDubMod](#) to test your scripts. This is what all AviSynth functionality is tested against (by its developers). AviSynth does of course work with other programs, but if you get errors in other applications it's most likely not an AviSynth problem, but a limitation within the software you are using.

These limitations are mostly linked to:

- Color format problems. The application you are using does not support the color format you are using as script output.
- Size problems. Some programs does not accept all sizes of images.

### 9.43.1.6 Go through the script step by step

As mentioned in "Write Simple" it is always a good thing to test every step of your script, if there are problems.

You can comment out a filter (filters) by placing a '#' in front of the line (or before filter). That way it (and all rest of the line) will be ignored by AviSynth.

You can put in a "return last" or "return myvariable" any place in the script.

At any place in the script you can add the [Info\(\)](#) filter to get information about the image and sound at the current stage of the filtering.

### 9.43.1.7 Check your autoloading plugin directory for a files

Plugins autoloading usually works fine, but you must NOT put here:

- any plugins for incompatible AviSynth versions (e.g. old 2.0.x).
- special LoadPluginEx.DLL plugin (from WarpSharp package) used for loading of old 2.0 plugins.
- AviSynth C–plugins which use AviSynth C API instead of regular interface.

## Avisynth 2.5 Selected External Plugin Reference

- too many AviSynth plugins (this 50 plugins auto prescan load limit is removed in v2.57 though).
- any other DLL files (usually it is safe, but is not recommended).

You must also remember, that all AVSI files in your plugin–directory are automatically included in your script. This is a feature, to allow you to include your own (or borrowed) functions, without have to copy/paste them into every script.

*Notes. In old AviSynth versions (up to 2.0.7) all AVS files in your plugin–directory were automatically included in your script. This also means that if you copy any sample scripts into your plugin directory they will always be included, and may generate errors (in old versions!).*

In general, any AVSI (early AVS) file whose commands are not wrapped into functions will be problematic.

All other file formats besides AVSI and DLL files are ignored, so you can safely leave your documentation there.

How to empty plugin dir? Simply create some subfolder (e.g. "hide") and move all (or some) files there.

Remember some files (DirectShowSource.dll, TCPDeliver.dll plugins, ColorRGB.avsi) are part of AviSynth (since v2.56).

### 9.43.1.8 Use conservative image sizes

If you have problems with distorted images, try using conservative frame sizes. That means, use sizes, where height and width are always divisible by 16. Using image sizes that are not divisible by 2 is in many cases problematic, and should always be avoided.

If you do however find that there is a problem with certain sizes of images, please submit a bug–report. See below how to do that.

### 9.43.1.9 Finally check the AviSynth Q&A

If you still got problems (loading scripts in certain encoders, or colorspace errors) have a look at the AviSynth Q&A, especially [Q2.4: Problems when Encoder X reads AVS–files ?](#) Be also sure to check [Q1.4: What are the main bugs in these versions ?](#) in the FAQ.

### 9.43.1.10 Reporting bugs / Asking for help

We will need many informations to be able to help you. If you don't supply us with that, there is a good chance that we won't be able to help you or locate the error.

Be sure to **always** include:

- AviSynth version. (and date of beta, if not a SourceForge final release)
- The simplest possible script for recreating the error.
- The EXACT error message you get.
- VirtualDub (Mod) version.
- All file information from VirtualDub / File / File Information.
- Used plugin versions.
- Codecs and image sizes of input material.

Bug reports should be submitted at the [SourceForge Project page](#). Be sure to check if there is already a bug submitted similar to yours – there might just be. Errors in external plugins shouldn't be reported here, but to the author of the filter.

A very good place to get help is the [Doom9 Forums](#). Be sure to search the forum before asking questions. Many topics have been covered there! – Then enter into the discussion.

\$Date: 2009/09/12 20:57:20 \$

## 9.44 Advanced Topics

blabla

### 9.44.1 Interlaced and field-based video

Currently (v2.5x and older versions), AviSynth has no interlaced flag which can be used for interlaced video. There is a field-based flag, but contrary to what you might expect, *this flag is not related to interlaced video*. In fact, all video (progressive or interlaced) is frame-based, unless you use AviSynth filters to change that. There are two filters who turn frame-based video into field-based video: [SeparateFields](#) and [AssumeFieldBased](#). More information about this can be found [here](#).

### 9.44.2 Color format conversions, the Chroma Upsampling Error and the 4:2:0 Interlaced Chroma Problem

The *Chroma Upsampling Error* is the result of your video is upsampled incorrectly (interlaced YV12 upsampled as progressive or vice versa). Visually, it means that you will often see gaps on the top and bottom of colored objects and "ghost" lines floating above or below the objects. The *4:2:0 Interlaced Chroma Problem* is the problem that 4:2:0 Interlaced itself is flawed. The cause is that frames which show both moving parts and static parts are upsampled using interlaced upsampling. This result in chroma problems which are visible on bright-colored diagonal edges (in the static parts of the frame). More about these issues can be found [here](#).

### 9.44.3 Colorspace Conversions

About the different RGB <-> YUV [color conversions](#). <in progress>

### 9.44.4 Wrong levels and colors upon playback

When playing back video content, several issues might go wrong. The levels could be wrong, resulting in washed out colors (black is displayed as dark gray and white is displayed as light gray). This is described in more detail [here](#). The other issue is a slight distortion in color (which often looks like a small change in brightness) and this is described [here](#).

### 9.44.5 AviSynth, variable framerate (vfr) video and Hybrid video

There are two kinds of video when considering framerate. Constant framerate (cfr) video and variable framerate (vfr) video. For cfr video the frames have a constant duration, and for vfr video the frames have a non-constant duration. Many editing programs (including VirtualDub and AviSynth) assume that the video has cfr. One of the reasons is that avi doesn't support vfr. This won't change in the near future for [various](#)

## Avisynth 2.5 Selected External Plugin Reference

[reasons](#). Although the avi container doesn't support vfr, there are several contains (mkv, mp4 and wmv for example) which do support vfr.

It's important to realize that in general video is intrinsically cfr (at least in the capping video or ripping dvds arena). There is one exception where converting to vfr is very useful, which is hybrid video. Hybrid video consists of parts which are interlaced/progressive NTSC (29.97 fps) and FILM (which is telecined to 29.97 fps). When playing hybrid video the NTSC part (also called video part) is played back at 29.97 fps and the telecined part at 23.976 fps. Examples of hybrid video include some of the anime and Star Trek stuff.

More info about creating vfr video and opening it in AviSynth can be found [here](#).

### 9.44.6 Importing your media in AviSynth

A lot of media formats (video, audio and images) can be imported into AviSynth by using one of AviSynth's internal filters, specific plugins or DirectShowSource in combination with the appropriate DirectShow filters. It is not always trivial to import your media into AviSynth, because there are often many ways to do so, and for each way you need to have some specific codecs installed. [This document](#) describes which formats can be imported in AviSynth and how they should be imported. Also a short summary is included about how to make graphs (graphs of appropriate DirectShow filters which can be used to play you media file) in Graphedit and how to open the graphs in AviSynth.

### 9.44.7 Resizing

...

§Date: 2009/09/12 20:57:20 §

work under construction.

Should cover RGB→YUV conversions and lumarange scaling/preservation and when to use which conversion.

## 9.45 Color conversions

coefficients	Rec.601	Rec.709	FCC
Kr : Red channel coefficient	0.299	0.2125	0.3
Kg : Green channel coefficient	0.587	0.7154	0.59
Kb : Blue channel coefficient	0.114	0.0721	0.11

$(0.0 \leq [Y,R,G,B] \leq 1.0) ; (-1.0 < [U,V] < 1.0)$

$$K_g = 1 - K_r - K_b$$

$$Y = K_r * R + K_g * G + K_b * B$$

$$V = (R - Y) / (1 - K_r) = R - G * K_g / (1 - K_r) - B * K_b / (1 - K_r)$$

$$U = (B - Y) / (1 - K_b) = -R * K_r / (1 - K_b) - G * K_g / (1 - K_b) + B$$

$$\begin{aligned} R &= Y + V*(1-Kr) \\ G &= Y - U*(1-Kb)*Kb/Kg - V*(1-Kr)*Kr/Kg \\ B &= Y + U*(1-Kb) \end{aligned}$$

### 9.45.1 Converting to programming values

**YUV [0,255] <-> RGB [0,255]** ( $0 \leq [r,g,b] \leq 255$ ,  $0 \leq y \leq 255$ ,  $0 < [u,v] < 255$ )

$$\begin{aligned} y &= Y * 255 \\ v &= V * 127.5 + 128 \\ u &= U * 127.5 + 128 \\ r &= R * 255 \\ g &= G * 255 \\ b &= B * 255 \end{aligned}$$

Substituting (Y,V,U,R,G,B) in the equations above and multiplying with 127.5 and respectively 255 gives

$$\begin{aligned} y &= Kr*r + Kg*g + Kb*b \\ v - 128 &= 0.5*(r - y)/(1-Kr) = 0.5 * r - 0.5 * g * Kg/(1-Kr) - 0.5 * b * Kb/(1-Kr) \\ u - 128 &= 0.5*(b - y)/(1-Kb) = -0.5 * r * Kr/(1-Kb) - 0.5 * g * Kg/(1-Kb) + 0.5 * b \\ r &= y + 2*(v-128)*(1-Kr) \\ g &= y - 2*(u-128)*(1-Kb)*Kb/Kg - 2*(v-128)*(1-Kr)*Kr/Kg \\ b &= y + 2*(u-128)*(1-Kb) \end{aligned}$$

**YUV [16,235] <-> RGB [0,255]** ( $0 \leq [r,g,b] \leq 255$ ,  $16 \leq y \leq 235$ ,  $16 \leq [u,v] \leq 240$ )

$$\begin{aligned} y &= Y * 219 + 16 \\ u &= U * 112 + 128 \\ v &= V * 112 + 128 \\ r &= R * 255 \\ g &= G * 255 \\ b &= B * 255 \end{aligned}$$

## 9.46 References

[http://www.poynton.com/notes/colour\\_and\\_gamma/ColorFAQ.html](http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html)  
 ITU BT.601 ...  
 ITU BT.709 ...

⌘Date: 2006/03/26 18:06:55 ⌘

# 10 AviSynth, variable framerate (vfr) video and hybrid video

There are two kinds of video when considering framerate, constant framerate (cfr) video and variable framerate (vfr) video. For cfr video the frames have a constant duration, and for vfr video the frames have a non-constant duration. Many editing programs (including VirtualDub and AviSynth) assume that the video is cfr, partly because avi doesn't support vfr. This won't change in the near future for [various reasons](#). Although the avi container doesn't support vfr, there are several containers (mkv, mp4 and wmv/asf for example) which do support vfr.

Hybrid video is commonly defined as being a mix of pulled-down material and non-pulled-down material (where the pulldown can be of fields, as in standard 3:2 pulldown, or full frames). It's not relevant whether the pulldown is hard (the fields/frames are duplicated before the encoding) or soft (adding the appropriate flags in the stream which indicate which fields/frames should be duplicated during playback). So, it can be either cfr or vfr. Thus hybrid video is simply video with different base framerates (for example 8, 12, and 16 fps at which anime is often drawn). The base framerate is the rate before any pulldown. What makes hybrids challenging is the need to decide what final framerate to use.

## 10.1 Table of contents

- [Variable framerate and hybrid video](#)
- [How to recognize vfr content \(mkv/mp4\)](#)
- [Opening hybrid video \(MPEG-2\) in AviSynth and re-encoding](#)
  - ◆ [encoding to cfr – 23.976 fps or 29.97 fps](#)
  - ◆ [encoding to cfr – 120 fps](#)
  - ◆ [encoding to vfr – mkv](#)
  - ◆ [encoding to vfr – mp4](#)
  - ◆ [summary](#)
- [Opening non MPEG-2 hybrid video in AviSynth and re-encoding](#)
  - ◆ [opening avi vfr content – 120 fps – in AviSynth](#)
  - ◆ [opening non-avi vfr content in AviSynth](#)
  - ◆ [re-encoding 120 fps video](#)
  - ◆ [converting vfr to cfr avi for AviSynth](#)
  - ◆ [encoding to MPEG-2 vfr video](#)
- [Audio synchronization](#)
- [References](#)

## 10.2 Variable framerate and hybrid video

It's important to understand that usually video is cfr. There is one example where converting to vfr can be very useful, which is hybrid video. Hybrid video is video with different base framerates (for example 8, 12, and 16 fps at which anime is often drawn). The most common example of hybrid video consists of parts that are interlaced/progressive NTSC (29.97 fps) and other parts which are FILM (telecined from 23.976 fps to 29.97 fps). For soft pulldown, the NTSC part (also called video part) is played back at 29.97 fps and the telecined part also by duplicating fields (to go from 23.976 fps to 29.97 fps). For hard pulldown, it is played back at 29.97 fps without adding any fields. Other examples of hybrid video include many of the modern anime TV Series, many of the Sci-Fi TV Series, such as Stargate: SG1, Star Trek: TNG, and Babylon 5), and many of the "Making Of" documentaries included on DVD.

Examples of hybrid video include many of the modern anime TV Series, many of the Sci-Fi TV Series (such as Stargate: SG1, Star Trek: TNG, and Babylon 5), and many of the "Making Of" documentaries included on DVD.

The TIVTC package is designed to work with hybrid video losslessly, while the Decomb package has routines to convert to cfr via blending.

### 10.3 How to recognize vfr content (mkv/mp4)

Here are some ways to determine if the mkv/mp4 is vfr:

mpeg-2: DGIndex will report a Film/Video percentage, which can tell you much hybrid content a soft-pulldowned file has. It will not work with hard pulldown, and isn't always accurate if hard/soft are mixed.

mkv: get timecodes file using [mkv2vfr](#) to check this.

mp4: this can be found out by using mp4dump (from the [MPEG4 tools by MPEG4ip package](#)). Open a dos prompt and type (using appropriate paths)

```
mp4dump -verbose=2 holly_xvid.mp4 > log.txt
```

Open the log file, and look for output like this (look up the stts atom to figure out the length of each frame):

```
type stts
  version = 0 (0x00)
  flags = 0 (0x000000)
  entryCount = 41 (0x00000029)
  sampleCount = 3 (0x00000003)
  sampleDelta = 1000 (0x000003e8)
  sampleCount[1] = 1 (0x00000001)
  sampleDelta[1] = 2000 (0x000007d0)
  sampleCount[2] = 3 (0x00000003)
  sampleDelta[2] = 1000 (0x000003e8)
  sampleCount[3] = 1 (0x00000001)
  sampleDelta[3] = 2000 (0x000007d0)
  etc ...
```

*sampleDelta* indicates how long the frames get displayed and *sampleCount* tells how many frames. Thus on the example above:

3 frames are displayed with length 1000

1 frame are displayed with length 2000

3 frames with length 1000

1 frame with length 2000

etc ...

The time values are not seconds, but "ticks", which you have to calculate into seconds via the "timescale" value. This "timescale" is stored in timescale atom for the video track (make sure that you look at the right timescale for your track, cause every track has its own timescale). Look for output like this:

```
type mdia
  type mdhd
  ...
```

## Avisynth 2.5 Selected External Plugin Reference

```
timeScale = 24976 (0x00006190)
duration = 208000 (0x00032c80)
language = 21956 (0x55c4)
reserved = <2 bytes> 00 00
```

In this example the timeScale is 24976. Most of the frames have a length of 1000.  $1000/24976 = 0.04$  which means each frame of the first 3 gets displayed with a length of 0.04 seconds, which is the equivalent to 25 fps ( $1/25 = 0.04$ ). The next frame has a length of 2000.  $2000/24976 = 0.08$  which means that it is displayed with a length of 0.08, which is the equivalent to 12.5 fps ( $1/12.5 = 0.08$ ). etc ...

The log file above comes from a video which is in fact hybrid.

### 10.4 Opening MPEG-2 hybrid video in AviSynth and re-encoding

Assuming you have hybrid video, there are several ways to encode it. They are listed below. The first method is to convert it to cfr video (either 23.976 or 29.97 fps). The second one is to encode it at 120 fps using avi and dropped frames (where duplicate frames are dropped upon playback). The third one is to create true vfr using the mkv or mp4 container.

#### 10.4.1 encoding to cfr (23.976 fps or 29.97 fps)

If we choose the video rate, the video sequences will be OK, but the FILM sequences will not be decimated, appearing slightly jumpy (due to the duplicated frames). On the other hand, if we choose the FILM rate, the FILM sequences will be OK, but the video sequences will be decimated, appearing jumpy (due to the "missing" frames). Additionally, when encoding to 29.97 fps, you will get lower quality for the same file size, because of the 25% greater number of frames. It's a tough decision which to choose. If the clip is mostly FILM you might choose 23.976 fps, and if the clip is mostly video you might choose 29.97 fps. The source also is a factor. If the majority of the video portions are fairly static "talking heads", for example, you might be able to decimate them to 23.976 fps without any obvious stutter on playback.

When you create your d2v project file you will see whether the clip is mostly video (NTSC) or FILM (in the information box). However, many of these hybrids are encoded entirely as NTSC, with the film portions being "hard telecined" (the already telecined extra fields having also been encoded) so you'll have to examine the source carefully to determine what you have, and how you wish to treat it.

The AviSynth plugins Decomb and TIVTC provide two special decimation modes to better handle hybrid clips by blending. This will eat bitrate quickly, but it appears very smooth. Here is a typical script to enable this mode of operation:

```
Telecide(order=0, guide=1)
Decimate(mode=X) # tweak "threshold" for film/video detection
```

or

```
TFM(mode=1)
TDecimate(mode=0, hybrid=X) # tweak "vidThresh" for film/video detection
```

There are 2 factors that enable Decimate to treat the film and nonfilm portions appropriately. First, when Telecide declares guide=1, it is able to pass information to Decimate about which frames are derived from film and which from video. For this mechanism to work, Decimate must immediately follow Telecide.



## Avisynth 2.5 Selected External Plugin Reference

Clearly, the better job you do with pattern locking in Telecide (by tweaking parameters as required), the better job Decimate can do.

The second factor is the threshold. If a cycle of frames is seen that does not have a duplicate, then the cycle is treated as video. The threshold determines what percentage of frame difference is considered to be a duplicate. Note that `threshold=0` disables the second factor.

Make sure to get the field order correct – DVDs are generally `order=1`, and captured video is generally `order=0`. The included `DecombTutorial?.html` explains how to determine the field order.

### *Mostly Film Clips (mode=3)*

When the clip is mostly film, we want to decimate the film portions normally so they will be smooth. For the nonfilm portions, we want to reduce their frame rate by blend decimating each cycle of frames from 5 frames to 4 frames. Video sequences so rendered appear smoother than when they are decimated as film. Set Decimate to `mode=3`, or TDecimate to `hybrid=1` for this behavior.

Another IVTC was developed specifically to handle hybrid material without blended frames: SmartDecimate. While you do get "clean" frames as a result, it also may play with slightly more stutter than does Decomb's result. A typical script might go:

```
B = TDeint(mode=1) # or KernelBob(order=1)
SmartDecimate(24, 60, B)
```

In order to keep the result as smooth playing as possible, it will insert the "Smart Bobbed" frames from time to time.

### *Mostly Video Clips (mode=1)*

When the clip is mostly video, we want to avoid decimating the video portions in order to keep playback as smooth as possible. For the film portions, we want to leave them at the video rate but change the duplicated frames into frame blends so it is not so obvious. Set Decimate to `mode=1`, or TDecimate to `hybrid=3` for this behavior.

In this case you may also consider leaving it interlaced and encoding as such, especially if you'll be watching on a TV later.

## 10.4.2 encoding to cfr – 120 fps

**This is the most widely compatible option (sentence can be removed ...).** For this you'll need [TIVTC and avi tc](#). Start by creating a [decimated avi with timecodes.txt](#), but skip the muxing. Then open tc-gui's tc2cfr tab and add your files or use this command line:

```
tc2cfr 120000/1001 c:\video\video.avi c:\video\timecodes.txt c:\video\video-120.avi
```

Then mux with your audio. This works because tc2cfr creates an avi with drop frames filling in the extra space with drop frames to create a smooth 120fps avi.

### 10.4.3 encoding to vfr (mkv)

First download [mkvtoolnix](#). We will use this to mux our video into the MKV container WITH a timecode adjustment file. Make sure that you have the latest version (1.6.0 as of this writing), as older ones read timecodes incorrectly.

There are several AviSynth plugins that you can use to generate the VFR video and required timecode file. An example is given below using the [Decomb521VFR](#) plugin. Another alternative is the TDecimate plugin contained in the [TITVC](#) package. See their respective documentations to learn more about tweaking them.

The [DeDup](#) plugin removes duplicate frames but does not change the framerate (leaving jerky video if not decimated first), so it won't be included. It can still be used after either method by using their timecodes as input to DeDup.

#### Decomb521VFR:

Add this to your script:

```
Decomb521VFR_Decimate(mode=4, threshold=1.0, progress=true, timecodes="timecodes.txt", vfrstats
```

Open this script in VirtualDub, it will create the timecodes and stats files, then encode. It will seem to freeze at first, because it examines every frame on the first load.

#### *TITVC*

This is a 2-pass mode. Add this to your script:

```
TFM(mode=1, output="tfm.txt")
TDecimate(mode=4, output="stats.txt")
```

Open this and play through it in VirtualDub. Then close it, comment those lines out (or start a second script) and add:

```
TFM(mode=1)
TDecimate(mode=5, hybrid=2, dupthresh=1.0, input="stats.txt", tfmin="tfm.txt", mkvout="timecodes.txt
```

Load and encode.

#### *framerate*

If you're encoding to a specific size using a bitrate calculator, vfr decimation will mess up the calculations. To make them work again add these to your script:

Before decimation:

```
oldcount = framecount # this line must be before decimation
oldfps = framerate
```

End of script:

```
averagefps = (float(framecount)/float(oldcount))*oldfps
AssumeFPS(averagefps)
```

*muxing*

Now mux to MKV:

1. Open mmg.exe (mkvmerge gui)
2. Add your video stream file
3. Add your audio stream file
4. Click on the imported video track
5. Browse for the "timecodes.txt" timecode file
6. Click on the audio track
7. If your audio already needs a delay, set one
8. Start muxing

To play it you need a Matroska splitter. For AVC you will need [Haali's Splitter](#), but for ASP you can use it or [Gabest's Splitter](#).

### 10.4.4 encoding to vfr (mp4)

If you create a 120 fps avi with drop-frames, however, the mp4 muxed from it will remove them along with any n-vops the encoder creates, leaving vfr. A more laborous way is to encode multiple cfr avi files (some with 23.976 fps film and some with 29.97 fps video) and join them directly into one vfr mp4 file with mp4box and the -cat option.

A third, much easier, method is to encode using the MKV method and then processing the video with tc2mp4: more details on tc2mp4 can be found on the [\[Doom9 forums\]](#).

### 10.4.5 summary of the methods

Summing up the advantages and disadvantages of the above mentioned methods. When encoding to 23.976 or 29.97 fps the clip will be cfr (which editors like AviSynth and Virtualdub need), but it may look jumpy on playback due to duplicated or missing frames. That can be avoided with blending, but encoders can't work as well with that. When encoding to 120 fps using drop frames, the clip is cfr, not jumpy on playback, and very compatible. Encoding to mkv using true vfr (using timecodes) neither loses nor duplicates frames, however it is not nearly as broadly supported as AVI.

## 10.5 Opening non MPEG-2 hybrid video in AviSynth and re-encoding

It is possible to open vfr video in AviSynth without losing sync: DirectShowSource. The most common formats that support hybrid video (vfr) are **mkv**, **mp4**, **wmv**, and **rmvb**, and the methods below work for all of them; however, if the source is mkv, you can also use [mkv2vfr](#) and [AviSource](#).

### 10.5.1 opening non-avi vfr content in AviSynth

The best way to get all frames while keeping sync and timing is to convert to a common framerate, such as 120 fps for 24/30 (or rather 119.88). (Always use convertfps=true, which adds frames like ChangeFPS, or your audio *will* go out of sync.)

```
DirectShowSource("F:\Hybrid\vfr.mp4", fps=119.88, convertfps=true)
```

## Avisynth 2.5 Selected External Plugin Reference

You can also open it as 30p, which then has to be re-decimated but has less frames to deal with, or 24p, breaking any 30p sections:

Re-encoding to 23.976 or 29.97 fps:

```
DirectShowSource("F:\Hybrid\vfr.mkv", \
    fps=29.97, convertfps=true) # or fps=23.976
```

or

```
DirectShowSource("F:\Hybrid\vfr_startrek.mkv", \
    fps=119.88, convertfps=true)
FDecimate(29.97) # or FDecimate(23.976)
```

Another way is to find out the average framerate (by dividing the total number of frames by the duration in seconds) and use this rate in DirectShowSource. Depending on the duration of a frame, frames will be added or dropped to keep sync, and it's almost guaranteed to stutter. DirectShowSource will not telecine.

### 10.5.2 re-encoding 120 fps video

The easiest way to convert vfr sources back into vfr in AviSynth is by using [DeDup](#):

1st pass:

```
DupMC(log="stats.txt")
```

2nd pass:

```
DeDup(threshold=.1,maxcopies=4,maxdrops=4,dec=true,log="stats.txt",times="timecodes.txt")
```

TIVTC can also do this:

1st pass:

```
TFM(mode=0,pp=0)
TDecimate(mode=4,output="stats.txt")
```

2nd pass:

```
TFM(mode=0,pp=0)
TDecimate(mode=6,hybrid=2,input="stats.txt",mkvout="timecodes.txt")
```

Once you've encoded your file, mux back to mkv or 120 fps avi.

This will chop out all the duplicate frames directshowsource inserts, while keeping framecount and timing nearly identical. But do not use the timecode file from the input video, use the new one. They may not be identical. (Of course you can play with parameters if you want to use more of the functionality of dedup.)

### 10.5.3 converting vfr to cfr avi for AviSynth

You can avoid analysing and decimating by using special tools to get a minimal constant-rate avi to feed avisynth. After processing and re-encoding, use tc2cfr or mmg on the output with the original timecodes to

## Avisynth 2.5 Selected External Plugin Reference

regain vfr and full sync. (If you perform any kind of decimation or frame-rate change you'll have to edit the timecode file yourself, although dedup does have a timesin parameter.)

*avi*

[avi\\_tc](#) will create a timecode and normal video, if the avi uses drop frames and not n-vops or fully encoded frames. It also requires that no audio or secondary tracks are present. To use it, open tc-gui and add your file, or use the following command line:

```
cfr2tc c:\video\video-120.avi c:\video\video.avi c:\video\timecodes.txt 1
```

*mkv*

[mkv2vfr](#) extracts all video frames from Matroska to a normal AVI file and a timecode file. This will only work if the mkv is in vfw-mode. The command-line to use it is:

```
mkv2vfr.exe input.mkv output.avi timecodes.txt
```

### 10.5.4 encoding to MPEG-2 vfr video

<http://forum.doom9.org/showthread.php?t=93691>

I didn't look at it yet, so i can't give any comments/hints.

## 10.6 Audio synchronization

Several methods are discussed to encode your video (at 23.976, 29.97 or vfr video). You might wonder why your audio stays in sync regardless of the method you used to encode your video. Prior to encoding, the video and audio have the same duration, so they start out in sync. The following two situations might occur:

- you change the framerate of the stream by speeding it up or slowing it down (as is often done by PAL-FILM conversions). This implies that the duration of the video stream will change, and hence the audio stream will become out of sync.
- you change the framerate of your the stream by adding or removing frames. This **implies** that the duration of the video stream will remain the same, and hence the audio stream will be in sync.

If you encode the video stream at 23.976 or 29.97 fps (both cfr) by using Decimate(mode=3, threshold=1.0) or Decimate(mode=1, threshold=1.0), frames will be removed or added, and thus your audio stream will be in sync. By a similar reasoning the vfr encoding will be in sync.

Finally, suppose you open vfr video in AviSynth with DirectShowSource. Compare the following

```
DirectShowSource("F:\Hybrid\vfr_startrek.mkv", \  
  fps=29.97) # or fps=23.976
```

and

```
DirectShowSource("F:\Hybrid\vfr_startrek.mkv", \  
  fps=29.97, convertfps=true) # or fps=23.976
```

## Avisynth 2.5 Selected External Plugin Reference

The former will be out of sync since 24p sections are speeded up, and the latter will be in sync since frames are added to convert it to cfr.

### **To Do:**

- <http://forums.animesuki.com/showthread.php?t=34738>  
<http://forum.doom9.org/showthread.php?t=112199> tc2mp4, subs/Aegisub and ffmpegsource for timecode file.
- download [WMVTIMES.exe](#).
- subs also: <http://forum.doom9.org/showthread.php?t=135889&page=2>.
- how to determine whether a video (MP4) is vfr or not?:  
<http://forum.doom9.org/showthread.php?t=137899>.
- **Wilbert: I don't understand the comment about DeDup in "encoding to vfr (mkv)": need to investigate.**

## 10.7 References

Essential reading: [Force Film, IVTC, and Deinterlacing and more](#) (an article written by some people from at doom9).

Creating [120 fps video](#).

Documentation of [Decomb521VFR](#).

About [Decomb521VFR1.0](#) mod for automated Matroska VFR.

[Mkvextract GUI](#) by DarkDudae.

*Besides all people who contributed to the tools mentioned in this guide, the author of this tutorial (Wilbert) would like to thank bond, manono, tritical and foxyshadis for their useful suggestions and corrections of this tutorial.*

**\$Date: 2008/12/21 09:23:02 \$**

# 11 Importing media into AviSynth

## 11.1 Contents

### [1\) Loading clips into AviSynth](#)

#### [1.1\) Loading clips with video and audio into AviSynth](#)

#### [1.2\) Loading video clips into AviSynth](#)

#### [1.3\) Loading audio clips into AviSynth](#)

#### [1.4\) Loading images into AviSynth](#)

### [2\) Changing the merit of DirectShow Filters](#)

### [3\) Using GraphEdit to make graphs of DirectShow filters and how do I load these graphs in AviSynth](#)

## 11.2 Loading clips into AviSynth

Most video/audio formats can be loaded into AviSynth, but there are some exceptions like swf video, flv4 (VP6) and dvr-ms. If it is not possible to load a clip into AviSynth, you will have to convert it into some other format which can be loaded. Remember to choose a format for which you will have a minimal downgrade in quality as a result of the conversion.

In general there are two ways to load your video into AviSynth:

1. using an AviSynth filter or plugin which is designed to open some specific format.
2. using the [DirectShowSource](#) plugin.

Make sure that your clip contains maximal one video and or one audio stream (thus remove the subtitles and remove other video/audio streams). If you want to load a clip which contains both video and audio, you have two options:

- Demux the audio stream and load the streams separately in AviSynth.
- Try to load the clip in AviSynth. This might or might not work. For AVIs, make sure you have a good AVI splitter installed [[Gabest AVI splitter](#)]. (Yes, Windows comes with an own AVI splitter, which will work in most cases.)

When loading a clip into AviSynth it is advised to follow the following guidelines:

- When it is possible to load your clip into AviSynth using either AviSource or a specific plugin then do so, since this is more reliable than the two alternatives which are listed below.
- If the above fails, load your clip using the DirectShowSource plugin.
- If the above fails, convert your clip into a different format (into one which is supported by AviSynth).

For many formats it is explained how to load them in AviSynth, but if your format is not discussed here, there should be enough discussion of how to get you starting. *There are often multiple ways to load your format into AviSynth, so if one of them doesn't work, don't forget to try the other ones. As an example, suppose you got an AVI with 5.1 DTS, but it doesn't open as 5.1 with AviSource. Try other ways, like opening it with DirectShowSource (using AC3filter), or demux the audio and load the dts with NicAudio.*

## 11.2.1 1.1) Loading clips with video and audio into AviSynth

### 11.2.1.1 1.1.1) AVI with audio:

For loading your AVI with audio you need (1) a VfW ([Video for Windows](#)) codec to open (that is decode) your video in AviSynth and an ACM ([Audio Compression Manager](#)) codec to open your audio in AviSynth. For many video and audio format such codecs are available, but certainly not for all of them.

The number of different video formats which can be found in an AVI is pretty limited. A selection of them:

- MPEG-4 ASP (install XviD or DivX to decode this format).
- MPEG-4 AVC (to be more accurate; video encoded where a subset of the properties of MPEG-4 AVC has been used to create it can be placed into AVI; as far as I know there is no specific x264 VfW DEcoder so you need to open those files with DirectShowSource).
- DV (install the [Cedocida codec \[link to codec\]](#)).
- MJPEG: as far as I know there is no free VfW MJPEG decoder (ffdshow can decode them though).
- WMV (install the [WM9/VC-1 codecs](#)).
- [VP7](#).

A list of audio formats is given below:

audio: uncompressed WAV, CBR/VBR MP2 or CBR/VBR MP3:

There is an ACM codec for MP2 called [[QDesign](#)]. There are ACM codecs for MP3 [[Radium Codec](#)] or [[Lame ACM](#)].

audio: AC3/DTS:

There is an ACM codec for AC3 called [[AC3ACM](#)]. There is an ACM codec for AC3/DTS called valex ACM codec [[vac3acm](#)].

Examples:

```
AviSource("d:\xvid_dts.avi")
```

Get an AC3/DTS directshow filter like [AC3Filter](#) (make sure that downmixing is turned off, unless you want it to be downmixed or it is stereo/mono) and an [AVI-AC3/DTS splitter](#), or ffdshow (with DTS set to libdts). Use AC3Filter and create the script:

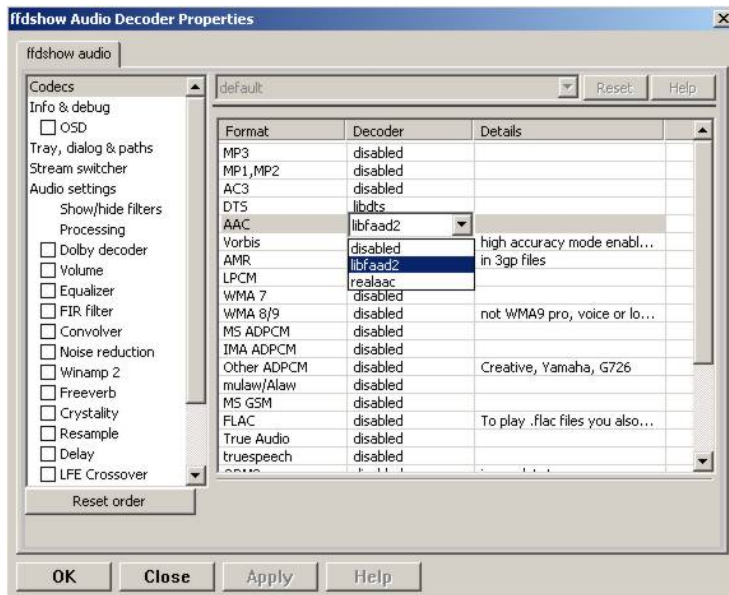
```
DirectShowSource("d:\xvid_dts.avi")
```

audio: AAC:

For AAC there is no reliable ACM codec, so it is not possible to load your clip with AAC with AviSource. Get an AAC directshow filter like CoreAAC (make sure downmixing is turned off, unless you want it to be downmixed) or ffdshow (with AAC set to libfaad2 or realaac). We used [ffdshow](#) here:



## Avisynth 2.5 Selected External Plugin Reference



use the script

```
DirectShowSource("d:\xvid_aac.avi")
```

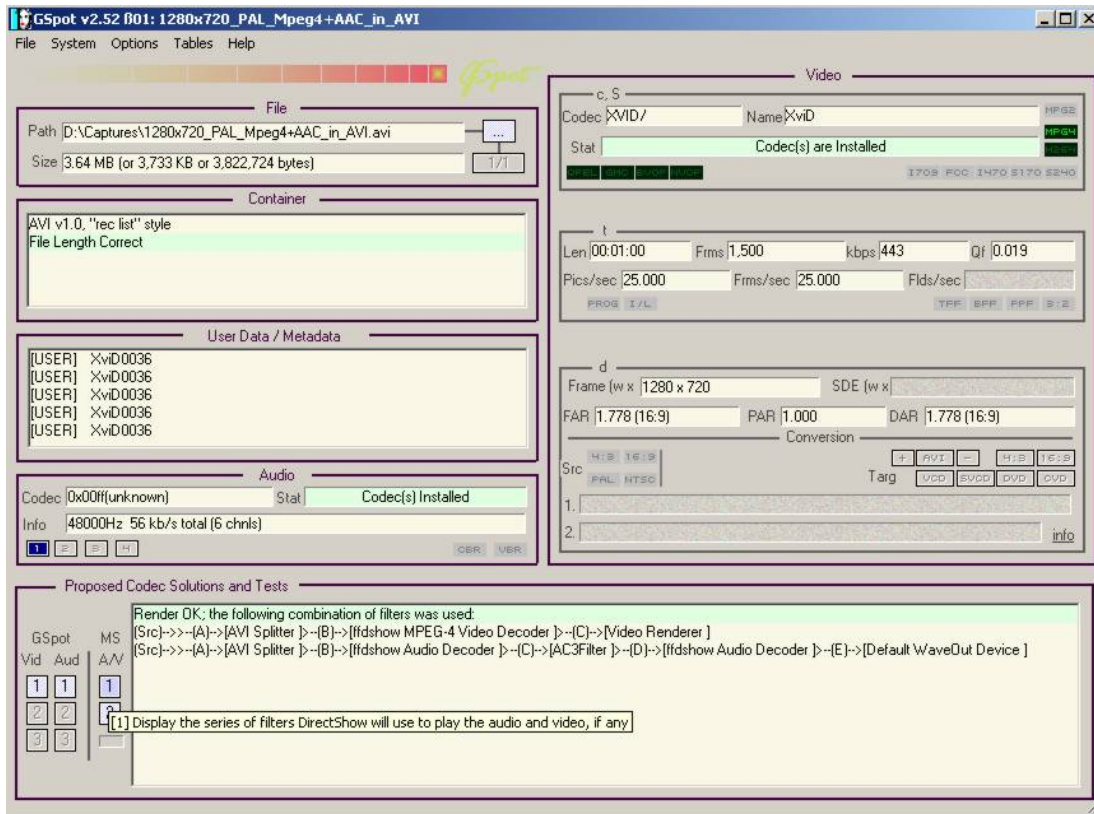
to load your AVI.

### 11.2.1.2 1.1.2) Other containers with audio:

It is not always possible to load your clips in AviSynth using AviSource (or one of the specific plugins which will be discussed below). Examples are non-AVIs which are clips contained inside a different container, like MKV, MP4, RMVB, OGM or ASF/WMV. In that case DirectShowSource is your last bet. It might also be possible that you have an AVI, with an appropriate VfW codec installed, but you want to use DirectShow codecs to open them in AviSynth. In that case you should also use DirectShowSource.

When playing a clip in WMP6.4 (mplayer2.exe), DirectShow filters (\*.ax) are used to play it. Those are the same ones which are "used" by DirectShowSource. So you need to be sure that you have the appropriate DirectShow filters installed. To find out which filters are used to play the clip, open the clip in WMP6.4 and check under: file -> properties -> Advanced. Here you can also change the settings of the filters. You will get more information about the filters when you open the clip in for example [GSpot](#). Just open the clip and press "1" under A/V in the "Proposed Codec Solutions and Tests" box. A "graph" is constructed which the filters which are used to play it:

## Avisynth 2.5 Selected External Plugin Reference



(In case you are wondering, due to my settings in AC3Filter it always shows up in the filter chain. But in this example it shouldn't be loaded because it doesn't support AAC.)

If you got the message "rendering failed (...)", it means that the appropriate DirectShow filters are not installed. Make also sure the file is playing correctly by pressing the "2" under the "1". Because if it is not playing, DirectShowSource can't load the clip. In general, you can have the following problem with this approach: *other DirectShow filters are used to play the media file than the ones you installed or you want to use. This can happen because the used filters have a higher merit (playing priority) than the ones you want to use.* There are two solutions for this problem:

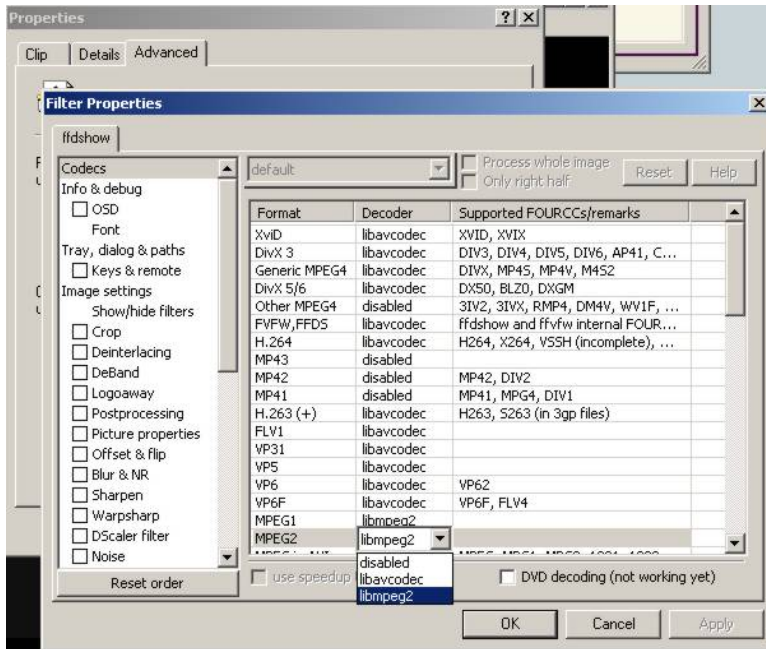
1. change the merit of the used filter using [Radlight Filter Manager](#).
2. use [GraphEdit \(last post of the thread\)](#) to construct a graph using the DirectShow filters of your choice and load that graph with DirectShowSource.

This will be discussed in the sections "[Changing the merit of DirectShow Filters](#)" and "[Using GraphEdit to make graphs of DirectShow filters and how do I load these graphs in AviSynth](#)".

Luckily you can install [ffdshow](#) (which comes with several DirectShow decoders), which is able to decode many formats. For example:

- MPEG1/2: enable mpeg1/2 by selecting the libavcodec or libmpeg2 library:

## Avisynth 2.5 Selected External Plugin Reference



- MJPEG in AVI: enable mjpeg by selecting the libavcodec library.
- DV in AVI: enable DV by selecting the libavcodec library.
- MPEG–4 ASP in OGM: **xxx**
- **MKV / MP4 / TS**
- h.264 in MKV/MP4: install **xxx** and use ffdshow
- h.264 in TS: install Haali splitter and use ffdshow (or CoreAVC)
- h.263 in FLV1: get the [[flv splitter](#)] and enable h.263 playback by selecting the libavcodec library.

Example:

Load MP4 (video: h.264, audio: aac) using DirectShowSource and ffdshow (aac decoding enabled in ffdshow; when the audio is AC3 or DTS you can also use AC3Filter instead). Your script becomes for example:

```
# adjust fps if necessary
DirectShowSource("d:\x264_aac.mp4", fps=25, convertfps=true)
```

### some other formats:

\* RM/RMVB (RealMedia / RealMedia Variable Bitrate; usually containing Real Video/Audio): install the [rmvb splitter](#) and the Real codecs by installing RealPlayer/[RealAlternative](#). Create the script:

```
# adjust fps if necessary
DirectShowSource("d:\clip.rmvb", fps=25, convertfps=true)
```

\* WMV/ASF (Windows Media Video / Advanced Systems Format; usually containing WMV/WMA): this format is not fully supported by ffdshow, so you will have to install wmv codecs. Get [WMF SDK v9 for W2K or later for XP/Vista](#) which contains the codecs (and the DMO wrappers necessary to use DMO filters in DirectShow). (Note that Microsoft's own VC1 codec is not supported in W2K since you need WMF SDK v11.) Create the script:

```
# adjust fps if necessary
DirectShowSource("d:\clip.wmv", fps=25, convertfps=true)
```

If the source material has variable framerate video, read this helpful [guide](#).

### 11.2.2 1.2) Loading video clips into AviSynth

As already explained, in general there are two ways to load your video into AviSynth:

1. using an AviSynth plugin which is designed to open some specific format.
2. using the DirectShowSource plugin.

A list of all these plugins and their accepted formats is given below.

#### 1) AviSynth filters and plugins which are designed to open specific formats:

##### AviSource – AVI/VDR:

[AviSource](#) supports all kind of AVIs with MP3 (VBR MP3) or AC3 audio. It also supports DV type 1 and type 2, and VirtualDub frameserver files (VDR).

An AVI can be loaded in AviSynth provided you have an appropriate VfW codec installed which can be used to decode the AVI. The default codec which is used to decode the AVI is specified in the beginning of the media file (in its header) itself as the FourCC (FOUR Character Code). From v2.55, an option fourCC is added, which lets you use other codecs to load your AVI in AviSynth.

A few examples:

```
AviSource("d:\filename.avi")
```

or without the audio:

```
AviSource("d:\filename.avi", false)
```

Forcing a decoder being used for loading the clip into AviSynth:

```
# load your avi using the XviD codec:  
# opens an avi (for example DivX3) using the XviD Codec  
AviSource("d:\filename.avi", fourCC="XVID")  
  
# load your dv-avi using the Canopus DV Codec:  
AviSource("d:\filename.avi", fourCC="CDVC")  
  
# vdr-files (VirtualDub frameserver files):  
AviSource("d:\filename.vdr")
```

If AviSynth is complaining about not being able to load your avi (**couldn't decompress ...**) you need to install an appropriate codec. GSpot, for example, will tell you what codec you need to install in order to be able to play your avi.

##### Mpeg2Source/DGDecode – MPEG1/MPEG2/VOB/TS/PVA:

DGDecode (old version Mpeg2Dec3) is an external plugin and supports MPEG-1, MPEG-2 / VOB, TS and PVA streams. Open them into DGIndex (or Dvd2avi 1.76/1.77.3 for Mpeg2Dec3) first and create a d2v script which can be opened in AviSynth (note that it will only open the video into AviSynth):

## Avisynth 2.5 Selected External Plugin Reference

A few examples:

```
# old Mpeg2dec3; if you need a d2v script
# which is created with dvd2avi 1.76/1.77.3
LoadPlugin("d:\mpeg2dec3.dll")
mpeg2source("d:\filename.d2v")

# DGDecode:
LoadPlugin("d:\dgedecode.dll")
mpeg2source("d:\filename.d2v")
```

Note that Mpeg2Dec3 is very limited compared to DGDecode, because it's actually an old version of DGDecode and it only supports MPEG-2 / VOB.

[DGAVCDec – raw AVC/H.264 elementary streams \[1\]](#)

DGAVCIndex: Index your raw AVC/H.264 stream.

Make an Avisynth script to frameserve the video:

```
LoadPlugin("d:\DGAVCDecode.dll")
AVCSource("d:\file.dga")
```

RawSource – raw formats with/without header:

The external plugin RawSource supports all kinds of raw video files with the YUV4MPEG2 header and without header (video files which contains YUV2, YV16, YV12, RGB or Y8 video data).

Examples:

```
# This assumes there is a valid YUV4MPEG2-header inside:
RawSource("d:\yuv4mpeg.yuv")

# A raw file with RGBA data:
RawSource("d:\src6_625.raw", 720, 576, "BGRA")

# You can enter the byte positions of the video frames
# directly (which can be found with yuvscan.exe). This
# is useful if it's not really raw video, but e.g.
# uncompressed MOV files or a file with some kind of header:
RawSource("d:\yuv.mov", 720, 576, "UYVY", \
    index="0:192512 1:1021952 25:21120512 50:42048512 75:62976512")
```

QTSource (with QuickTime 6 or 7) and QTReader – MOV/QT:

There are two ways to load your quicktime movies into AviSynth (and also RawSource for uncompressed movs): QTSource and QTReader. The former one is very recent and able to open many quicktime formats (with the possibility to open them as YUY2), but you need to install QuickTime player in order to be able to use this plugin. The latter one is very old, no installation of a player is required in order to be able to open quicktime formats in AviSynth.

*QTSource:*

You will need Quicktime 6 for getting video only or Quicktime 7 for getting audio and video.

## Avisynth 2.5 Selected External Plugin Reference

```
# YUY2 (default):
QTInput("FileName.mov", color=2)

# with audio (in many cases possible with QuickTime 7)
QTInput("FileName.mov", color=2, audio=true)

# raw (with for example a YUYV format):
QTInput("FileName.mov", color=2, mode=1, raw="yuyv")

# dither = 1; converts raw 10bit to 8bit video (v210 = 10bit uyvy):
QTInput("FileName.mov", color=2, dither=1, raw="v210")
```

### *QTReader:*

```
# If that doesn't work, or you don't have QuickTime,
# download the QTReader plugin (can be found in
# Doooms download section):
LoadVFAPIPlugin("C:\QTReader\QTReader.vfp", "QTReader")
QTReader("C:\quicktime.mov")
```

### Import filter – AviSynth scripts:

Just import the script using [Import](#) at the beginning of your script:

```
Import("d:\filename.avs")
```

In v2.05 or more recent version you can use the autopluging loading. Just move your AVS-file in the plugin folder containing the other (external) plugins, and rename the extension to 'avsi'. See also [FAQ](#) for more discussion.

### 2) DirectShowSource:

Have a look at section "[Other containers with audio](#)" for more information.

## 11.2.3 1.3) Loading audio clips into AviSynth

Most audio formats can be loaded in AviSynth, but there are some exceptions like MPL or multichannel WMA using W98/W2K. If it is not possible to load a clip in AviSynth, you will have to convert it to some other format which can be loaded. Remember to choose a format for which you will have a minimal downgrade in quality as a result of the conversion.

In general there are two ways to load your audio into AviSynth:

1. using an AviSynth plugin which is designed to open some specific format.
2. using the DirectShowSource plugin.

A list of all these plugins and their accepted formats is given below.

### 1) AviSynth filters and plugins which are designed to open specific formats:

#### WavSource – WAV:

[WavSource](#) supports all kind of WAVs, such as uncompressed WAV or MP3/MP3/AC3/DTS audio with a

## Avisynth 2.5 Selected External Plugin Reference

WAVE header. A WAV can be loaded in AviSynth provided you have an appropriate [ACM codec](#) installed which can be used to decode the WAV. The default codec which is used to decode the WAV is specified in the beginning of the media file (in its header; a bit similar as fourCC for video codecs).

audio: MP2/MP3 with a WAVE header:

There is an ACM codec for MP2 called [[QDesign](#)]. There are ACM codecs for MP3 [[Radium Codec](#)] or [[Lame ACM](#)].

audio: AC3/DTS in a WAVE header (also called DD-AC3 and DTSWAV):

There is an ACM codec for AC3 called [[AC3ACM](#)]. There is an ACM codec for AC3/DTS called valex ACM codec [[vac3acm](#)].

Example:

```
# DTS in WAV:
V = BlankClip(height=576, width=720, fps=25)
A = WAVSource("D:\audio_dts.wav")
AudioDub(V, A)
```

or if you have WinDVD platinum, install [hypercube's DTSWAV filter](#).

**A few examples:**

xxx

MPASource – MP1/MP2/MP3/MPA:

Example:

```
LoadPlugin("C:\Program Files\AviSynth25\plugins\mpasource.dll")
V = BlankClip(height=576, width=720, fps=25)
A = MPASource("D:\audio.mp3", normalize = false)
AudioDub(V, A)
```

NicAudio – MP1/MP2/MP3/MPA/AC3/DTS/LPCM:

Some examples:

```
LoadPlugin("C:\Program Files\AviSynth25\plugins\NicAudio.dll")

# AC3 audio:
V = BlankClip(height=576, width=720, fps=25)
A = NicAC3Source("D:\audio.AC3")
#A = NicAC3Source("D:\audio.AC3", downmix=2) # downmix to stereo
AudioDub(V, A)

# LPCM audio (48 kHz, 16 bit and stereo):
V = BlankClip(height=576, width=720, fps=25)
A = NicLPCMSource("D:\audio.lpcm", 48000, 16, 2)
AudioDub(V, A)
```

BassAudio –

## Avisynth 2.5 Selected External Plugin Reference

### MP3/MP2/MP1/OGG/WAV/AIFF/WMA/FLAC/WavPack/Speex/Musepack/AAC/M4A/APE/CDA:

BassAudio can be downloaded [here](#). Some examples:

```
# FLAC files:
bassAudioSource("C:\ab\Dido\001 Here With Me.flc")

# OGG files:
bassAudioSource("C:\ab\Dido\001 Here With Me.ogg")

# AAC files:
bassAudioSource("C:\ab\Dido\001 Here With Me.aac")

# Audio-CD Ripping using this plugin
# Download BASSCD 2.2 from official BASS homepage
# Extract basscd.dll and rename it to bass_cd.dll
# Place bass_cd.dll in the same folder as bassAudio.dll
bassAudioSource("D:\Track01.cda")
```

### 2) DirectShowSource:

Have a look at section "[Other containers with audio](#)" for more information. Some directshow filters (besides the ones available in ffdshow) for Ogg Vorbis, Speex, Theora and FLAC can be found [here](#).

#### audio: MP2/MP3 with a WAVE header:

Use ffdshow: MP1/2/3 decoding enabled and select mp3lib or libmad.

#### audio: AC3/DTS with a WAVE header (also called DD-AC3 and DTSWAV):

Use ffdshow: DTS decoding enabled and uncompressed: support all formats.

Example:

```
# DTS in WAV:
V = BlankClip(height=576, width=720, fps=25)
A = DirectShowSource("D:\audio_dts.wav")
AudioDub(V, A)
```

or making a graph:

```
# DTS in WAV:
# use WAVE parser and ffdshow or AC3filter: [add screenshots]
V = BlankClip(height=576, width=720, fps=25)
A = DirectShowSource("D:\audio_dts_wav.grf", video=false)
AudioDub(V, A)
```

#### aac:

Get an AAC directshow filter like CoreAAC (make sure downmixing is turned off, unless you want it to be downmixed;) or ffdshow (with AAC set to libfaad or realaac), and use

```
DirectShowSource("d:\audio.aac")
```

to load your AAC. You might need an AAC parser filter for DirectShow. Get one here [\[2\]](#).



## 11.2.4 1.4) Loading images into AviSynth

1) Use ImageReader or ImageSource to load your pictures into AviSynth (can load the most popular formats, except GIF and animated formats). See [internal documentation](#) for information.

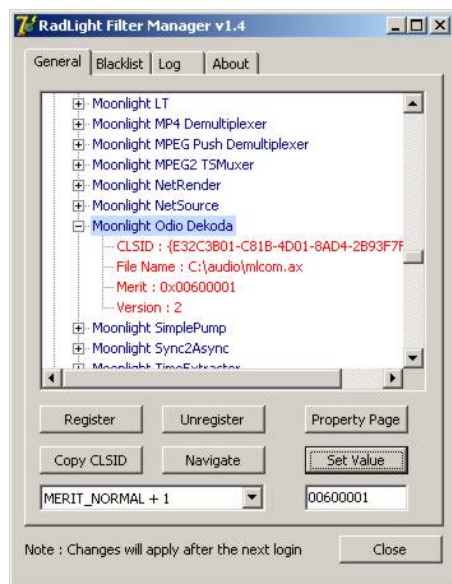
2) Use the Immaavs plugin for GIF, animated formats and other type of pictures.

```
# single picture:
immareadpic("x:\path\pic.bmp")

# animated gif:
immareadanim("x:\path\anim.gif")
```

## 11.3 2) Changing the merit of DirectShow Filters

Open Radlight Filter Manager and check which of the filters which can be used to play your clip has a higher merit. Change the merit of the filter you want to use for playback, for example:



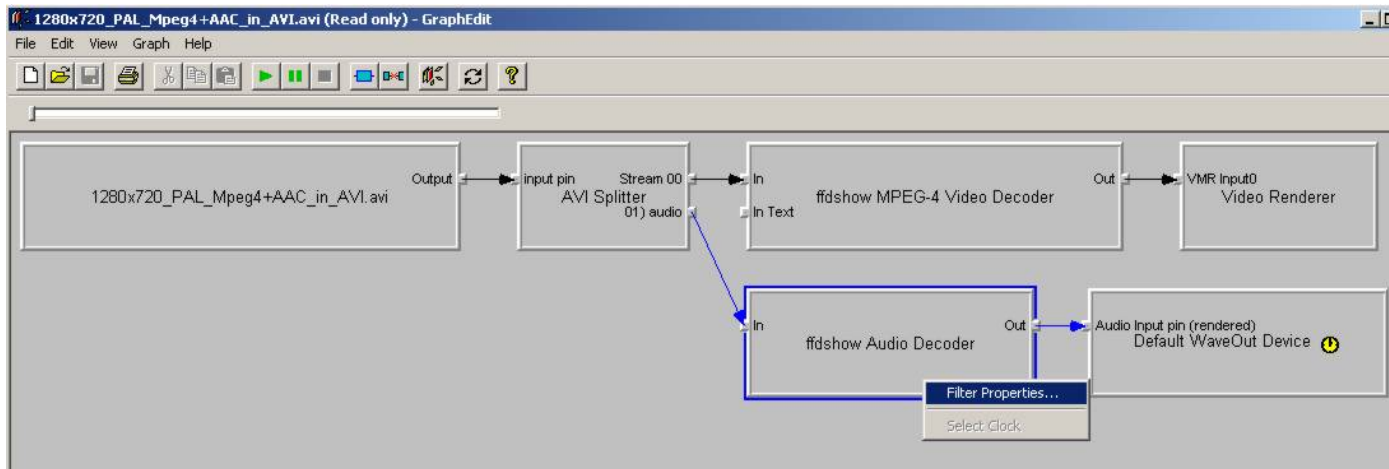
and restart your PC. In my experience this won't always work. In AC3Filter, for example, there is setting called 'Filter Merit' which is set to 'Prefer AC3Filter' by default. Although in my case the merit for AC3Filter was set lower than the merit for Moonlight Odio Dekoda (MERIT\_NORMAL versus MERIT\_NORMAL+1), the former was used to play my AC3 (I assume as a result of that 'Prefer AC3Filter' setting in AC3Filter; setting it to 'Prefer other decoder' solves this problem). Other filters might have such settings too.

## 11.4 3) Using GraphEdit to make graphs of DirectShow filters and loading these graphs in AviSynth

As an example it will be shown how to make a graph where CoreAAC Audio Decoder will be used to render the audio in an AVI-AAC file. More accurately it will be shown how the ffdshow Audio decoder should be replaced by the CoreAAC Audio Decoder, where the former has a higher merit (which implies that filter will be used when playing the clip in a DirectShow based player like WMP6.4 or when opening the AVI directly in AviSynth by using DirectShowSource):

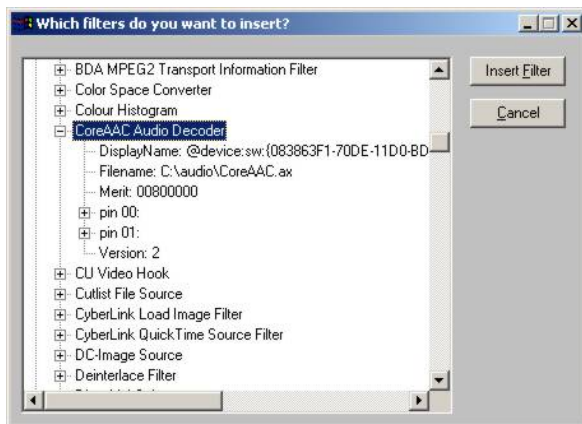
## Avisynth 2.5 Selected External Plugin Reference

Open GraphEdit and open your clip: File tab -> Open Graph -> Select All Files -> open your clip:

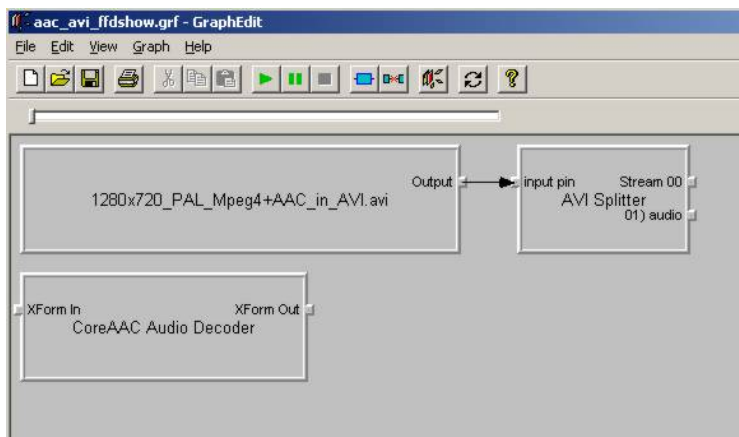


Right-click on the filter (ffdshow Audio decoder) -> Filter Properties -> change settings if necessary.

Graph tab -> Insert Filters -> under DirectShow Filters -> select CoreAAC Audio Decoder:

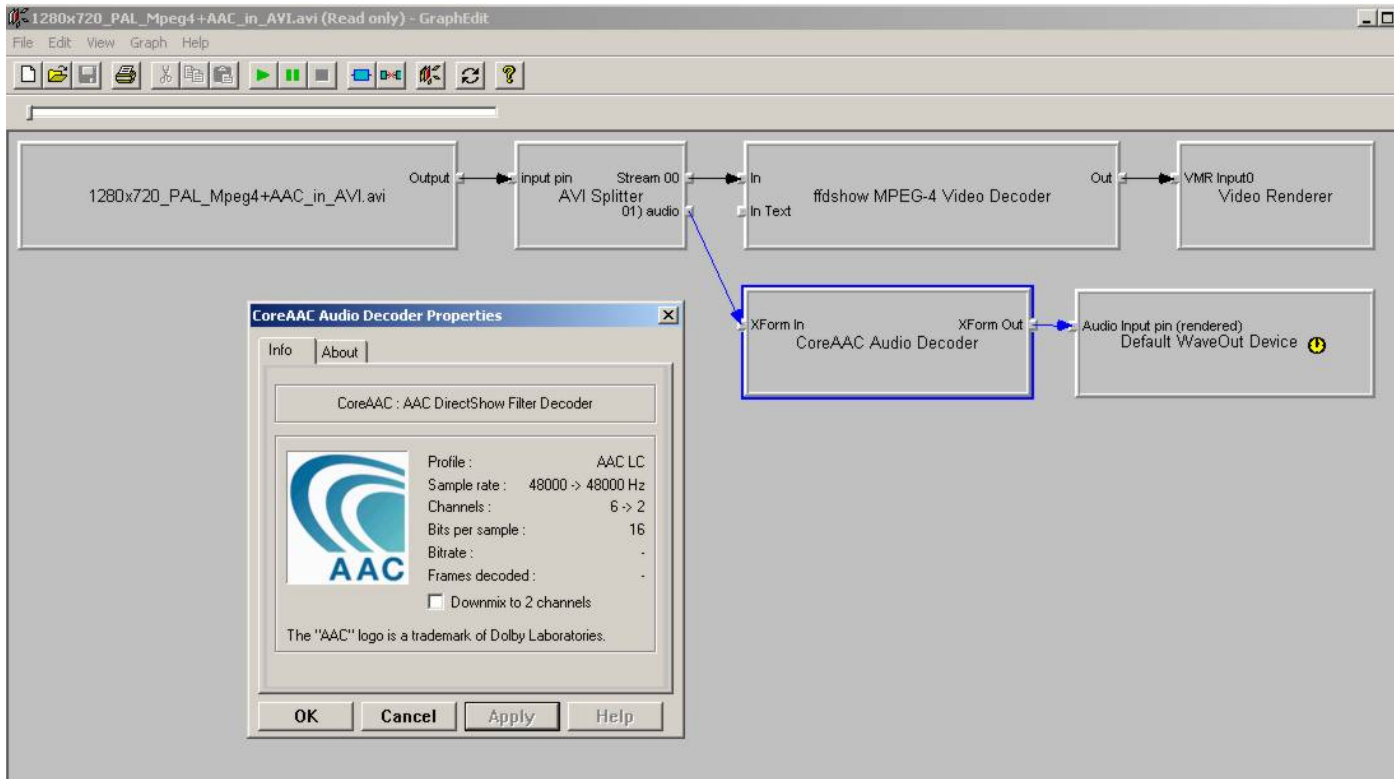


Press Insert Filter. Remove the ffdshow Audio decoder by selecting it and pressing Delete:



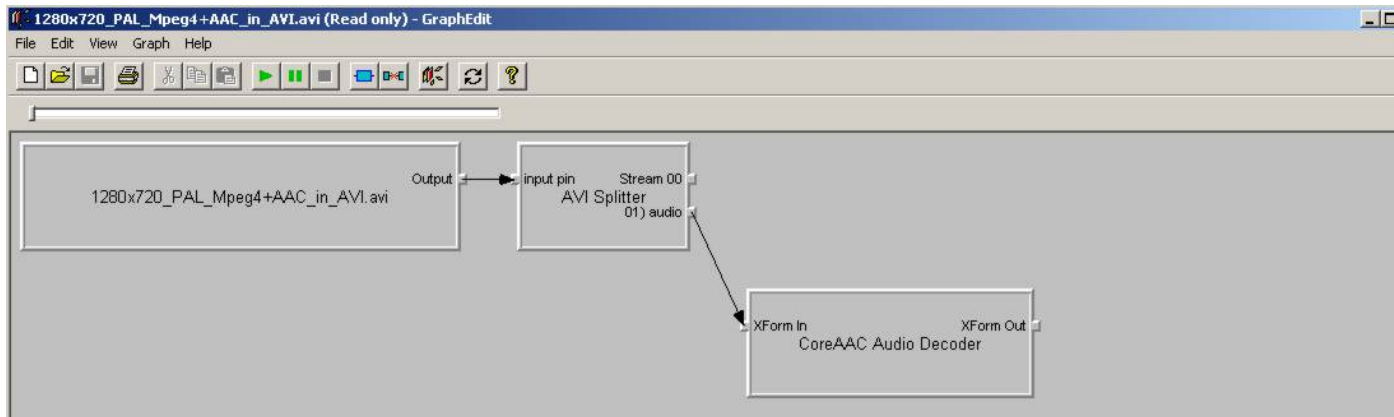
Finally, connect the CoreAAC Audio Decoder, by connecting the pins with your left mouse button (arrows will be drawn automatically):

## Avisynth 2.5 Selected External Plugin Reference



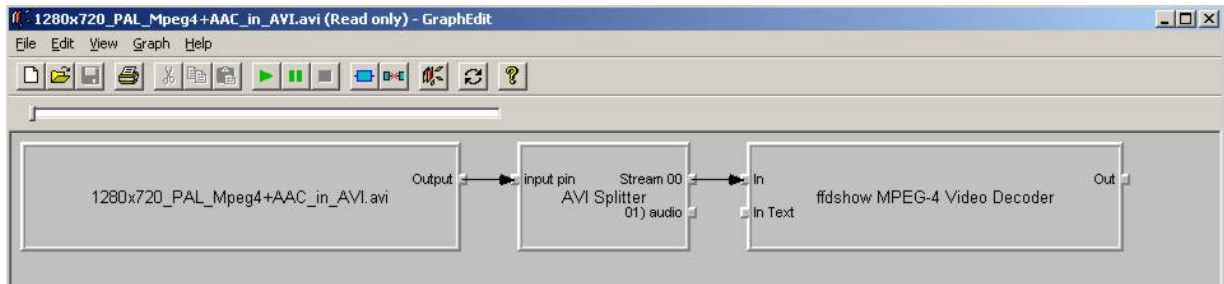
Check and adjust the Filter Properties if necessary. Press play to check that the clip is playable by the selected combination of DirectShow filters. This is very important, because if it's not playable, AviSynth will not be able to open the clip. In that case you should select and or install other filters which can play the clip.

Finally remove the WaveOut Device and the video filters, because AviSynth needs a free pin to connect itself when DirectShowSource is called in a script.



Save the graph as `audio.grf`. If you want to load the video too in AviSynth, it should be loaded separately, using a different graph (where the audio part and the Video Renderer is removed):

## Avisynth 2.5 Selected External Plugin Reference



Save the graph as video.grf. Your script becomes:

```
# change fps if necessary
vid = DirectShowSource("D:\video.grf", \
    fps=25, convertfps=true, audio=false)
aud = DirectShowSource("D:\audio.grf", video=false)
AudioDub(vid, aud)
```

## 11.5 To Do

- The following should be added: tpr, aup, RaWav (>4 GB WAVE files), ffmpegsource,
- ../docs/english/faq\_loading\_clips.htm#How\_do\_I\_load\_MP4.2FMKV.2FM2TS\_into\_AviSynth.3F, swf (ffmpegsource) should be added.

\$Date: 2008/07/11 18:28:55 \$

## 11.6 Interlaced and Field-based video

Currently (v2.5x and older versions), Avisynth has no interlaced flag which can be used for interlaced video. There is a field-based flag, but contrary to what you might expect, this flag is not related to interlaced video. In fact, all video (progressive or interlaced) is frame-based, unless you use Avisynth filters to change that. There are two filter who turn frame-based video into field-based video: [SeparateFields](#) and [AssumeFieldBased](#).

- SeparateFields / Weave
  - ◆ SeparateFields: If needed (for example when denoisers require progressive input), you can split up interlaced streams in their fields by using SeparateFields. The output clip will have twice the framerate of the original clip.
  - ◆ Weave: After applying a spatial denoiser (for temporal denoisers the situation is a bit more [involved](#)), the filter Weave can be used to combine the fields again to produce interlaced frames.
- AssumeFieldBased / AssumeFrameBased
  - ◆ AssumeFieldBased: Applying this filter results in a field-based clip. This can be useful when making "artificial" field-based clips.
  - ◆ AssumeFrameBased: Applying this filter results in a frame-based clip. This can be useful when making "artificial" frame-based clips.

### 11.6.1 Color conversions and interlaced / field-based video

Let's assume you have **interlaced video**, you want to work in field-based mode (to apply some filtering for example) and you also need to do some color conversion. Do you need to do the conversion on the frame-based clip or can you do it on the field-based clip? Well, that depends on the [color conversion](#) you

want to apply:

- \* YUY2<->RGB conversions can be done on either of them. (Note, that in this case, the setting `interlaced=true/false` doesn't do anything. It's simply ignored.)
- \* YV12<->YUY2/RGB conversions should be done on the frame-based clip (with the **`interlaced=true`** setting). Doing them on the field-based clip will yield incorrect results. The exact reason of this is outside the scope of this page, but it is a consequence of how the color format YV12 is defined. The main issue is that chroma is shared between pixels on two different lines in a frame. More information can be found here [Sampling](#).

The more experienced users should consider the following. In general, interlaced video has parts where there is no or little movement. Thus, you won't hardly see any interlacing effects (also called combing) in these parts. They can be considered progressive, and when doing a YV12<->YUY2/RGB conversion on a progressive video you should use the **`interlaced=false`** setting to get better results. It is possible to the YV12<->YUY2/RGB conversion on frame basis while switching between `interlaced=true` and `interlaced=false`. Here's how to do it (you will need to have `decomb` installed in order to be able to use the function `IsCombed`)

```
function ConvertHybridToYUY2(clip a, int "threshold", bool "debug")
{
  debug = default(debug, false)
  global threshold = default(threshold, 20)

  b = ConvertToYUY2(a, interlaced=false)
  c = ConvertToYUY2(a, interlaced=true)
  ConditionalFilter(a, b, c, "IsCombed(threshold)", "equals", "true", show=debug)
}

function ConvertHybridToRGB(clip a, int "threshold", bool "debug")
{
  debug = default(debug, false)
  global threshold = default(threshold, 20)

  b = ConvertToYUY2(a, interlaced=false)
  c = ConvertToYUY2(a, interlaced=true)
  ConditionalFilter(a, b, c, "IsCombed(threshold)", "equals", "true", show=debug)
}

AviSource("D:\captures\interlaced-clip.avi") # interlaced YV12
#ConvertHybridToYUY2(debug=true)
ConvertHybridToYUY2()
```

However, the downside of this is that it may lead to [chroma shimmering](#) in the combed–progressive frame transitions. So, it's not a perfect solution.

### 11.6.2 Color conversions, interlaced / field-based video and the interlaced flag of dvd2avi

For the more experienced users. `Dvd2avi` keeps track of whether a frame is interlaced or progressive (by using the `interlaced` flag). In principle, `dvd2avi` can be modified to store this in a text–file and `AviSynth` can read and use it on frame–basis. However, it's useless. The problem is that sometimes progressive video is encoded as interlaced, and thus is detected as interlaced by `dvd2avi`. In the previous section, it is explained, that in that case you should use `interlaced=false` during the YV12<->YUY2/RGB conversion (since there's no movement) to get more accurate results. **So, it's the presence of combing which is important for the**

**YV12<->YUY2/RGB conversion, and not whether a frame is interlaced.**

### 11.6.3 Changing the order of the fields of a clip

There is a filter which swaps the even/odd fields [SwapFields](#), and a plugin which reverses the field dominance [ReverseFieldDominance](#). The former changes the spatial order and the latter the temporal order.

#### 11.6.3.1 Swapping fields:

before using SwapFields:

line	frame 0
0	t0
1	b1
2	t2
3	b3
4	t4
5	b5

field order (top field first then bottom field):

line	field 0	field 1
0	t0	
1		b1
2	t2	
3		b3
4	t4	
5		b5

after using SwapFields:

line	frame 0
0	b1
1	t0
2	b3
3	t2
4	b5
5	t4

field order (top field first then bottom field):

line	field 0	field 1
0		b1
1	t0	
2		b3
3	t2	
4		b5

## Avisynth 2.5 Selected External Plugin Reference

5	t4	
---	----	--

Note that the even and odd lines are swapped, so you can call the Top Field as Bottom Field, and vice versa.

### 11.6.3.2 Reversing field dominance:

before reversing the field dominance:

line	frame 0
0	t0
1	b1
2	t2
3	b3
4	t4
5	b5

field order (top field first then bottom field):

line	field 0	field 1
0	t0	
1		b1
2	t2	
3		b3
4	t4	
5		b5

after reversing the field dominance (assuming the lines will be shifted up, and the last one will be duplicated):

line	frame 0
0	b1
1	t2
2	b3
3	t4
4	b5
5	b5

field order (bottom field first then top field):

line	field 0	field 1
0	b1	
1		t2
2	b3	
3		t4
4	b5	
5		b5

Note that the top and bottom fields are swapped, but the even and odd lines are not swapped.

### 11.6.3.2 Reversing field dominance:

### 11.6.4 The parity (= order) of the fields in AviSynth

If a clip is field-based AviSynth keeps track of the parity of each field (that is, whether it's the top or the bottom field of a frame). If the clip is frame-based it keeps track of the dominant field in each frame (that is, which field in the frame comes first when they're separated).

However, this information isn't necessarily correct, because field information usually isn't stored in video files and AviSynth's source filters just normally default to assuming bottom field first (with the exception of the MPEG2Source plugin which gets it right!).

### 11.6.5 About DV / DVD in relation to field dominance

The field dominance is not the same for every source. DV (with interlaced content) has bottom field first, while DVD (or CVD/SVCD) has top field first. Thus when convert between those two, you need to change the field dominance. This can be done in AviSynth (see above), but also in the encoder itself (for bff material like DV footage, you need to set the Upper field first flag). Some comments on other [\[methods\]](#).

### 11.6.6 References

[\[DV / DVD and field dominance\]](#)

About [\[field dominance\]](#).

[\[Doom thread\]](#) about swapped fields and field dominance.

[\[ReverseFieldDominance plugin\]](#)

\$Date: 2006/12/15 19:29:25 \$



# 12 Resampling

<trivial def of resampling>

## 12.1 Resampling

### 12.1.1 The Nyquist–Shannon sampling theorem

The idea behind resampling is to reconstruct to continuous signal from the original sampled signal and resample it again using more samples (which is called interpolation or upsampling) or fewer samples (which is called decimation or downsampling). In the context of image processing the original samples signal is the source image (where the value of the pixels are the samples). Interpolation is called upsampling and decimation is called downsampling.

The Nyquist–Shannon sampling theorem asserts that the uniformly spaced discrete samples are a complete representation of the signal if this bandwidth is less than half the sampling rate. [source: [wikipedia](#)]. In theory the continuous signal can be reconstructed as follows (n runs over the integers)

$$s(x) = \sum_n s(n^*T) * \text{sinc}((x-n^*T)/T), \text{ with } \text{sinc}(x) = \frac{\sin(\pi*x)}{(\pi*x)} \text{ for } x \neq 0, \text{ and } = 1 \text{ for } x=0$$

with  $f_s = 1/T$  the sampling rate,  $s(n^*T)$  the samples of  $s(x)$  and  $\text{sinc}(x)$  the resampling kernel. Although in practice, signals are never perfectly bandlimited, nor can we construct a perfect reconstruction filter (because an image always has a limited number of pixels), we can get as close as we want in a prescribed manner. This is the reason why many resizers are based on this construction. Note that the reconstruction filter has the following properties:

1. The reconstruction is correct for  $x=m^*T$  (m integer), meaning that  $s(m^*T)$  are indeed samples of the continuous signal  $s(x)$ . (This follows from the fact that  $\text{sinc}(0)=1$ .)
2. The sum of the coefficients is one, thus  $\sum_n \text{sinc}((x-n^*T)/T) = 1$  (this seems to be true for all x and T, but I can't find a reference of it). This implies that if all samples have the same value (say  $s(n^*T)=y$  for all n), then it follows that the continuous signal itself is constant and equal to y. That is,  $s(x)=y$  for all x.
3. The resampling kernel,  $\text{sinc}(x)$ , is maximal and equal to one for  $x=0$ . It has such a shape that the sample  $s(n^*T)$  for which its location ( $n^*T$ ) is the closest to the value x will contribute the most to  $s(x)$ .
4. The resampling kernel,  $\text{sinc}(x)$ , is symmetric. That is,  $\text{sinc}(x) = \text{sinc}(-x)$ . This means that the samples  $s(n^*T)$  and  $s(-n^*T)$  will contribute equally to  $s(0)$ .
5. The resampling kernel,  $\text{sinc}(x)$ , is analytic (infinitely differentiable) everywhere [source: [wikipedia](#)].



The normalized sinc kernel

As an approximation of this reconstruction function, many interpolation functions  $s(x)$  are written in the form

$$s(x) = \sum_n s(nT) * f((nT-x)/T) := \sum_n s_T(n) * f(n - x/T)$$

with  $f(x)$  the (symmetric) resampling filter (or resampling kernel) obeying some of the properties listed above and  $\{s_T(n)\}_n$  the samples. Unfortunately such an approximation causes artefacts such as blurring, aliasing and ringing. The possible artefacts are explained [here](#).

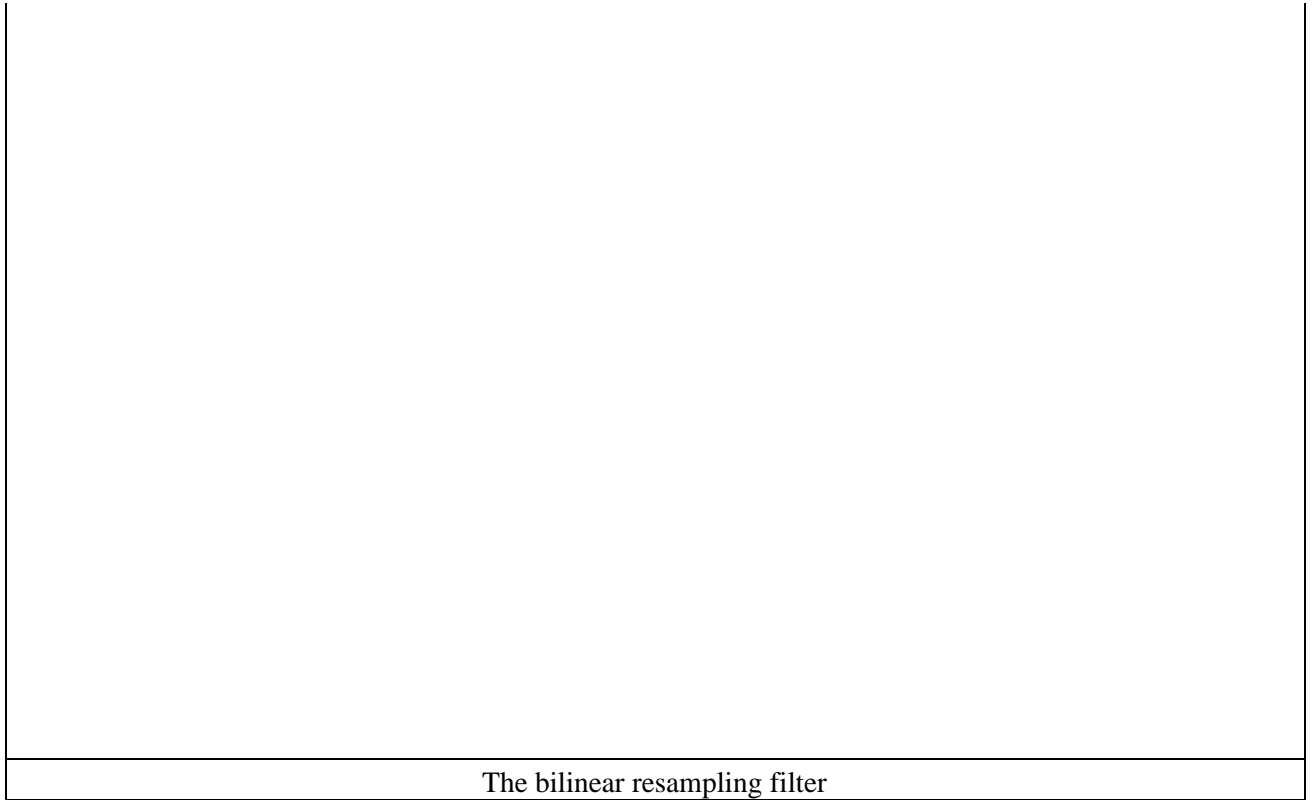
### 12.1.2 Example: the bilinear resampling filter – upsampling

Let's consider a bilinear resampling filter

$$f(x) = 1 - |x| \text{ for } |x| < 1 \text{ and } = 0 \text{ elsewhere}$$

Note that this filter satisfies the properties 2, 3 and 4 that are mentioned above.

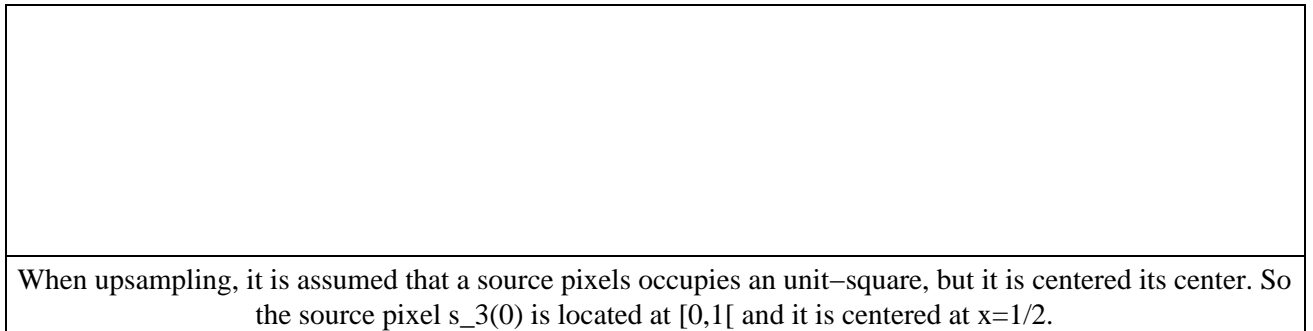




The bilinear resampling filter

Let's upsample an image (consisting of three pixels) by a factor  $T=3$ , and suppose that the target image has nine pixels.

Let  $s_3(j)$  be the luma-channel (or one of the color-channels) of the  $j+1$ -th source pixel, and  $t(j)$  the luma-channel (or one of the color-channels) of the  $j+1$ -th target pixel. Two fake pixels are added:  $s_3(-1)$  and  $s_3(3)$ . They will be set equal to the value of the edge pixels:  $s_3(0)$  and  $s_3(2)$ . The center pixels of the source (here  $s_3(1)$ ) and the target (here  $t(4)$ ) should coincide.



When upsampling, it is assumed that a source pixels occupies an unit-square, but it is centered its center. So the source pixel  $s_3(0)$  is located at  $[0,1[$  and it is centered at  $x=1/2$ .

More generally: source pixels:  $s_T(0) \dots s_T(m-1)$  and target pixels:  $t(0) \dots t(n-1)$ . The centers should coincide  $\Rightarrow t(c_n) = s_T(c_m) = s(c_m * T)$  with  $c_m = (m-1)/2$  and  $c_n = (n-1)/2$ .

This implies that

$$t(x) = s(x - (c_n - c_m * T)) = s_T((x - c_n)/T - c_m)$$

with

## Avisynth 2.5 Selected External Plugin Reference

$$s(x) = \sum_n s_T(n) * f(n - x/T)$$

In our example:  $T=3$ ,  $m=3$ ,  $n=9$ , thus  $c_m = 1$ ,  $c_n = 4$  and thus

$$t(x) = s(x - 1)$$

with

$$s(x) = \sum_n s_3(n) * f(n - x/3)$$

For the third target pixel, for example, we will get

$$\begin{aligned} t(2) &= s(1) = \sum_n s_3(n) * f(n - 1/3) = s_3(-1) * f(-1 - 1/3) + s_3(0) * f(0 - 1/3) + s_3(1) * f(1 - 1/3) + \\ & s_3(2) * f(2 - 1/3) + s_3(3) * f(3 - 1/3) \\ &= s_3(-1) * f(-4/3) + s_3(0) * f(-1/3) + s_3(1) * f(2/3) + s_3(2) * f(5/3) + s_3(3) * f(8/3) \\ &= s_3(0) * f(-1/3) + s_3(1) * f(2/3) \\ &= s_3(0) * 2/3 + s_3(1) * 1/3 \end{aligned}$$

$s_3(0)$ contributes $2/3$ to the target pixel $t(2)$ , and $s_3(1)$ contributes $1/3$ to the target pixel $t(2)$

The remaining target pixels are given by:

$$\begin{aligned} t(0) &= 1/3 * s_3(-1) + 2/3 * s_3(0) = f(-2/3) * s_3(-1) + f(1/3) * s_3(0) = s_3(0) \\ t(1) &= 3/3 * s_3(0) + 0/3 * s_3(1) = f(0) * s_3(0) + f(1) * s_3(1) = s_3(0) \\ t(2) &= 2/3 * s_3(0) + 1/3 * s_3(1) = f(-1/3) * s_3(0) + f(2/3) * s_3(1) \\ t(3) &= 1/3 * s_3(0) + 2/3 * s_3(1) = f(-2/3) * s_3(0) + f(1/3) * s_3(1) \\ t(4) &= 1 * s_3(1) + 0 * s_3(2) = f(0) * s_3(1) + f(1) * s_3(2) \\ t(5) &= f(-1/3) * s_3(1) + f(2/3) * s_3(2) \\ t(6) &= f(-2/3) * s_3(1) + f(1/3) * s_3(2) \\ t(7) &= 3/3 * s_3(2) + 0/3 * s_3(3) = f(0) * s_3(2) + f(1) * s_3(3) = s_3(2) \\ t(8) &= 2/3 * s_3(2) + 1/3 * s_3(3) = f(-1/3) * s_3(2) + f(2/3) * s_3(3) = s_3(2) \end{aligned}$$

### 12.1.3 Example: the bilinear resampling filter – downsampling

Let's upsample an image (consisting of nine pixels) by a factor  $T=1/3$  (thus downsample by a factor of 3), with the target image having three pixels.

We have

## Avisynth 2.5 Selected External Plugin Reference

$$t(x) = s(x - (c_n - c_m * T))$$

with

$$s(x) = \sum_n s_T(n) * f_T(n - x/T)$$

In our example:  $T=1/3$ ,  $m=9$ ,  $n=3$ , thus  $c_m = 4$ ,  $c_n = 1$  and thus

$$t(x) = s(x + 1/3)$$

with

$$s(x) = \sum_n s_{1/3}(n) * f_T(n - 3*x)$$

For the first three target pixels, for example, we will get (note that  $f_T$  is zero outside  $[-1/T, 1/T]$  for the bilinear resampling filter;  $f_T(x) = f(T*x)$ )

$$\begin{aligned} t(0) &= s(1/3) = \sum_n s_{1/3}(n) * f_T(n - 1) = s_{1/3}(-1) * f_T(-1 - 1) + s_{1/3}(0) * f_T(0 - 1) + s_{1/3}(1) * \\ & f_T(1 - 1) + s_{1/3}(2) * f_T(2 - 1) + s_{1/3}(3) * f_T(3 - 1) \\ &= s_{1/3}(-1) * f_T(-2) + s_{1/3}(0) * f_T(-1) + s_{1/3}(1) * f_T(0) + s_{1/3}(2) * f_T(1) + s_{1/3}(3) * f_T(2) \\ &= s_{1/3}(-1) * f(-2/3) + s_{1/3}(0) * f(-1/3) + s_{1/3}(1) * f(0) + s_{1/3}(2) * f(1/3) + s_{1/3}(3) * f(2/3) \\ &= s_{1/3}(-1) * 1/3 + s_{1/3}(0) * 2/3 + s_{1/3}(1) * 1 + s_{1/3}(2) * 2/3 + s_{1/3}(3) * 1/3 \end{aligned}$$

$$\begin{aligned} t(1) &= s(4/3) = \sum_n s_{1/3}(n) * f_T(n - 4) = s_{1/3}(2) * f_T(2 - 4) + s_{1/3}(3) * f_T(3 - 4) + s_{1/3}(4) * \\ & f_T(4 - 4) + s_{1/3}(5) * f_T(5 - 4) + s_{1/3}(6) * f_T(6 - 4) \\ &= s_{1/3}(2) * f_T(-2) + s_{1/3}(3) * f_T(-1) + s_{1/3}(4) * f_T(0) + s_{1/3}(5) * f_T(1) + s_{1/3}(6) * f_T(2) \\ &= s_{1/3}(2) * f(-2/3) + s_{1/3}(3) * f(-1/3) + s_{1/3}(4) * f(0) + s_{1/3}(5) * f(1/3) + s_{1/3}(6) * f(2/3) \\ &= s_{1/3}(2) * 1/3 + s_{1/3}(3) * 2/3 + s_{1/3}(4) * 1 + s_{1/3}(5) * 2/3 + s_{1/3}(6) * 1/3 \end{aligned}$$

$$\begin{aligned} t(2) &= s(7/3) = \sum_n s_{1/3}(n) * f_T(n - 7) = s_{1/3}(5) * f_T(5 - 7) + s_{1/3}(6) * f_T(6 - 7) + s_{1/3}(7) * \\ & f_T(7 - 7) + s_{1/3}(8) * f_T(8 - 7) + s_{1/3}(9) * f_T(9 - 7) \\ &= s_{1/3}(5) * f_T(-2) + s_{1/3}(6) * f_T(-1) + s_{1/3}(7) * f_T(0) + s_{1/3}(8) * f_T(1) + s_{1/3}(9) * f_T(2) \\ &= s_{1/3}(5) * f(-2/3) + s_{1/3}(6) * f(-1/3) + s_{1/3}(7) * f(0) + s_{1/3}(8) * f(1/3) + s_{1/3}(9) * f(2/3) \\ &= s_{1/3}(5) * 1/3 + s_{1/3}(6) * 2/3 + s_{1/3}(7) * 1 + s_{1/3}(8) * 2/3 + s_{1/3}(9) * 1/3 \end{aligned}$$

$t(1) = s_{1/3}(2) * 1/3 + s_{1/3}(3) * 2/3 + s_{1/3}(4) * 1 + s_{1/3}(5) * 2/3 + s_{1/3}(6) * 1/3$
---

### 12.1.4 Example: the bicubic resampling filter – upsampling

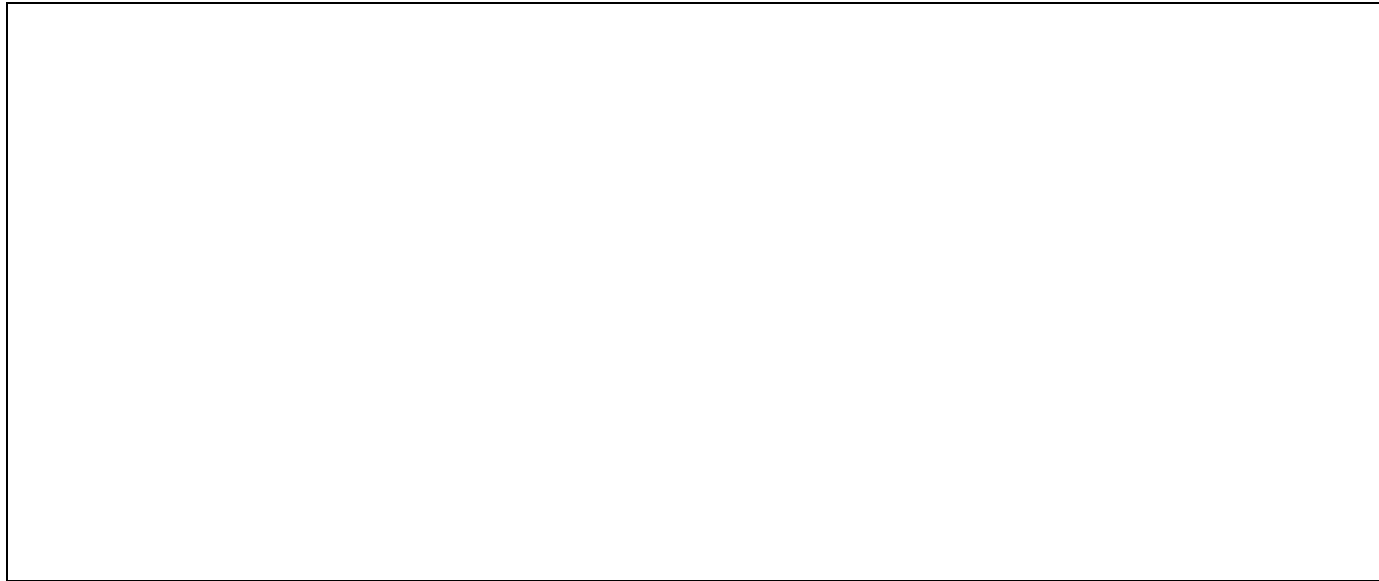
Suppose we want to obtain the value of  $t(2)$  in our first example.

We will define our coordinate system  $(x,y)$  in such a way that  $s(0)$  is centered at  $x=1/2$  (and thus located at  $[0,1]$ ). It follows that the source pixels  $s(j)$  are centered at  $x=(2*j+1)/2$ , and the target pixels  $t(j)$  are centered at  $x=(2*j+1)/6$ . The distances from  $t(2)$  to the source pixels  $s(j)$  are (with  $d(x;y) = x-y$ ):

$$\begin{aligned}d(s(-1);t(2)) &= -4/3 \\d(s(0);t(2)) &= -1/3 \\d(s(1);t(2)) &= 2/3 \\d(s(2);t(2)) &= 5/3 \\d(s(3);t(2)) &= 8/3\end{aligned}$$

Since the bicubic filter has support=2, this means that the pixels  $s(3)$  and higher don't contribute to the value of  $t(2)$ . We will get

$$\begin{aligned}t(2) &= f(-4/3) * s(-1) + f(-1/3) * s(0) + f(2/3) * s(1) + f(5/3) * s(2) \\t(2) &= -0.0329 * s(-1) + 0.7099 * s(0) + 0.3457 * s(1) + -0.0226 * s(2) \quad // \quad b=1/3, c=1/3\end{aligned}$$



For the other pixels we will get:

$$\begin{aligned}t(0) &= f(-5/3) * s(-2) + f(-2/3) * s(-1) + f(1/3) * s(0) + f(4/3) * s(1) \\t(1) &= f(-1) * s(-1) + f(0) * s(0) + f(1) * s(1) + f(2) * s(2) \\t(2) &= f(-4/3) * s(-1) + f(-1/3) * s(0) + f(2/3) * s(1) + f(5/3) * s(2) \\t(3) &= f(-5/3) * s(-1) + f(-2/3) * s(0) + f(1/3) * s(1) + f(4/3) * s(2) \\t(4) &= f(-1) * s(0) + f(0) * s(1) + f(1) * s(2) + f(2) * s(3) \\t(5) &= f(-4/3) * s(0) + f(-1/3) * s(1) + f(2/3) * s(2) + f(5/3) * s(3) \\t(6) &= f(-5/3) * s(0) + f(-2/3) * s(1) + f(1/3) * s(2) + f(4/3) * s(3) \\t(7) &= f(-1) * s(1) + f(0) * s(2) + f(1) * s(3) + f(2) * s(4) \\t(8) &= f(-4/3) * s(1) + f(-1/3) * s(2) + f(2/3) * s(3) + f(5/3) * s(4)\end{aligned}$$

## 12.2 An implementation of the resampling filters

To get implement resampling we need to know the first source pixel (also called the offset) and also the last source pixel that contributes to a certain target pixel.

Let  $T = n/m$  ( $m$  = number of source pixels,  $n$  = number of target pixels) and  $\text{filter\_step} = \min(T; 1)$ . Thus  $\text{filter\_step} = 1$  when upsampling and  $\text{filter\_step} = T (<1)$  when downsampling.

In source space, a source pixel occupies a square of length  $\text{filter\_step}$ , and it is centered its center of this square. The source pixels are located at  $s(v) = (2*v+1)/2 * \text{filter\_step}$ . The target pixels are located at  $t(v) = (2*v+1)/(2*T) * \text{filter\_step}$ .

The location,  $r(j)$ , of the source pixel  $s_{-T}(j)$  that is located directly to the left of the target pixel is given by

$$t(j) = s_{-T}((j - c_n)/T + c_m) \Rightarrow r(j) = \text{floor}((j - c_n)/T + c_m) \quad (\text{thus rounding down to the nearest integer})$$

and rewriting this expression gives

$$\begin{aligned} r(j) &= \text{floor}((j - c_n)/T + c_m) \\ &= \text{floor}((j - (n-1)/2)/T + (m-1)/2) \\ &= \text{floor}(j/T - n/(2*T) + 1/(2*T) + (m-1)/2) \\ &= \text{floor}(j/T - m/2 + 1/(2*T) + (m-1)/2) \\ &= \text{floor}(j/T + 1/(2*T) - 1/2) \\ &= \text{floor}(j/T + 1/(2*T) + 1/2) - 1 \end{aligned}$$

The offset depends on the support of the resampling filter, since the support will determine how many source pixels will contribute to the target pixel  $t(j)$ . In source\_space, all source pixels located in the interval  $[-\text{support}/\text{filter\_step}, \text{support}/\text{filter\_step}]$  contribute to the target pixel. This means that  $(\text{support} / \text{filter\_step} - 1)$  pixels located to the left of the position  $r(j)$  contribute to the target pixel  $t(j)$ . Thus the offset is given by

$$\begin{aligned} \text{off}(j) &= \text{floor}(j/T + 1/(2*T) + 1/2) - 1 - (\text{support} / \text{filter\_step} - 1) \\ &= \text{floor}(1/(2*T) - 1/2 + j/T) - \text{support} / \text{filter\_step} + 1 \\ &= r(j) - \text{support} / \text{filter\_step} + 1 \end{aligned}$$

So, in general, the target pixel  $t(j)$  is calculated as follows

$$\begin{aligned} t(j) &= f(b_{-T}(j)) * s(\text{off}(j)) + f(b_{-T}(j)+1) * s(\text{off}(j)+1) + f(b_{-T}(j)+2) * s(\text{off}(j)+2) + f(b_{-T}(j)+3) * s(\text{off}(j)+3) + \\ &\dots // \text{upsampling} \\ t(j) &= f(b_{-T}(j)) * s(\text{off}(j)) + f(b_{-T}(j)+T) * s(\text{off}(j)+1) + f(b_{-T}(j)+2*T) * s(\text{off}(j)+2) + f(b_{-T}(j)+3*T) * \\ &s(\text{off}(j)+3) + \dots // \text{downsampling} \\ t(j) &= f(b_{-T}(j)) * s(\text{off}(j)) + f(b_{-T}(j)+\text{filter\_step}) * s(\text{off}(j)+1) + f(b_{-T}(j)+2*\text{filter\_step}) * s(\text{off}(j)+2) + \dots // \\ &\text{general case} \end{aligned}$$

The value  $b_{-T}(j)$  can be determined as follows. Note that

$$\begin{aligned} d(0;s(v)) &= |2*v+1|/2 // \text{upsampling} \\ d(0;s(v)) &= |2*v+1|/2 * T // \text{downsampling} \\ d(0;s(v)) &= |2*v+1|/2 * \text{filter\_step} // \text{general case} \\ d(0;t(0)) &= 1/(2*T) // \text{upsampling} \\ d(0;t(0)) &= 1/2 // \text{downsampling} \end{aligned}$$

## Avisynth 2.5 Selected External Plugin Reference

$d(0;t(0)) = 1/(2*T) * \text{filter\_step}$  // general case  
 $d(0;t(v)) = (2*v+1)/(2*T) * \text{filter\_step}$  // general case ( $v \geq 0$ )

The left most source pixel which contributes to  $t(0)$  is  $s(\text{off}(0))$ . It follows that

$d(s(\text{off}(0));t(0)) = d(0;t(0)) + d(0;s(\text{off}(0)))$  //  $s(\text{off}(0))$  is located to the left of  $x=0$ ; thus  $\text{off}(0) \leq -1$   
 $= 1/(2*T) * \text{filter\_step} - (2*\text{off}(0) + 1)/2 * \text{filter\_step}$   
 $= (1/(2*T) - (2*\text{off}(0) + 1)/2) * \text{filter\_step}$   
 $= (1/(2*T) - \text{off}(0) - 1/2) * \text{filter\_step}$   
 $= (-\text{off}(0) + 1/(2*T) - 1/2) * \text{filter\_step}$

Thus

$b\_T(0) = -d(s(\text{off}(0));t(0))$   
 $= (\text{off}(0) - (1/(2*T) - 1/2)) * \text{filter\_step}$   
 $= (\text{floor}(1/(2*T) - 1/2) - \text{support} / \text{filter\_step} + 1 - (1/(2*T) - 1/2)) * \text{filter\_step}$

and for  $j > 0$  (in that case  $\text{off}(j) \geq 0$ )

$b\_T(j) = -d(s(\text{off}(j));t(j)) = -(d(0;t(j)) - d(0;s(\text{off}(j)))) = d(0;s(\text{off}(j))) - d(0;t(j))$   
 $= (2*\text{off}(j)+1)/2 * \text{filter\_step} - (2*j+1)/(2*T) * \text{filter\_step}$   
 $= ((2*\text{off}(j)+1)/2 - (2*j+1)/(2*T)) * \text{filter\_step}$   
 $= (\text{off}(j) + 1/2 - j/T - 1/(2*T)) * \text{filter\_step}$   
 $= (\text{off}(j) - (1/(2*T) - 1/2 + j/T)) * \text{filter\_step}$   
 $= ((\text{floor}(1/(2*T) - 1/2 + j/T) - \text{support} / \text{filter\_step} + 1) - (1/(2*T) - 1/2 + j/T)) * \text{filter\_step}$

Note that

$b\_T(j) = b\_T(j-1) - 1/T * \text{filter\_step}$  (general case), if the right-hand side is smaller than or equal to " $-\text{support}/\text{filter\_step}$ " then one (or  $1/\text{filter\_step}$ ) should be added to the right-hand side.

### 12.2.0.1 AviSynth's implementation

The pseudocode in AviSynth is as follows

```
// resample_functions.cpp
// the function GetResamplingPatternRGB is called twice:
// if source_height/target_height < source_width/target_width then the clip is resampled vertically first,
// then horizontally (and the other way around).

int* GetResamplingPatternRGB( int original_width, double subrange_start, double subrange_width,
int target_width, ResamplingFunction* func, IScriptEnvironment* env )
/**
 * This function returns a resampling "program" which is interpreted by the
 * FilteredResize filters. It handles edge conditions so FilteredResize
 * doesn't have to.
 */

the function returns a pointer to cur, where
```



## Avisynth 2.5 Selected External Plugin Reference

```

// *cur = [fir_filter_size, start_pos(j=0), f(b_T(j=0)[0])/total(j=0), f(b_T(j=0)[1])/total(j=0), ...,
f(b_T(j=0)[fir_filter_size])/total(j=0)),
//           start_pos(j=1), f(b_T(j=1)[0])/total(j=1), f(b_T(j=1)[1])/total(j=1), ...,
f(b_T(j=1)[fir_filter_size])/total(j=1)),
//           ...
//           start_pos(j=n-1), f(b_T(j=n-1)[0])/total(j=n-1), f(b_T(j=n-1)[1])/total(j=n-1),
..., f(b_T(j=n-1)[fir_filter_size])/total(j=n-1))]

filter_step = min(T; 1) // T = n/m
filter_support = support / filter_step
fir_filter_size = ceil(2*filter_support)
int* result = (int*) _aligned_malloc((1 + target_width*(1+fir_filter_size)) * 4, 64);
int* cur = result;
*cur++ = fir_filter_size;

pos_step = 1/T
// pos(j) = 1/(2*T) - 1/2 + j * pos_step
pos = 1/(2*T) - 1/2

for (int j=0; j<target_width; ++j) {
    ok_pos = max(0; min(m-1; pos)) // boundary condition [a]
    end_pos = int(pos + filter_support)
    if (end_pos > m-1) { // boundary condition [b]
        end_pos = m-1;
    }
    start_pos = end_pos - fir_filter_size + 1;
    if (start_pos < 0) { // boundary condition [c]
        start_pos = 0;
    }

    total = f((start_pos + 0 - ok_pos) * filter_step) + ... + f((start_pos + fir_filter_size - 1 - ok_pos) *
filter_step)
    for (int k=0; k<fir_filter_size; ++k) {
        // b_T(j)[k] = (start_pos(j) + k - ok_pos(j)) * filter_step = b_T(j)[0] + k * filter_step := b_T(j) + k *
min(T;1)
        // *cur++ = f(b_T(j)[k])/total(j)
        *cur++ = f((start_pos + k - ok_pos) * filter_step) / total;
    }

    pos += pos_step;
}

```

The implementation in AviSynth is the same as the described algorithm above, since:

```

filter_step = min(T; 1) // T = n/m = target pixels / source pixels
= 1 for upsampling
= T for downsampling

filter_support = support / filter_step
= support for upsampling
= support * 1/T for downsampling

```

```

fir_filter_size = ceil(2*filter_support)
pos_step = 1/T

pos(j) = 1/(2*T) - 1/2 + j * pos_step
        = 1/(2*T) - 1/2 + j/T

end_pos(j) = floor(pos(j) + filter_support)
            = floor( 1/(2*T) - 1/2 + j/T + filter_support )

start_pos(j) = end_pos(j) - fir_filter_size + 1
              = floor(1/(2*T) - 1/2 + j/T + filter_support) - fir_filter_size + 1 // floor(..) = int(..)
              = floor(1/(2*T) - 1/2 + j/T) + support / filter_step - ceiling(2*support / filter_step) + 1
              [= floor(1/(2*T) - 1/2 + j/T) + 0 - 1 + 1 = floor(1/(2*T) - 1/2 + j/T) // for NN]
              = floor(1/(2*T) - 1/2 + j/T) - support / filter_step + 1
              = off(j)

ok_pos(j) = max(0; min(m-1; pos(j)))
           [= pos(j) // without the boundary conditions ]

b_T(j) = (start_pos(j) - ok_pos(j)) * filter_step
        [= ((floor(1/(2*T) - 1/2 + j/T) - support / filter_step + 1) - (1/(2*T) - 1/2 + j/T)) * filter_step //
without the boundary conditions ]
        [= (floor(1/(2*T) - 1/2 + j/T) - (1/(2*T) - 1/2 + j/T)) * filter_step // for NN ]

```

These expressions (for start\_pos(j) and b\_T(j)) are the same expressions as in the derived algorithm. So, the only difference is caused by the boundary conditions:

```

ok_pos = max(0; min(m-1; pos)) // boundary condition [a]
end_pos = min(end_pos; m-1) // boundary condition [b]
start_pos = max(start_pos; 0) // boundary condition [c]

```

**Note that: pos is the position of the target pixel in source space. If it is located outside the source image, then it is shifted to the boundary of the source image. The conditions [b] and [c] mean that the filter will be applied to existing source pixels.**

## 12.3 Resizing Algorithms

### 12.3.1 Nearest Neighbour resampler

$$f(x) = 1 \text{ for } |x| < 1/2$$

$$= 0 \text{ elsewhere}$$

So, the condition of symmetry (and  $\sum_{k=-\infty}^{\infty} f(x-k) = 1$ ) is already included above. In our example we get

$$t(2) = s(1) = \sum_n s_3(n) * f(n - 1/3) = s_3(0) * f(0 - 1/3)$$

$$= s_3(0) * f(-1/3)$$

$$= s_3(0) * 1$$

$$= s_3(0)$$

## Avisynth 2.5 Selected External Plugin Reference

So it picks the source pixel which is the closest to the target pixel (note that there is only one source pixel within a distance of 1/2 of the target pixel). Hence the name of the filter. This is also called a point sampler.



Note that the resampler is not differentiable at the endpoints ( $|x|=1/2$ ).

### 12.3.2 Bilinear resampler

$$f(x) = f_1(x) \text{ for } |x| < 1 \\ = 0 \quad \text{elsewhere}$$

$$f_1(x) = P * |x| + Q$$

So, the condition of symmetry is already included above.

These two free parameters are solved by requiring (1) the filter to be continuous and (2) if all the samples are a constant value then the reconstruction should be a flat signal. The latter means that if the source picture has a solid color, then the target picture has the same solid color. This is only possible if the sum of the coefficients of the filter is one.

(1) This means that the value and first derivative are continuous at  $|x| = 0$  and  $|x| = 1$ . It results in the following condition:

$$f_1(1) = 0 \Rightarrow P + Q = 0 \\ f_1'(0) = \text{constant} \Rightarrow \\ f_1'(x) = -P \text{ for } x <= 0 \\ f_1'(x) = P \text{ for } x >= 0 \\ \text{thus for } x=0: -P = P \Rightarrow P = 0 \text{ (so a continuous derivate can't be possible)}$$

(2) This means that  $\sum_{k=-\infty}^{\infty} f(x-k) = 1$  for all  $x$ :

$$f_1(x) + f_1(x+1) = 1 \Rightarrow 2*Q + P = 1$$

Solving the two conditions gives

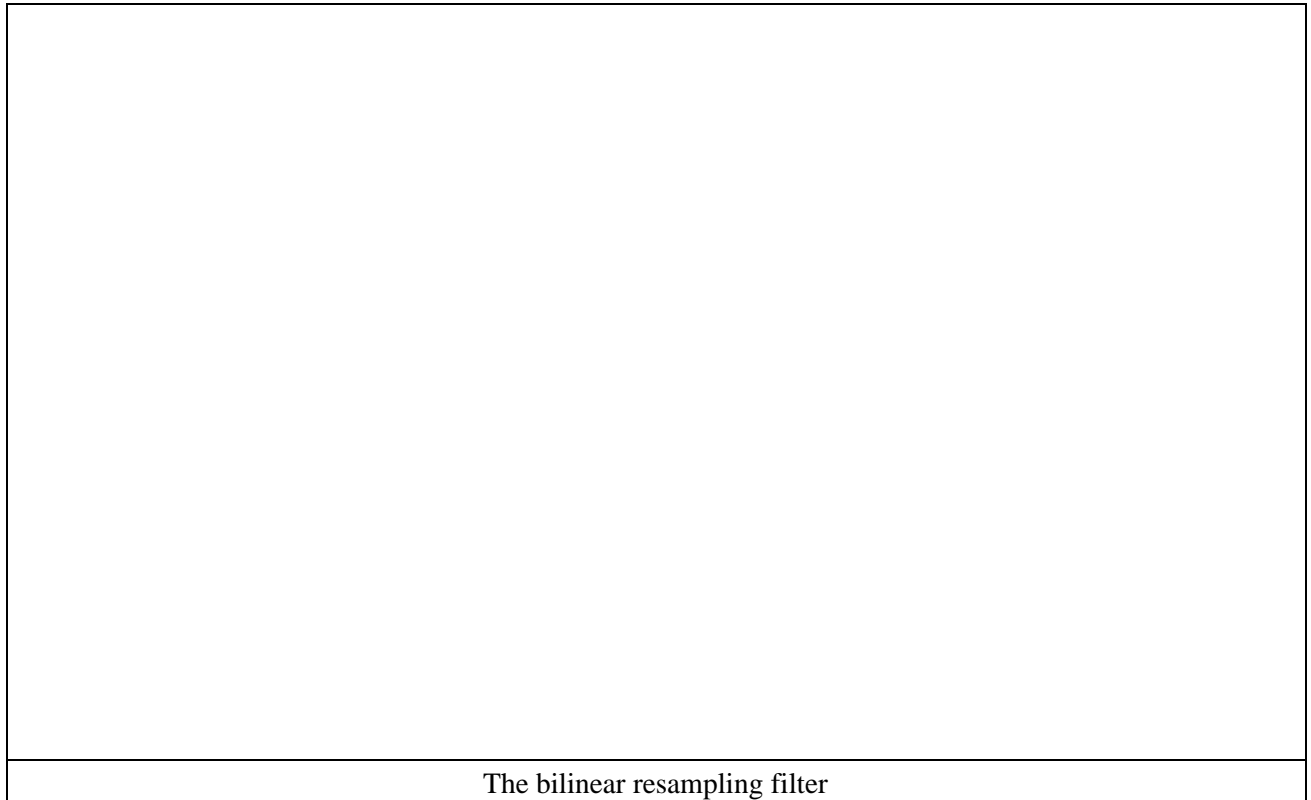
$$\begin{aligned} > \text{eqn} := \{P+Q=0, 2*Q+P=1\}; \\ > \text{solve}(\text{eqn}, \{P, Q\}); \end{aligned}$$

$$\{P=1, Q=-1\}$$

This results in the following bilinear resampler

$$f(x) = 1 - |x| \text{ for } |x| < 1$$

$$f(x) = 0 \text{ elsewhere}$$



### 12.3.3 Bicubic resampler

[Mitchell and Netravali](#) eventually introduced a quite popular family of cubic splines, the BC-splines. A cubic spline filter is in its general form given by

$$f(x) = f1(x) \text{ for } |x| < 1$$

$$= f2(x) \text{ for } 1 \leq |x| < 2$$

$$= 0 \text{ elsewhere}$$

$$f1(x) = P * |x|^3 + Q * |x|^2 + R * |x| + S$$

$$f2(x) = T * |x|^3 + U * |x|^2 + V * |x| + W$$

So, the condition of symmetry is already included above.

These eight free parameters are reduced to two by requiring (1) the filter to be smooth and (2) if all the samples are a constant value then the reconstruction should be a flat signal. The latter means that if the source picture has a solid color, then the target picture has the same solid color. This is only possible if the sum of the coefficients of the filter is one.

(1) This means that the value and first derivative are continuous at  $|x| = 0$ ,  $|x| = 1$  and  $|x| = 2$ . It results in the following four conditions:

## Avisynth 2.5 Selected External Plugin Reference

$$\begin{aligned}
 f1(1) = f2(1) &\Rightarrow \mathbf{P + Q + R + S = T + U + V + W} \\
 f2(2) = 0 &\Rightarrow \mathbf{8*T + 4*U + 2*V + W = 0} \\
 f1'(1) = f2'(1) &\Rightarrow \mathbf{3*P + 2*Q + R = 3*T + 2*U + V} \\
 f1'(0) = \text{constant} &\Rightarrow \\
 f1'(x) &= -3*P*x^2 + 2*Q*x - R \text{ for } x \leq 0 \\
 f1'(x) &= 3*P*x^2 + 2*Q*x + R \text{ for } x > 0 \\
 \text{thus for } x=0: & -R = R \Rightarrow \mathbf{R = 0}
 \end{aligned}$$

(2) This means that  $\sum_{k=-\infty}^{\infty} f(x-k) = 1$  for all  $x$ :

$$f2(x) + f1(x+1) + f1(x+2) + f2(x+3) = 1 \Rightarrow$$

$$(2*U + 9*T + 2*Q + 3*P) * x^2 + (6*U + 27*T + 6*Q + 9*P) x + 27*T + 9*U + 3*V + 2*W + 7*P + 5*Q + R + 2*S = 1$$

This results in the following two different conditions

$$\begin{aligned}
 (2*U + 9*T + 2*Q + 3*P) &= \mathbf{0} \\
 9*T + 5*U + 3*V + 2*W + P + Q + R + 2*S &= \mathbf{1} \\
 27*T + 9*U + 3*V + 2*W + 7*P + 5*Q + R + 2*S &= \mathbf{1} \text{ (redundant because this holds if the first two} \\
 \text{conditions are met)}
 \end{aligned}$$

Solving the six equations (in terms of Q and S) using Maple gives:

```

> eqn:={P+Q+R+S=T+U+V+W, 8*T+4*U+2*V+W=0, 3*P+2*Q+R=3*T+2*U+V, -R=R,
2*U+9*T+2*Q+3*P=0, 9*T+5*U+3*V+2*W+P+Q+R+2*S=1}:
> solve(eqn,{P,R,T,U,V,W});

```

$$\{Q = Q, S = S, R = 0, P = 1/2 - Q - 3/2 S, T = 5/2 - Q - 11/2 S, V = 18 - 8 Q - 42 S, U = -12 + 5 Q + 27 S, W = -8 + 4 Q + 20 S\}$$

Setting

$$\begin{aligned}
 S &= 1 - B/3 \\
 Q &= -3 + 2*B + C
 \end{aligned}$$

gives us the other coefficients as function of B and C:

$$\begin{aligned}
 P &= 1/2 - Q - 3/2*S = 2 - 3/2*B - C \\
 T &= 5/2 - Q - 11/2*S = -1/6*B - C \\
 V &= 18 - 8*q - 42*s = -2*B - 8*C \\
 U &= -12 + 5*q + 27*s = B + 5*C \\
 W &= -8 + 4*q + 20*s = 4/3*B + 4*C
 \end{aligned}$$

This results in the following family of cubic filters

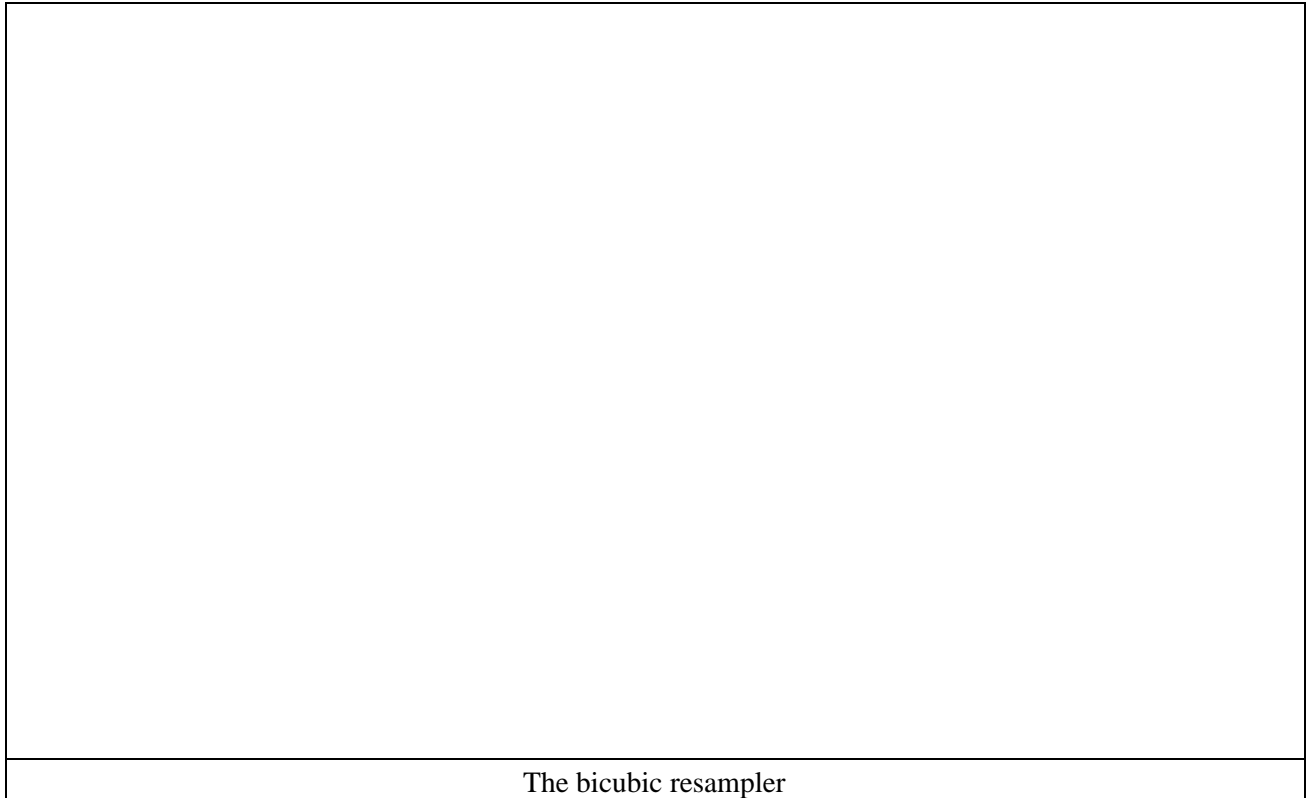
$$\begin{aligned}
 f(x) &= [ (12-9*B-6*C)*|x|^3 + (-18+12*B+6*C)*|x|^2 + (6-2*B) ] / 6 \text{ for } |x| < 1 \\
 &= [ (-B-6*C)*|x|^3 + (6*B+30*C)*|x|^2 + (-12*B-48*C)*|x| + (8*B+24*C) ] / 6 \text{ for } 1 <= |x| < 2 \\
 &= 0 \text{ elsewhere}
 \end{aligned}$$

Some of the values for B and C correspond to well known filters (source: [\[1\]](#) [\[2\]](#)). For example:

### 12.3.3 Bicubic resampler

## Avisynth 2.5 Selected External Plugin Reference

- B=1, C=0: cubic B-spline;
- B=0, C=C: cardinal splines (C=1/2 the Catmull-Rom spline; C=0 the family of Duff's tensioned B-splines)



### 12.3.4 Windowed sinc resamplers

Windows are used to approximate an "ideal" impulse response of infinite duration, such as a sinc function, to an impulse response of finite duration. Recall that  $\text{sinc}(x)$  is maximal and equal to one for  $x=0$ . It has such a shape that the source pixels closest to the target pixel (that is the center of the resampler) will contribute the most. This implies that the window should also fulfil this property. All windows mentioned here ([source](#)) are symmetric, and thus are the resamplers also symmetric.

#### 12.3.4.1 Sinc resampler

The sinc resampler is just a truncated sinc function.

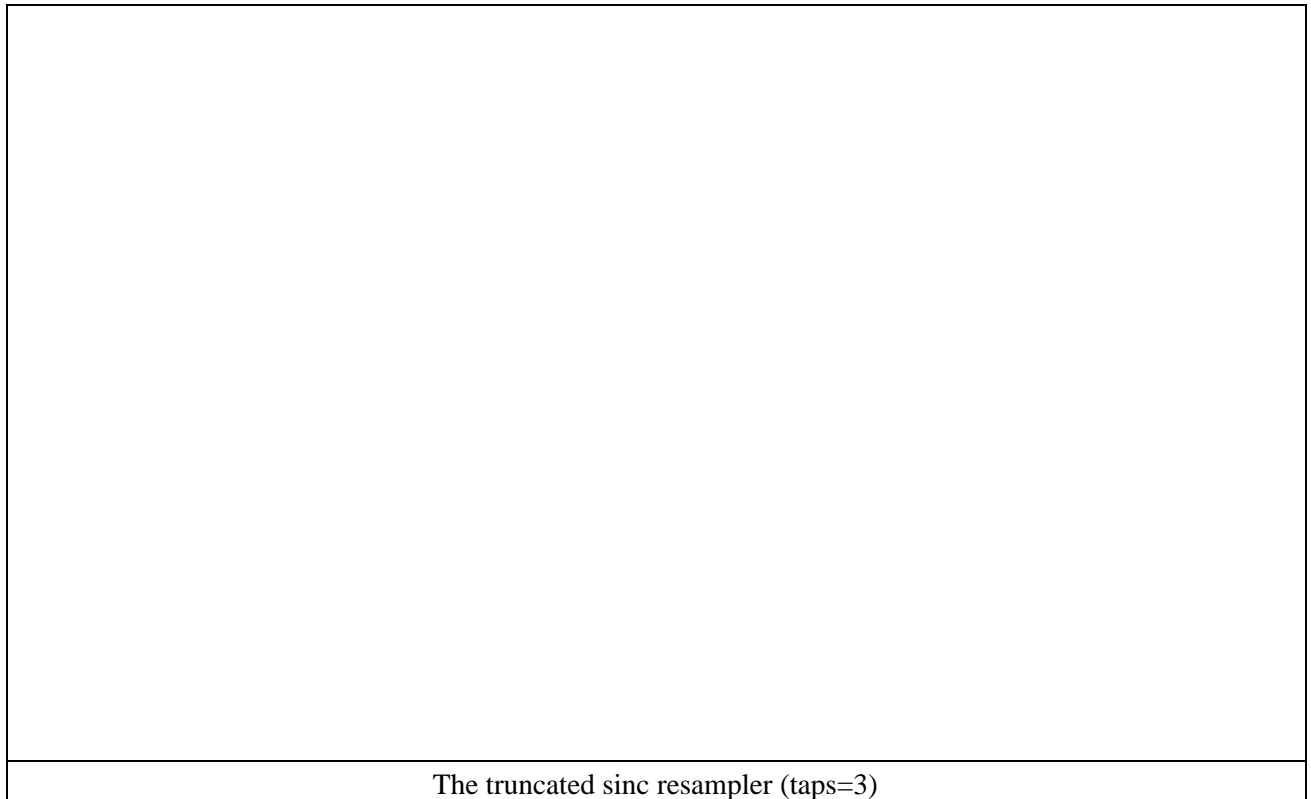
The box window is given by

$$w(x, \text{taps}) = \begin{cases} 1 & \text{for } |x| < \text{taps} \\ 0 & \text{elsewhere} \end{cases}$$

and the resampler is given by

$$f(x) = \text{sinc}(x) * w(x, \text{taps})$$

**So, the condition of symmetry (and  $\sum_{k=-\infty}^{\infty} f(x-k) = 1$ ) is already included above.** Note that the resampler is continuous everywhere, but not differentiable at the endpoints ( $|x|=\text{taps}$ ).



The truncated sinc resampler (taps=3)

#### 12.3.4.2 Lanczos resampler

The Lanczos window is given by

$$w(x, \text{taps}) = \text{sinc}(x/\text{taps}) \text{ for } |x| < \text{taps} \\ = 0 \text{ elsewhere}$$

and the resampler is given by

$$f(x) = a(\text{taps}) * \text{sinc}(x) * w(x, \text{taps})$$

with

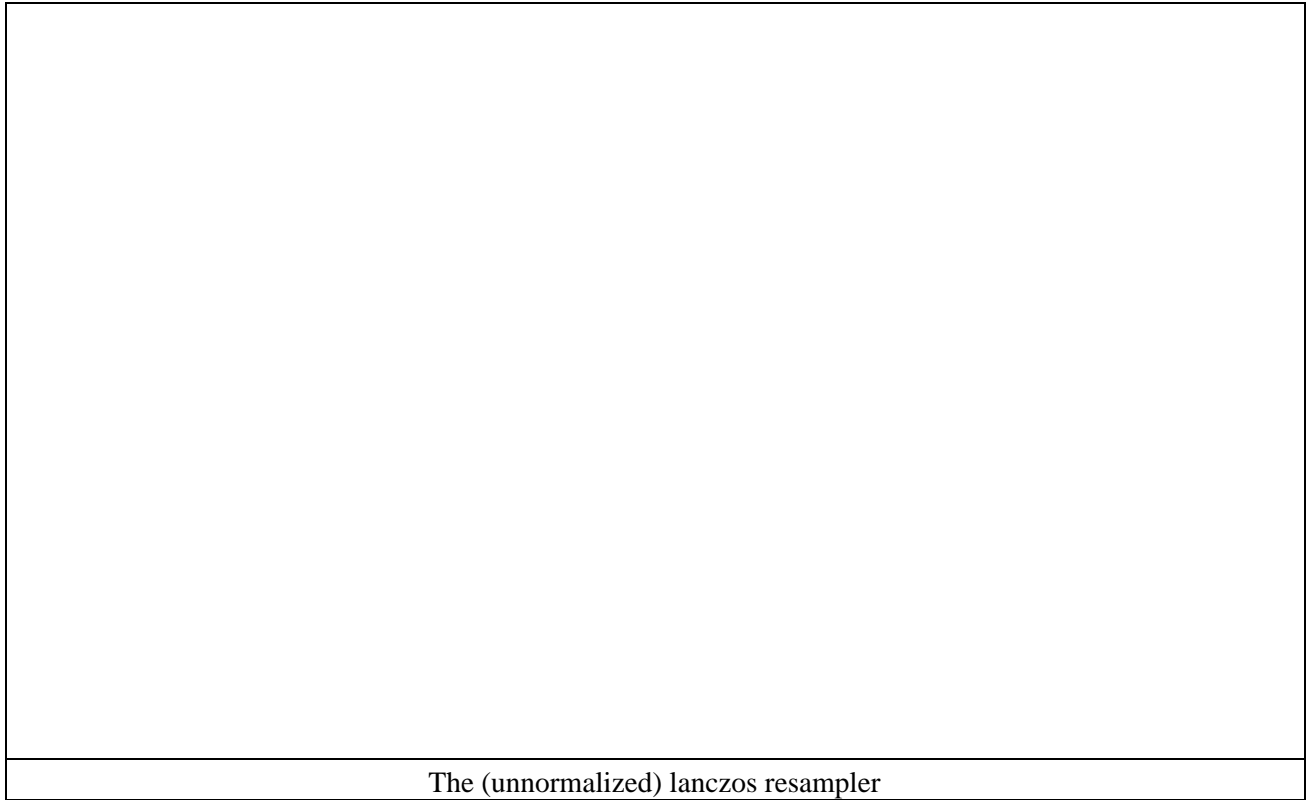
$$\text{sinc}(x) = \sin(\pi * x) / (\pi * x) \text{ for } x \neq 0 \\ = 1 \text{ elsewhere}$$

The resampler is differentiable everywhere (also at the points  $|x| = \text{taps}$ ??).

$a(\text{taps})$  needs to be chosen such that the following condition is fulfilled:

$$\sum_{k=-\infty}^{\infty} f(x-k) = 1 \text{ for all } x$$

Note that  $a(x, \text{taps})$  is very close to one for  $\text{taps} \geq 3$ .



The (unnormalized) lanczos resampler

#### 12.3.4.3 Blackman resampler

The blackman window is given by

$$w(x, \text{taps}) = (1-a)/2 + 1/2*\cos(x/\text{taps}) + a/2*\cos(2*x/\text{taps}) \text{ with } a = 0.16 \text{ for } |x| < \text{taps}$$

$$= 0 \text{ elsewhere}$$

and the resampler is given by

$$f(x) = a(\text{taps}) * \text{sinc}(x) * w(x, \text{taps})$$

Note that the resampler is continuous everywhere, but not differentiable at the endpoints ( $|x| = \text{taps}$ ).  $a(\text{taps})$  needs to be chosen such that the following condition is fulfilled:

$$\sum_{k=-\infty}^{\infty} f(x-k) = 1 \text{ for all } x$$





The (unnormalized) blackman resampler

### 12.3.5 Spline resampler

The Spline16/36/64 resamplers were introduced by Helmut Dersch, the author of [Panorama Tools](#). It fits a cubic spline through the sample points and then derives the filter kernel from the resulting blending polynomials.

Let's consider Spline16 as an example which uses 4 sample points  $(-1,0,1,2)$  (the others are derived in a similar way). Consider three (cubic) splines for the interval:

$$S1(x) = c[3]*x^3 + c[2]*x^2 + c[1]*x + c[0] \text{ for } x \text{ in } [-1,0]$$

$$S2(x) = a[3]*x^3 + a[2]*x^2 + a[1]*x + a[0] \text{ for } x \text{ in } [0,1]$$

$$S3(x) = b[3]*x^3 + b[2]*x^2 + b[1]*x + b[0] \text{ for } x \text{ in } [1,2]$$

Let's define (under the condition that the polynomials will be continuous everywhere):

- $y0 = S1(-1)$
- $y1 = S1(0) = S2(0)$
- $y2 = S2(1) = S3(1)$
- $y3 = S3(2)$

Under the condition that the polynomials have continuous first and second derivatives and that the endpoints have a zero second derivative we will get the following additional conditions

- $S1'(0) = S2'(0)$
- $S2'(1) = S3'(1)$
- $S1''(0) = S2''(0)$
- $S2''(1) = S3''(1)$

## Avisynth 2.5 Selected External Plugin Reference

- $S1''(-1) = 0$
- $S3''(2) = 0$

The coefficients,  $(a[j])_j$ , are fixed by these boundary conditions. This results in the following set equations:

$$\begin{aligned} -c_3 + c_2 - c_1 + c_0 &= y_0 \\ c_0 &= y_1 \\ a_0 &= y_1 \\ a_3 + a_2 + a_1 + a_0 &= y_2 \\ b_3 + b_2 + b_1 + b_0 &= y_2 \\ 8*b_3 + 4*b_2 + 2*b_1 + b_0 &= y_3 \\ c_1 - a_1 &= 0 \\ 3*a_3 + 2*a_2 + a_1 - 3*b_3 - 2*b_2 - b_1 &= 0 \\ 2*c_2 - 2*a_2 &= 0 \\ 6*a_3 + 2*a_2 - 6*b_3 - 2*b_2 &= 0 \\ -6*c_3 + 2*c_2 &= 0 \\ 12*b_3 + 2*b_2 &= 0 \end{aligned}$$

Solving this set of equations with Maple results in:

```
> solution:=solve(eqn,{a0,a1,a2,a3,b0,b1,b2,b3,c0,c1,c2,c3});
> a0s:=subs(solution, a0); a1s:=subs(solution, a1); a2s:=subs(solution, a2); a3s:=subs(solution, a3);

a0s := y1
a1s := - 7/15 y0 - 1/5 y1 + 4/5 y2 - 2/15 y3
a2s := 4/5 y0 - 9/5 y1 + 6/5 y2 - 1/5 y3
a3s := - 1/3 y0 + y1 - y2 + 1/3 y3
```

We are only interested in the spline for  $[0,1]$ . The values of the coefficients of the other two splines are not important (so they are not shown here). So, it turns out that the coefficients  $(a[j])_j$  (and thus the spline itself) can be expressed as a linear combination of the control points  $y[j]$ . This means that

$$S2(x) = w_0(x) * y_0 + w_1(x) * y_1 + w_2(x) * y_2 + w_3(x) * y_3$$

where

$$\begin{aligned} w_0(x) &= - 1/3 * x^3 + 4/5 * x^2 - 7/15 * x \\ w_1(x) &= x^3 - 9/5 * x^2 - 1/5 * x + 1 \\ w_2(x) &= -x^3 + 6/5 * x^2 + 4/5 * x \\ w_3(x) &= 1/3 * x^3 - 1/5 * x^2 - 2/15 * x \end{aligned}$$

Since  $w_0, w_1, w_2, w_3$  blend the points  $y_0, y_1, y_2, y_3$  together, they are called blending polynomials (or basis functions). As a result of the boundary conditions we have:

$$\begin{aligned} y_1 = S2(0) &= w_0(0) * y_0 + w_1(0) * y_1 + w_2(0) * y_2 + w_3(0) * y_3 \Rightarrow w_0(0)=0, w_1(0)=1, w_2(0)=0, w_3(0)=0 \\ y_2 = S2(1) &= w_0(0) * y_0 + w_1(0) * y_1 + w_2(0) * y_2 + w_3(0) * y_3 \Rightarrow w_0(0)=0, w_1(0)=0, w_2(0)=1, w_3(0)=0 \end{aligned}$$

It's also true that the sum of the blending polynomials is equal to one, that is

$$w_0(x) + w_1(x) + w_2(x) + w_3(x) = 1$$

## Avisynth 2.5 Selected External Plugin Reference

(The reason is that, if all the  $y[i]$  are equal (to  $k$ , say), then it is clear that the solution  $S_i(x)=k$  (for all  $i,x$ ) satisfies the initial constraints. (Intuitively, interpolating a set of constant values gives that same value.) Since  $S_2(x) = \sum_i (w_i(x)*y[i]) = k * \sum_i w_i(x)$ , it follows that the sum of the weights is 1.)

What happens if you replace  $y[j]$  by  $y[3-j]$  (for  $j=0,1$ )? In that case the polynomial  $S_2$  will be mirrored at  $x=1/2$ , since (1)  $S_2$  is fixed by  $(y_0,y_1,y_2,y_3)$  and (2)  $z_0=z_1$ . In other words:

$$S_2[y_0;y_1;y_2;y_3](x) = S_2[y_3;y_2;y_1;y_0](1-x)$$

since the mapping  $x \mapsto 1-x$  results in the same mirroring of  $S_2$  at  $x=1/2$ . It follows that

$$w_2(x) = w_1(1-x), w_3(x) = w_0(1-x)$$

The filter kernel is defined as

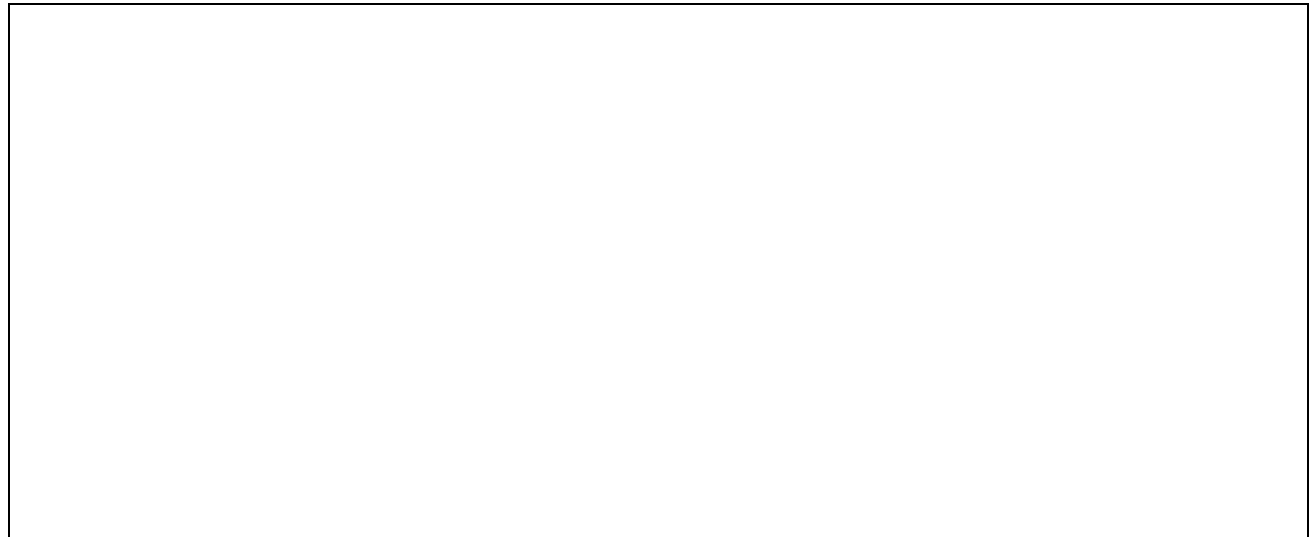
$$\begin{aligned} f(x) &= w_1(x) = x^3 - 9/5 * x^2 - 1/5 * x + 1 \quad \text{for } |x| < 1 \\ f(x) &= w_0(x-1) = -1/3 * (x-1)^3 + 4/5 * (x-1)^2 - 7/15 * (x-1) \quad \text{for } 1 \leq |x| < 2 \\ f(x) &= 0 \quad \text{elsewhere} \end{aligned}$$

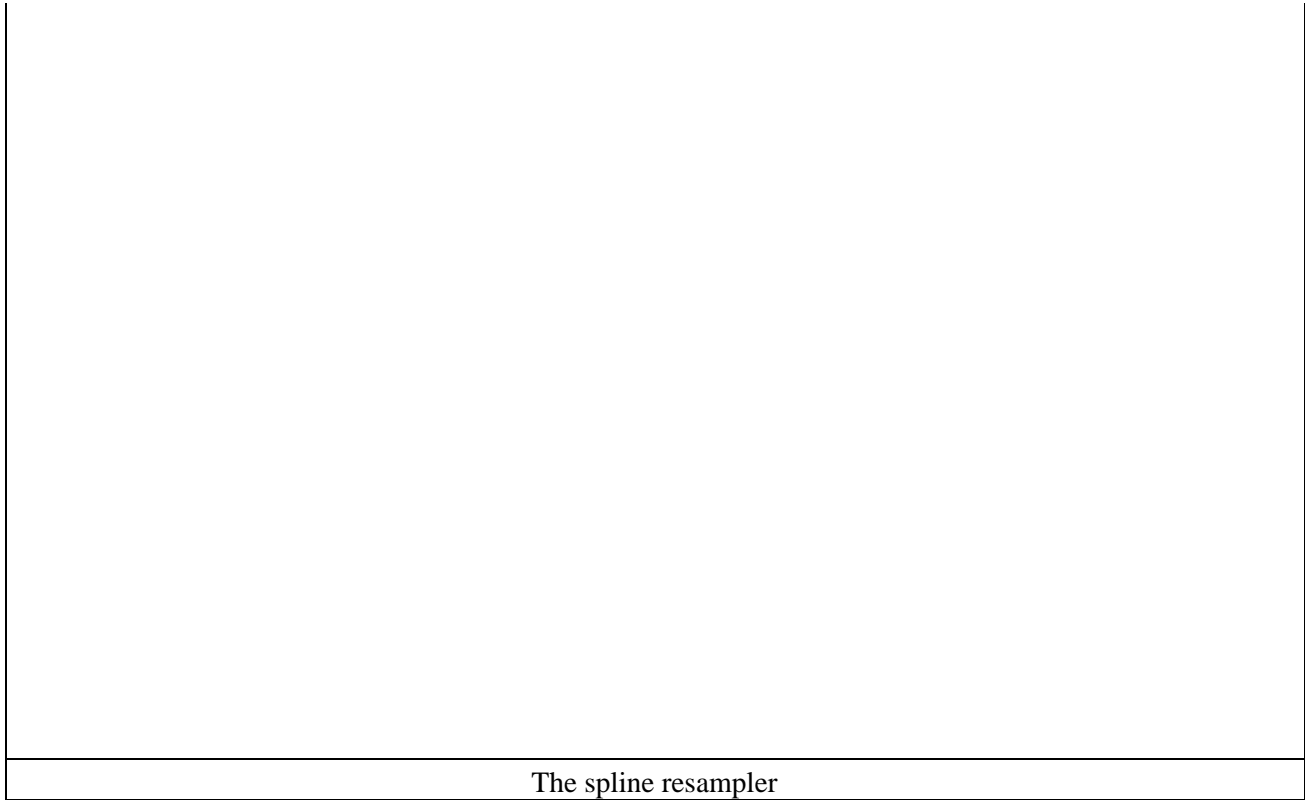
Note that the weights are just continuous, but not continuous differentiable at the joints.

In general for Spline  $k^2$  ( $k$  even), we start with  $k-1$  bicubic splines. They are located at  $(-k/2+1, -k/2+2), \dots, (k/2-1, k/2)$ . The bending polynomials are calculated from the polynomial located at  $[0,1]$ , which is expressed as a linear combination of  $(y_0, \dots, y(k-1))$ . The boundary conditions are:

$$\begin{aligned} S[1](-k/2+1) &= y_0 \\ S[1](-k/2+2) &= y_1 \\ S[2](-k/2+2) &= y_1 \\ &\dots \\ S[k-2](k/2-1) &= y(k-2) \\ S[k-2](k/2-1) &= y(k-2) \\ S[k-1](k/2) &= y(k-1) \end{aligned}$$

and also continuity of first and second derivatives, where the second derivatives of the two endpoints are zero. This results in  $k$  blending functions. More discussion can be found [here](#).





The spline resampler

### 12.3.6 Gaussian resampler

$$f(x, q) = a(q) * 2^{(-q*x^2)} \text{ with } q = p*0.1 \text{ for } |x| < \text{support}$$

$$= 0 \text{ elsewhere}$$

The support is chosen in such a way that  $2^{(-q*\text{support}^2)} = 0$ , or as an approximation  $2^{(-q*\text{support}^2)} = 1.0/512.0$  (i.e 0.5 bit). It follows that for example:

p	0.1	5.625	22.5	30	100
support	30.0	4.0	2.0	1.73	0.949

In AviSynth's implementation of GaussResize, it always has support=4.0, regardless of the value of "p". The resampler is differentiable everywhere except at  $|x|=\text{support}$ .  $a(q)$  needs to be chosen such that the following condition is fulfilled:

$$\sum_{k=-\infty}^{\infty} f(x-k) = 1 \text{ for all } x$$



The (unnormalized) gaussian resampler

## 12.4

When discussing resampling, we explained the Nyquist–Shannon sampling theorem. But this theorem is about signals, while we applied it to images. The reason is that an image is a signal with respect to position.

A resampling filter is a low–pass filter with a filter cutoff less than or equal to the lower of the [Nyquist rates](#) of the source and destination spaces (we use  $f_c = \min(f_{src}, f_{dst})$  with  $f_{src} = 2 / \text{'number of source pixels'}$  and  $f_{dst} = 2 / \text{'number of destination pixels'}$ ). The triangle filter has a width of 2.0, the bicubic 4.0, and the Lanczos3 6.0 samples. The filter kernel is sampled in source space, whether or not it is stretched depends on whether the source or destination has the higher sampling rate.

For  $dst > src$  (interpolation), the filter cutoff is  $0.5 * f_{src} / f_{dst} = 0.5$  and thus the filter kernel size is a constant 2, 4, or 6 taps, respectively.

For  $src > dst$  (decimation), the filter cutoff is  $0.5 * f_{dst} / f_{src}$  in source space. This leads to kernel widths of  $2.0/T$ ,  $4.0/T$ , and  $6.0/T$  taps with  $T = \text{'number of destination pixels'} / \text{'number of source pixels'}$ . The taps are then rounded up to an even number, padding the kernel with zeros, for convenience in the code. [source](#).

Thus a resampling filter is a low–pass FIR filter, sampled at source rate and normalized to unity gain (meaning that the sum of the coefficients of the filter should be one). What this all means will hopefully become clear in the remainder of this document.

<http://books.google.nl/books?id=T3yqrSpIJAC&pg=PA362&lpg=PA362&dq=%22Cutoff+frequency%22+sampling>

Calculate frf for (1) interpolation and (2) decimation for some filter (say triangle) for several int/dec factors. Look at the cutt–off frequency.

## Avisynth 2.5 Selected External Plugin Reference

When upsampling, the [frequency response](#) of the bilinear resampler is given by

$$H(z) = f(x+1) + f(x)/z \quad (\text{for some } x \text{ in } [-1,0])$$

with  $z = \exp(2\pi i f)$ .

When downsampling, the [frequency response](#) of the bilinear resampler is given by

$$H(z) = f(x+5/3) + f(x+4/3)/z + f(x+1)/z^2 + f(x+2/3)/z^3 + f(x+1/3)/z^4 + f(x)/z^5 \quad (\text{for some } x \text{ in } [-1,-2/3])$$

When plotting  $|H(z)|$  this results in

<picture>

<http://www.dspguru.com/info/faqs/fir/props.htm>

**\$Date: \$**

# 13 Sampling

## 13.1 1. Color format conversions and the Chroma Upsampling Error

The following figures show errors, which are examples of **the Chroma Upsampling Error**, called this way because the video is upsampled incorrectly (interlaced YV12 upsampled as progressive or vice versa). As a result, you will often see gaps on the top and bottom of colored objects and "ghost" lines floating above or below the objects.



figure 1a: example of interlaced source (YV12) being upsampled as progressive video (YUY2) (from <http://zenaria.com/gfx/>)



figure 1b: the same image with correct chroma upsampling (from <http://zenaria.com/gfx/>)



figure 2a: example of progressive source (YV12) being upsampled as interlaced video (YUY2)



figure 2b: the same image with correct chroma upsampling

In this section, it will be shown what causes it, and how to fix it. Where fixing mean making it less visible, because it's not possible to correct it completely.

References:

[\[The Chroma Upsampling Error\]](#)

[\[The Chroma Upsampling Error – Television and Video Advice\]](#)

### 13.1.1 1.1 Chroma Upsampling Error (or CUE)

As previously stated, the Chroma Upsampling Error occurs when you convert from (trully) interlaced YV12 to mostly any other format and the converter thinks the video is progressive. Or, the other way around, if material is progressive (or interlaced encoded as progressive), and upsampled as interlaced. This is however not as bad as the other way around.

When VDub previews your video, it will need to convert it to RGB. Since AviSynth delivers YV12, it asks the codec (for example XviD or DivX) to convert YV12 to RGB. The codec however ALWAYS upsamples progressively. Hence you will get artifacts in VDub preview on interlaced YV12 material. This is however not



present in the YV12 video (or in the resulting encoding). To confirm this, let AviSynth do the conversion by adding `ConvertToRGB(interlaced=true)` at the end of your script.

### 13.1.2 1.2 Correcting video having the Chroma Upsampling Error

You will have to blur the chroma in some way (leaving the luma intact).

For example (using `tomsocomp.dll`):

```
AviSource(...)
MergeChroma(TomsMoComp(-1,5,0))
```

## 13.2 2. Theoretical Aspects

In this section, the chroma placement will be explained, how this is related to subsampling (RGB → YUY2 → YV12) and how the upsampling is done in AviSynth.

It should also explain in detail why the CUE occurs. To summarize the latter, the problem is that there is a difference between YV12 progressive and YV12 interlaced, because the chroma is shared vertically between neighboring pixels.

See also <http://forum.doom9.org/showthread.php?s=&threadid=52151&highlight=upsampling>.

### 13.2.1 2.1 The color formats: RGB, YUY2 and YV12

In order to be able to understand how YV12 ↔ YUY2 sampling works and why it matters whether your source is interlaced or progressive, the YV12/YUY2 color formats will be discussed first. It's not important here how they are stored in your memory. Information about that can be found here: [ColorSpaces](#).

#### 13.2.1.1 YUV 4:4:4 color format

The term 4:4:4 denotes that for every four samples of the luminance (Y), there are four samples each of U and V. Thus each pixel has a luminance value (Y), a U value (blue difference sample or Cb) and a V value (red difference sample or Cr). Note, "C" is just a chroma sample (UV-sample).

The layout of a 4:4:4 encoded image looks as follows

frame	line
YC YC YC YC	line 1
YC YC YC YC	line 2
YC YC YC YC	line 3
YC YC YC YC	line 4

#### 13.2.1.2 YUY2 color format

YUY2 (or YUYV) is a 4:2:2 format. The term 4:2:2 denotes that for every four samples of the luminance (Y), there are two samples each of U and V, giving less chrominance (color) bandwidth in relation to luminance. So for each pixel, it is horizontally sharing UV (chroma) with a neighboring pixel.

## Avisynth 2.5 Selected External Plugin Reference

The layout of a 4:2:2 encoded image looks as follows

frame	line
YC Y YC Y	line 1
YC Y YC Y	line 2
YC Y YC Y	line 3
YC Y YC Y	line 4

### 13.2.1.3 YV12 color format

For the YV12 color format, there's a difference between progressive and interlaced. The cause is that chroma values are also shared vertically between two neighboring lines.

YV12 is a 4:2:0 format. The term 4:2:0 denotes that for every four samples (two horizontal and two vertical) of the luminance (Y), there is one sample each of U and V, giving less chrominance (color) bandwidth in relation to luminance.

#### YV12 progressive

For each pixel, it is horizontally sharing UV (chroma or C) with a neighboring pixel and vertically sharing UV with the neighboring line (thus line 1 with line 2, line 3 with 4, etc).

The layout of a progressive 4:2:0 encoded image looks as follows (MPEG 2 scheme – see below)

frame	line
Y_Y_Y_Y	line 1
C__C__	
Y_Y_Y_Y	line 2
Y_Y_Y_Y	line 3
C__C__	
Y_Y_Y_Y	line 4

#### YV12 interlaced

For each pixel, it is horizontally sharing UV (chroma or C) with a neighboring pixel and vertically sharing UV with the next to neighboring line (thus line 1t with line 3t, line 2b with 4b, etc).

The layout of a interlaced 4:2:0 encoded image looks as follows (MPEG 2 scheme – see below)

frame	line
Y_Y_Y_Y	line 1t
C__C__	
Y_Y_Y_Y	line 2b
Y_Y_Y_Y	line 3t
C__C__	

## Avisynth 2.5 Selected External Plugin Reference

Y_Y_Y_Y	line 4b

or

field 1	field 2	line
Y_Y_Y_Y		line 1t
C__C__		
	Y_Y_Y_Y	line 2b
Y_Y_Y_Y		line 3t
	C__C__	
	Y_Y_Y_Y	line 4b

### 13.2.2 2.2 Subsampling

Subsampling is used to reduce the storage and broadcast bandwidth requirements for digital video. This is effective for a !YCbCr signal because the human eye is more sensitive for changes in black and white than for changes in color. So drastically reducing the color info shows very little difference. YUY2 and YV12 are examples of reduced color formats.

#### 13.2.2.1 RGB -> YUY2 conversion

More about RGB -> YUV color conversions can be found here: [ColorConversions](#).  
Recall the layout of a 4:4:4 encoded image

frame	line
YC1 YC2 YC3 YC4	line 1
YC1 YC2 YC3 YC4	line 2
YC1 YC2 YC3 YC4	line 3
YC1 YC2 YC3 YC4	line 4

In AviSynth, the default mode is using a 1-2-1 kernel to interpolate chroma, that is

$$C1x = (C1+C1+C1+C2)/4 \text{ (C1 is used three times, since this is the border)}$$

$$C3x = (C2+C3+C3+C4)/4$$

$$C5x = (C4+C5+C5+C6)/4$$

The 4:2:2 encoded image becomes

frame	line
Y1C1x Y2 Y3C3x Y4	line 1
Y1C1x Y2 Y3C3x Y4	line 2
Y1C1x Y2 Y3C3x Y4	line 3
Y1C1x Y2 Y3C3x Y4	line 4

## Avisynth 2.5 Selected External Plugin Reference

The other mode [ConvertBackToYUY2](#) uses chroma from the left pixel, thus

frame	line
Y1C1 Y2 Y3C3 Y4	line 1
Y1C1 Y2 Y3C3 Y4	line 2
Y1C1 Y2 Y3C3 Y4	line 3
Y1C1 Y2 Y3C3 Y4	line 4

*Note (as with the layout of other formats) the position of the chroma values, represent the WEIGHT result of the subsampling.*

### YUY2 → YV12 interlaced conversion

Recall the layout of a interlaced 4:2:0 encoded image, but with the weights included:

frame	line	weights
Y_Y_Y_Y	line 1t	
C__C__		chroma of YUY2_lines $(0.75)*1t +$ $(0.25)*3t$
Y_Y_Y_Y	line 2b	
Y_Y_Y_Y	line 3t	
C__C__		chroma of YUY2_lines $(0.25)*2b +$ $(0.75)*4b$
Y_Y_Y_Y	line 4b	

or

field 1	field 2	line	weights
Y_Y_Y_Y		line 1t	
C__C__			chroma of YUY2_lines $(0.75)*1t +$ $(0.25)*3t$
	Y_Y_Y_Y	line 2b	
Y_Y_Y_Y		line 3t	
	C__C__		chroma of YUY2_lines $(0.25)*2b +$ $(0.75)*4b$
	Y_Y_Y_Y	line 4b	

## Avisynth 2.5 Selected External Plugin Reference

--	--	--	--

*Note (as with the layout of other formats) the position of the chroma values, represent the WEIGHT as a result of the subsampling.*

Thus the chroma is stretched across two luma lines in the same field!

### YUY2 → YV12 progressive conversion

Recall the layout of a 4:2:0 encoded image

frame	line	weights
Y_Y_Y_Y	line 1	
C__C__		chroma of YUY2_lines (0.5)*1 + (0.5)*2
Y_Y_Y_Y	line 2	
Y_Y_Y_Y	line 3	
C__C__		chroma of YUY2_lines (0.5)*3 + (0.5)*4
Y_Y_Y_Y	line 4	

*Note (as with the layout of other formats) the position of the chroma values, represent the WEIGHT result of the subsampling.*

Thus the chroma is stretched across two luma lines in the same frame!

## 13.2.3 2.3 Upsampling

### 13.2.3.1 YUY2 conversion → RGB

Recall the layout of a 4:2:2 encoded image

frame	line
Y1C1 Y2 Y3C3 Y4	line 1
Y1C1 Y2 Y3C3 Y4	line 2
Y1C1 Y2 Y3C3 Y4	line 3
Y1C1 Y2 Y3C3 Y4	line 4

For the 4:2:2 → 4:4:4 conversion, the missing chroma samples are interpolated (using a 1–1 kernel), that is

$$C2x = (C1+C3)/2$$

$$C4x = (C3+C5)/2$$

and the existing chroma samples are just copied.

## Avisynth 2.5 Selected External Plugin Reference

The 4:4:4 encoded image becomes

frame	line
Y1C1 Y2C2x Y3C3 Y4C4x	line 1
Y1C1 Y2C2x Y3C3 Y4C4x	line 2
Y1C1 Y2C2x Y3C3 Y4C4x	line 3
Y1C1 Y2C2x Y3C3 Y4C4x	line 4

### 13.2.3.2 YV12 interlaced conversion → YUY2

In AviSynth, the missing chroma samples are interpolated as follows

frame	line	weights
Y_Y_Y_Y	line 1t	chroma of YV12_lines 1t
C__C__		
Y_Y_Y_Y	line 2b	chroma of YV12_lines 4b
Y_Y_Y_Y	line 3t	chroma of YV12_lines (0.75)*1t + (0.25)*5t
C__C__		
Y_Y_Y_Y	line 4b	chroma of YV12_lines (0.75)*4b + (0.25)*8b
Y_Y_Y_Y	line 5t	chroma of YV12_lines (0.25)*1t + (0.75)*5t
C__C__		
Y_Y_Y_Y	line 6b	chroma of YV12_lines (0.25)*4b + (0.75)*8b
Y_Y_Y_Y	line 7t	chroma of YV12_lines (0.75)*5t +

### Avisynth 2.5 Selected External Plugin Reference

		$(0.25)*9t$
C__C__		
Y_Y_Y_Y	line 8b	chroma of YV12_lines $(0.75)*8b +$ $(0.25)*12b$

or

field 1	field 2	line	weights
Y_Y_Y_Y		line 1t	chroma of YV12_lines 1t
C__C__			
	Y_Y_Y_Y	line 2b	chroma of YV12_lines 4b
Y_Y_Y_Y		line 3t	chroma of YV12_lines $(0.75)*1t +$ $(0.25)*5t$
	C__C__		
	Y_Y_Y_Y	line 4b	chroma of YV12_lines $(0.75)*4b +$ $(0.25)*8b$
Y_Y_Y_Y		line 5t	chroma of YV12_lines $(0.25)*1t +$ $(0.75)*5t$
C__C__			
	Y_Y_Y_Y	line 6b	chroma of YV12_lines $(0.25)*4b +$ $(0.75)*8b$
Y_Y_Y_Y		line 7t	chroma of YV12_lines $(0.75)*5t +$ $(0.25)*9t$
	C__C__		
	Y_Y_Y_Y	line 8b	chroma of YV12_lines $(0.75)*8b +$ $(0.25)*12b$

## Avisynth 2.5 Selected External Plugin Reference

--	--	--	--

AviSynth uses a different interpolation as the one suggested by the mpeg2 specs (perhaps due to speed issues). The latter is

field 1	field 2	line	weights
Y_Y_Y_Y		line 1t	chroma of YV12_lines 1t
C__C__			
	Y_Y_Y_Y	line 2b	chroma of YV12_lines 4b
Y_Y_Y_Y		line 3t	chroma of YV12_lines (5/8)*1t + (3/8)*5t
	C__C__		
	Y_Y_Y_Y	line 4b	chroma of YV12_lines (7/8)*4b + (1/8)*8b
Y_Y_Y_Y		line 5t	chroma of YV12_lines (1/8)*1t + (7/8)*5t
C__C__			
	Y_Y_Y_Y	line 6b	chroma of YV12_lines (3/8)*4b + (5/8)*8b
Y_Y_Y_Y		line 7t	chroma of YV12_lines (5/8)*5t + (3/8)*9t
	C__C__		
	Y_Y_Y_Y	line 8b	chroma of YV12_lines (7/8)*8b + (1/8)*12b

### YV12 progressive conversion -> YUY2

The missing chroma samples are interpolated as follows

#### 13.2.3.2 YV12 interlaced conversion -> YUY2



## Avisynth 2.5 Selected External Plugin Reference

frame	line	weights
Y_Y_Y_Y	line 1	chroma of YV12_lines 1
C__C__		
Y_Y_Y_Y	line 2	chroma of YV12_lines (0.75)*1 + (0.25)*3
Y_Y_Y_Y	line 3	chroma of YV12_lines (0.25)*1 + (0.75)*3
C__C__		
Y_Y_Y_Y	line 4	chroma of YV12_lines (0.75)*3 + (0.25)*5
Y_Y_Y_Y	line 5	chroma of YV12_lines (0.25)*3 + (0.75)*5
C__C__		
Y_Y_Y_Y	line 6	chroma of YV12_lines (0.75)*5 + (0.25)*7

### 13.2.4 2.4 References

#### ColorSpaces

[\[4:4:4\]](#) sampling

[\[4:2:2\]](#) sampling

[\[4:2:0\]](#) sampling

[\[Chroma Upsampling\]](#)

[\[Chroma Subsampling Standards\]](#)

### 13.2.5 3.1 MPEG-1 versus MPEG-2 sampling

There are two common variants of 4:2:0 sampling. One of these is used in MPEG-2 (and CCIR-601) video, and the other is used in MPEG-1. **The MPEG-2 scheme is how AviSynth samples 4:2:0 video**, because it completely avoids horizontal resampling in 4:2:0 <-> 4:2:2 conversions.

The layout of a progressive MPEG-1 4:2:0 encoded image

## Avisynth 2.5 Selected External Plugin Reference

frame	line	weights
Y_Y_Y_Y	line 1	
_C__C__		chroma of YUY2_lines $(0.5)*1 + (0.5)*2$
Y_Y_Y_Y	line 2	
Y_Y_Y_Y	line 3	
_C__C__		chroma of YUY2_lines $(0.5)*3 + (0.5)*4$
Y_Y_Y_Y	line 4	

The layout of a MPEG-2 4:2:0 encoded image

frame	line	weights
Y_Y_Y_Y	line 1	
C__C__		chroma of YUY2_lines $(0.5)*1 + (0.5)*2$
Y_Y_Y_Y	line 2	
Y_Y_Y_Y	line 3	
C__C__		chroma of YUY2_lines $(0.5)*3 + (0.5)*4$
Y_Y_Y_Y	line 4	

### 13.2.6 3.2 DV sampling

For completeness, we will mention DV sampling. DV is 4:2:0 (PAL) and 4:1:1 (NTSC). Note, that the sample positioning of the former is different from the 4:2:0 chroma in MPEG-1/MPEG-2!

The layout of a 4:2:0 encoded image (field-based)

field	line
YV Y YV Y YV Y YV Y	line 1
YU Y YU Y YU Y YU Y	line 2
YV Y YV Y YV Y YV Y	line 3
YU Y YU Y YU Y YU Y	line 4

The layout of a 4:1:1 encoded image (field-based)

field	line
YC Y Y Y YC Y Y Y	line 1

## Avisynth 2.5 Selected External Plugin Reference

YC Y Y Y YC Y Y Y	line 2
YC Y Y Y YC Y Y Y	line 3
YC Y Y Y YC Y Y Y	line 4

Some comments about this formats:

- 4:1:1 is supported natively in AviSynth v2.6.
- DV decoders all output YUY2 or RGB (with the exception of ffdshow when YV12 is enabled).
- When outputting YUY2/RGB (NTSC), the MainConcept codec duplicates the chroma samples instead of interpolating. The [\[ReInterpolate411 plugin\]](#) can be used to correct for this, resulting in better quality.

### 13.2.7 3.3 References

[\[MSDN: YUV sampling\]](#) Describes the most common YUV sampling techniques.  
[\[DV sampling\]](#)

### 13.3 4. 4:2:0 Interlaced Chroma Problem (or ICP)

In general interlaced content will have static parts. If it is upsampled correctly using interlaced upsampling, it will still have *chroma problems on diagonal edges of bright-colored objects in static parts of a frame*. The reason is that "When the two fields are put back together later by a deinterlacer (or by your eye and brain, if you watch it on an interlaced TV), the relatively smooth gradations and contours of each field are broken up by a slightly different set of gradations and contours from the other field." (quote from first reference). This is called **the Interlaced Chroma Problem**. The "solution" is a motion-adaptive upsampler, but such an AviSynth/VDub filter which attempts to do this doesn't exist yet.

References:

[\[The 4:2:0 Interlaced Chroma Problem\]](#)  
[\[The 4:2:0 Interlaced Chroma Problem – Television and Video Advice\]](#)

\$Date: 2008/07/11 18:23:00 \$

### 13.4 Introduction

The available (internal) filters are listed here and divided into categories. A short description is added, including the supported color formats (and sample types for the audio filters). There are some functions which combine two or more clips in different ways. How the video content is calculated is described for each function, but [here is a summary which explains which properties that the resulting clip will have](#).

### 13.5 Media file filters

These filters can be used to read or write media files. Usually they produce source clips for processing. See debug filters for non-file source filters.

<a href="#">AVISource / OpenDMLSource / AVIFileSource</a>	Opens an AVI file.
<a href="#">DirectShowSource</a>	DirectShowSource reads filename using DirectShow
<a href="#">ImageReader / ImageSource</a>	This filter produces a video clip by reading in still images.

## Avisynth 2.5 Selected External Plugin Reference

<a href="#">Imagewriter</a>	Writes frames as images to your hard disk.
<a href="#">Import</a>	Import an AviSynth script into the current script
<a href="#">SegmentedAVISource / SegmentedDirectShowSource</a>	The SegmentedAVISource filter automatically loads up to 100 avi files per argument
<a href="#">SoundOut</a>	SoundOut is a GUI driven sound output module for AviSynth (it exports audio to several compressors).
<a href="#">WAVSource</a>	Opens a WAV file or the audio of an AVI file.

### 13.6 Color conversion and adjustment filters

These filters can be used to change the color format or adjust the colors of a clip.

<a href="#">ColorYUV</a>	Adjusts colors and luma independently.
<a href="#">ConvertBackToYUY2 / ConvertToRGB / ConvertToRGB24 / ConvertToRGB32 / ConvertToYUY2 / ConvertToY8 / ConvertToYV411 / ConvertToYV12 / ConvertToYV16 / ConvertToYV24</a>	AviSynth can deal internally with the color formats, RGB24, RGB32, YUY2, Y8, YV411, YV12, YV16 and YV24. These filters convert between them.
<a href="#">FixLuminance</a>	Correct shifting vertical luma offset
<a href="#">Greyscale</a>	Converts a video to greyscale.
<a href="#">Invert</a>	Inverts selected color channels of a video.
<a href="#">Levels</a>	The Levels filter scales and clamps the blacklevel and whitelevel and adjusts the gamma.
<a href="#">Limiter</a>	A filter for clipping levels to within CCIR-601 range.
<a href="#">MergeARGB / MergeRGB</a>	This filter makes it possible to select and combine a color channel from each of the input video clips.
<a href="#">Merge / MergeChroma / MergeLuma</a>	This filter makes it possible to merge luma, chroma or both from a video clip into another. There is an optional weighting, so a percentage between the two clips can be specified.
<a href="#">RGBAdjust</a>	Adjust each color channel separately.
<a href="#">ShowAlpha / ShowRed / ShowGreen / ShowBlue</a>	Shows the selected channel of an (A)RGB clip.
<a href="#">SwapUV / UToY / VToY / YToUV</a>	Swaps/copies chroma channels of a clip.
<a href="#">Subtract</a>	Subtract produces an output clip in which every pixel is set according to the difference between the corresponding pixels
<a href="#">Tweak</a>	Adjust the hue, saturation, brightness, and contrast.
<a href="#">UToY8 / VToY8</a>	Shorthand for UToY.ConvertToY8 / VToY.ConvertToY8.

## 13.7 Overlay and Mask filters

These filters can be used to layer clips with or without using masks and to create masks.

<a href="#">ColorKeyMask</a>	Sets the alpha-channel (similar as Mask does) but generates it by comparing the color.
<a href="#">Layer</a>	Layering two videos.
<a href="#">Mask</a>	Applies an alpha-mask to a clip.
<a href="#">MaskHS</a>	Returns a mask (as Y8) of clip using a given hue and saturation range.
<a href="#">Overlay</a>	Overlay puts two clips on top of each other with an optional displacement of the overlaying image, and using different overlay methods. Furthermore opacity can be adjusted for the overlay clip.
<a href="#">ResetMask</a>	Applies an "all-opaque" alpha-mask to clip.

## 13.8 Geometric deformation filters

These filters can be used to change image size, process borders or make other deformations of a clip.

<a href="#">AddBorders</a>	AddBorders adds black borders around the image.
<a href="#">Crop / CropBottom</a>	Crop crops excess pixels off of each frame.
<a href="#">FlipHorizontal / FlipVertical</a>	Flips the video upside-down or left-to-right
<a href="#">Letterbox</a>	Letterbox simply blackens out the top and the bottom and optionally left and right side of each frame.
<a href="#">Overlay</a>	Overlay puts two clips on top of each other with an optional displacement of the overlaying image, and using different overlay methods. Furthermore opacity can be adjusted for the overlay clip.
<a href="#">ReduceBy2 / HorizontalReduceBy2 / VerticalReduceBy2</a>	ReduceBy2 reduces the size of each frame by half.
<a href="#">BilinearResize / BicubicResize / BlackmanResize / GaussResize / LanczosResize / Lanczos4Resize / PointResize / SincResize / Spline16Resize / Spline36Resize / Spline64Resize</a>	The Resize filters rescale the input video frames to an arbitrary new resolution, using different sampling algorithms.
<a href="#">TurnLeft / TurnRight / Turn180</a>	Rotates the clip 90 degrees counterclock wise / 90 degrees clock wise / 180 degrees.

## 13.9 Pixel restoration filters

These filters can be used for image detail (pixel) restoration (like denoising, sharpening) of a clip.

<a href="#">Blur / Sharpen</a>	These are simple 3x3–kernel blurring and sharpening filters.
<a href="#">GeneralConvolution</a>	General 3x3 or 5x5 convolution matrix.
<a href="#">SpatialSoften / TemporalSoften</a>	The SpatialSoften and TemporalSoften filters remove noise from a video clip by selectively blending pixels.
<a href="#">FixBrokenChromaUpsampling</a>	I noticed that the MS DV codec upsamples the chroma channels incorrectly, and I added a FixBrokenChromaUpsampling filter to compensate for it.

## 13.10 Timeline editing filters

These filters can be used to arrange frames in time (clip cutting, splicing and other editing).

<a href="#">AlignedSplice / UnalignedSplice</a>	AlignedSplice and UnalignedSplice join two or more video clips end to end.
<a href="#">AssumeFPS / AssumeScaledFPS / ChangeFPS / ConvertFPS</a>	Changes framerates in different ways.
<a href="#">DeleteFrame</a>	DeleteFrame deletes a set of single frames, given as a number of arguments.
<a href="#">Dissolve</a>	Dissolve is like AlignedSplice, except that the clips are combined with some overlap.
<a href="#">DuplicateFrame</a>	DuplicateFrame duplicates a set of single frames, given as a number of arguments.
<a href="#">FadeIn0 / FadeOut0 / FadeIn / FadeOut / FadeIn2 / FadeOut2 / FadeIO0 / FadeIO / FadeIO2</a>	FadeIn and FadeOut cause the video stream to fade linearly to black at the start or end.
<a href="#">FreezeFrame</a>	The FreezeFrame filter replaces all the frames between first–frame and last–frame with a selected frame.
<a href="#">Interleave</a>	Interleave interleaves frames from several clips on a frame–by–frame basis.
<a href="#">Loop</a>	Loops the segment from start frame to end frame a given number of times.
<a href="#">Reverse</a>	This filter makes a clip play in reverse.
<a href="#">SelectEven / SelectOdd</a>	SelectEven makes an output video stream using only the even–numbered frames
<a href="#">SelectEvery</a>	SelectEvery is a generalization of filters like SelectEven and Pulldown.
<a href="#">SelectRangeEvery</a>	This filters selects a range of frames with a certain period.

<a href="#">Trim</a>	Trim trims a video clip so that it includes only the frames first-frame through last-frame.
----------------------	---

### 13.11 Interlace filters

These filters can be used for creating and processing field-based material (which is frame-based material separated into fields). AviSynth is capable of dealing with both progressive and interlaced material. The main problem is, that it often doesn't know what it receives from source filters. This is the reason that the field-based flag exists and can be used when dealing with interlaced material. More information about field-based video can be found [here](#).

<a href="#">AssumeFrameBased / AssumeFieldBased</a>	Forces frame-based or field-based material.
<a href="#">AssumeTFF / AssumeBFF</a>	Forces field order.
<a href="#">Bob</a>	Bob takes a clip and bob-deinterlaces it
<a href="#">ComplementParity</a>	Changes top fields to bottom fields and vice-versa.
<a href="#">DoubleWeave</a>	The DoubleWeave filter operates like Weave, except that it produces double the number of frames by combining both the odd and even pairs of fields.
<a href="#">PeculiarBlend</a>	This filter blends each frame with the following frame in a peculiar way.
<a href="#">Pulldown</a>	The Pulldown filter simply selects two out of every five frames of the source video.
<a href="#">SeparateFields</a>	SeparateFields takes a frame-based clip and splits each frame into its component top and bottom fields.
<a href="#">SwapFields</a>	The SwapFields filter swaps the two fields in an interlaced frame
<a href="#">Weave</a>	Weave takes even pairs of fields from a Fields Separated input video clip and combines them together to produce interlaced frames.

### 13.12 Audio processing filters

These filters can be used to process audio. Audio samples from a clip will be automatically converted if any filters requires a special type of sample. This means that if a filter doesn't support the type of sample it is given, it will automatically convert the samples to something it supports. The internal formats supported in each filter is listed in the sample type column. A specific sample type can be forced by using the [ConvertAudio](#) functions.

If the sample type is float, when AviSynth has to output the data, it will be converted to 16 bit, since float cannot be passed as valid AVI data.

<a href="#">Amplify / AmplifydB</a>	Amplify multiply audio samples by amount.
<a href="#">AssumeSampleRate</a>	Adjusts the playback speed of the audio.
<a href="#">AudioDub / AudioDubEx</a>	AudioDub takes the video stream from the first argument and the audio stream from the second argument and combines them. AudioDubEx is similar, but it doesn't throw an exception if both

## Avisynth 2.5 Selected External Plugin Reference

	clips don't have a video or audio stream.
<a href="#">ConvertToMono</a>	Merges all audio channels.
<a href="#">ConvertAudioTo8bit / ConvertAudioTo16bit / ConvertAudioTo24bit / ConvertAudioTo32bit / ConvertAudioToFloat</a>	Converts audio samples to 8, 16, 24, 32 bits or float.
<a href="#">DelayAudio</a>	DelayAudio delays the audio track by seconds seconds.
<a href="#">EnsureVBRMP3Sync</a>	Corrects out-of-sync mp3-AVI's, when seeking or trimming.
<a href="#">GetChannel</a>	Returns a channel from an audio signal.
<a href="#">KillAudio</a>	Removes the audio from a clip completely.
<a href="#">KillVideo</a>	Removes the video from a clip completely.
<a href="#">MergeChannels</a>	Merges channels of two or more audio clips.
<a href="#">MixAudio</a>	Mixes audio from two clips.
<a href="#">Normalize</a>	Amplifies the entire waveform as much as possible, without clipping.
<a href="#">ResampleAudio</a>	Performs a change of the audio sample rate.
<a href="#">SSRC</a>	Performs a high-quality change of the audio sample rate. It uses SSRC by Naoki Shibata, which offers the best resample quality available.
<a href="#">SuperEQ</a>	High quality 16 band sound equalizer.
<a href="#">TimeStretch</a>	This filter can change speed of the sound without changing the pitch, and change the pitch of a sound without changing the length of a sound.

### 13.13 Meta filters

These special filters can be used to control other filters execution.

<a href="#">Animate / ApplyRange</a>	Animate (ApplyRange) is a meta-filter which evaluates its parameter filter with continuously varying (the same) arguments.
<a href="#">TCPDeliver</a>	This filter will enable you to send clips over your network. You can connect several clients to the same machine.

### 13.14 Conditional filters

The basic characteristic of conditional filters is that 'their scripts' are evaluated (executed) at every frame instead of the whole clip. This allows for complex video processing that would be difficult or impossible to be performed by a normal AviSynth script.

<a href="#">ConditionalFilter / FrameEvaluate / ScriptClip</a>	ConditionalFilter returns source1 if some condition is met, otherwise it returns source2. ScriptClip/FrameEvaluate returns the clip which is returned by the function evaluated on every frame.
--	---



## Avisynth 2.5 Selected External Plugin Reference

<a href="#">ConditionalReader</a>	ConditionalReader allows you to import information from a text file, with different values for each frame – or a range of frames.
<a href="#">WriteFile / WriteFileIf / WriteFileStart / WriteFileEnd</a>	These filters evaluate expressions and output the results to a text-file.

### 13.15 Debug filters

<a href="#">BlankClip / Blackness</a>	The BlankClip filter produces a solid color, silent video clip of the specified length (in frames).
<a href="#">ColorBars</a>	The ColorBars filter produces a video clip containing SMPTE color bars scaled to any image size.
<a href="#">Compare</a>	Compares two clips and prints out information about the differences.
<a href="#">Histogram</a>	Adds a Histogram.
<a href="#">Info</a>	Prints out image and sound information.
<a href="#">MessageClip</a>	MessageClip produces a clip containing a text message
<a href="#">ShowFiveVersions</a>	ShowFiveVersions takes five video streams and combines them in a staggered arrangement from left to right.
<a href="#">ShowFrameNumber / ShowSMPTE / ShowTime</a>	ShowFrameNumber draws text on every frame indicating what number Avisynth thinks it is. ShowSMPTE displays the SMPTE timecode. <b>hh:mm:ss:ff</b> ShowTime displays the duration with millisecond resolution. <b>hh:mm:ss.sss</b>
<a href="#">StackHorizontal / StackVertical</a>	StackHorizontal takes two or more video clips and displays them together in left-to-right order.
<a href="#">Subtitle</a>	The Subtitle filter adds a single line of anti-aliased text to a range of frames.
<a href="#">Tone</a>	This will generate sound.
<a href="#">Version</a>	The Version filter generates a video clip with a short version and copyright statement

\$Date: 2009/09/12 20:57:20 \$

### 13.16 AddBorders

AddBorders (*clip, int left, int top, int right, int bottom, int "color"*)

AddBorders adds black borders around the image, with the specified widths (in pixels).

See [colorspace conversion filters](#) for constraints when using the different color formats.

The *color* parameter is optional (added in v2.07), default=0 <black>, and is specified as an RGB value regardless of whether the clip format is RGB or YUV based. Color presets can be found in the file colors\_rgb.avsi, which should be present in your plugin folder. See [here](#) for more information on specifying

colors.

Be aware that many lossy compression algorithms don't deal well with solid-color borders, unless the border width happens to be a multiple of the block size (16 pixels for MPEG).

You can use this filter in combination with [Crop](#) to shift an image around without changing the frame size. For example:

```
# Shift a 352x240 image 2 pixels to the right
Crop(0,0,350,240).AddBorders(2,0,0,0)

# Add letterbox borders that are Black making a 720x308 image 720x480
# For YUV colorspace it will be correct (for CCIR-601) color (luma=16)
AddBorders(0, 86, 0, 86, $000000)
```

**\$Date: 2008/06/06 11:37:04 \$**

## 13.17 RGBAdjust

`RGBAdjust (clip, float "r", float "g", float "b", float "a", float "rb", float "gb", float "bb", float "ab", float "rg", float "gg", float "bg", float "ag", bool "analyze")`

This filter multiplies each color channel with the given value, adds the given bias offset then adjusts the relevant gamma, clipping the result at 0 and 255. Note that `RGBAdjust(1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1)` leaves the clip untouched.

*r* (-255.0 – 255.0; default 1.0): This option determines how much red is to be scaled. For example, a scale of 3.0 multiplies the red channel of each pixel by 3. Green and blue work similar.

*a* (-255.0 – 255.0; default 1.0) specifies the scale of the alpha channel. The alpha channel represents the transparency information on a per-pixel basis. An alpha value of zero represents full transparency, and a value of 255 represents a fully opaque pixel.

In *v2.56* the bias offsets *rb*, *gb*, *bb*, *ab* (default 0.0) add a value to the red, green, blue or alpha channels. For example, *rb* = 16 will add 16 to the red pixel values and -32 will subtract 32 from all red pixel values.

Also in *v2.56* the exponents *rg*, *gg*, *bg*, *ag* (default 1.0) adjust the gamma of the red, green, blue or alpha channels. For example, *rg* = 1.2 will brighten the red pixel values and *gg* = 0.8 will darken the green pixel values.

In *v2.56* *analyze* (can be true or false) will print out color statistics on the screen. There are maximum and minimum values for all channels. There is an average and a standard deviation for all channels. There is a "loose minimum" and "loose maximum". The "loose" values are made to filter out very bright or very dark noise specs creating an artificially low or high minimum / maximum (it just means that the amount of red/green/blue of 255/256 of all pixels is above (under) the loose minimum (maximum)).

Keep in mind ALL the values are not scaled to accomodate changes to one (for that you should use levels) so doing something like:

```
RGBAdjust(2, 1, 1, 1)
```

will get you a whole lot of clipped red. If you WANT a whole lot of clipped red, there you go – but if you

want MORE red without clipping you should do

```
Levels(0, 1, 255, 0, 128).RGBAdjust(2, 1, 1, 1)
```

This would scale all the levels (and average lum) by half, then double the red. Or more compact

```
RGBAdjust(1.0, 0.5, 0.5, 1.0)
```

This leaves the red and halves the green and blue.

To invert the alpha channel

```
RGBAdjust(a=-1.0, ab=255)
```

Thus alpha pixels values become  $a=(255-a)$

### Changelog:

v2.56	added offsets, gamma, analyze
-------	-------------------------------

\$Date: 2005/05/05 06:19:11 \$

## 13.18 Amplify / AmpiflydB

Amplify (*clip, float amount1* [, ...])

AmpiflydB (*clip, float amount1* [, ...])

Amplify multiplies the audio samples by *amount*. You can specify different factors for each channel. If there are more volumes than there are channels, they are ignored. If there are fewer volumes than channels, the last volume is applied to the rest of the channels.

AmpiflydB is the same except values are in decibels (dB).

You can use negative dB values (or scale factor between 0 and 1) for reducing volume. Negative scale factors will shift the phase by 180 degrees (i.e. invert the samples).

8bit and 24bit Audio samples are converted to float in the process, the other audio formats are kept.

```
# Amplifies left channel with 3 dB (adds 3 dB):
video = AviSource("c:\filename.avi")
stereo = WavSource("c:\audio.wav")
stereo_amp = AmpiflydB(stereo, 3, 0)
return AudioDub(video, stereo_amp)
```

```
# Amplifies front channels with 3 dB (adds 3 dB):
video = AviSource("c:\divx_6ch_wav.avi")
audio = WavSource(c:\divx_6ch_wav.avi)
multichannel_amp = AmpiflydB(audio, 3, 3, 3)
return AudioDub(video, multichannel_amp)
```

How the multichannels are mapped can be found in the description of [GetChannel](#).

\$Date: 2009/09/12 15:10:22 \$

## 13.19 Animate / ApplyRange

`Animate` (*clip*, *int start\_frame*, *int end\_frame*, *string filtername*, *start\_args*, *end\_args*)

`ApplyRange` (*clip*, *int start\_frame*, *int end\_frame*, *string filtername*, *args*)

`Animate` is a meta-filter which evaluates its parameter *filter* with continuously varying arguments. At frame *start\_frame* and earlier, the *filter* is evaluated with the arguments given by *start\_args*. At frame *end\_frame* and later, the *filter* is evaluated with the arguments given by *end\_args*. In between, the arguments are linearly interpolated for a smooth transition.

`ApplyRange` is a special case of `Animate` where *start\_args* = *end\_args*, and present in v2.51. It can be used when you want to apply a certain filter only on a certain range of frames of a clip – unlike `Animate`, frames outside the range *start\_frame* to *end\_frame* are passed through untouched. Another difference with `Animate` is that *args* can't contain a clip. Starting from v2.53 it supports audio, and *start\_frame* can be equal to *end\_frame* (such that only one frame is processed).

In cases where a large number of ranges need similar processing using many `ApplyRange` calls may cause resource issues. An alternative can be to use [ConditionalReader](#) with [ConditionalFilter](#) to select between the unprocessed and processed version of a source.

The filter name must be enclosed in quotation marks (it's a string), and the two nested argument lists are not parenthesized. Strings and video clips can't be interpolated, and therefore must be identical in the two argument lists. An important warning though: If you use a clip as first argument, that same clip shouldn't be included in *start\_args* and *end\_args*. Thus for example

```
v = Version()
Animate(v,0,149,"Crop", v,0,0,64,32, v,316,0,64,32)
```

results in an error, since the first frame is the same as the invalid syntax `Crop(v, v, 0, 0, 64, 32)`.

This filter will not correctly handle a changing sound track, so I don't recommend its use with filters which modify the sound track. And for heaven's sake don't make the initial and final parameters yield a different output frame size.

The *filtername* argument can even be `Animate` if you want quadratic rather than linear interpolation.

### Several examples:

```
# Make a scrolling version of the "Version" video:
ver = Version()
Animate(0,149,"Crop", ver,0,0,64,32, ver,316,0,64,32)

# or what is the same:
ver = Version()
Animate(ver,0,149,"Crop", 0,0,64,32, 316,0,64,32)

# Fade to white
AviSource("E:\pdwork\DO-Heaven.AVI")
Animate(100,200,"Levels", 0,1,255,0,255, 0,1,255,255,255)

# Do a gradual zoom into the center of a 320x240 video, starting at
```

## Avisynth 2.5 Selected External Plugin Reference

```
# 1:1 magnification in frame 100 and ending with 4:1 magnification
# in frame 200:
clip = AviSource("E:\pdwork\DO-Heaven.avi")
Animate(100,200,"BicubicResize", clip,320,240,0,0,320,240, clip,320,240,120,90,80,60)
# Animate(clip, 100,200,"BicubicResize", 320,240,0,0,320,240, 320,240,120,90,80,60) # also work

# Make the text "Hello, World!" zoom out from the center of a 320x240 video:
BlankClip(width=320, height=240)
Animate(0,48,"Subtitle", "Hello, World!",160,120,0,99999,"Arial",0,
  \ "Hello, World!",25,130,0,99999,"Arial",48)
```

### Zooming clip c2 while overlaying it on c1:

```
Function myfunc(clip c1, clip c2, int x, int y, int w, int h)
{
  w = w - w%2
  h = h - h%2
  my_c2 = BicubicResize(c2,w,h)
  Overlay(c1,my_c2,x,y)
}

c1 = AviSource("D:\Captures\jewel.avi") # c1 is larger than c2
c2 = AviSource("D:\Captures\atomic.avi").BicubicResize(320,240)
Animate(0,1000,"myfunc",c1,c2,10,10,10,10,c1,c2,300,300,360,288)
# or
# Animate(c1,0,1000,"myfunc", c2,10,10,10,10, c2,300,300,360,288)

# but the following doesn't work, since three clips are passed to myfunc (c1, c1 and c2), while
# Animate(c1,0,1000,"myfunc",c1,c2,10,10,10,10,c1,c2,300,300,360,288)
```

### A small picture enlarges on a black clip until replace the main clip:

```
function res(clip clip, clip "LClip", int "width", int "height", int "centerX", int "centerY")
LClip = BicubicResize(LClip, width, height)
Overlay(clip, LClip, centerX-LClip.width/2, centerY-LClip.height/2)
}

function resize(clip clip, clip "LClip", int "start_frame", int "start_width", int "start_height",
  \ int "end_frame", int "end_width", int "end_height", int "centerX", int "centerY") {
  return Animate(start_frame, end_frame, "res", clip, LClip, start_width, start_height, centerX,
  \ clip, LClip, end_width, end_height, centerX, centerY)
}

clip = AviSource("D:\captures\jewel.avi")
clip = clip.BicubicResize(640,480)
clip = clip.ConvertToRGB()
black = BlankClip(clip)

resize(black, clip, 0, 120, 120*clip.height/clip.width, 500, 640, 480, clip.width/2, clip.height)
```

### Examples using ApplyRange:

```
ver = Version()
return ver.ApplyRange(0,149,"Crop", 158,0,64,32)
# gives an error since cannot have different frame sizes within a clip

Version()
ApplyRange(100,149,"Blur", 1.0)
```

## Avisynth 2.5 Selected External Plugin Reference

```
AviSource("E:\pdwork\DO-Heaven.avi").BicubicResize(320,240)
ApplyRange(0,48,"Subtitle", "Hello, World!",25,130,0,99999,"Arial",48)
```

```
# or what is the same:
```

```
clip = AviSource("E:\pdwork\DO-Heaven.avi").BicubicResize(320,240)
ApplyRange(clip, 0,48,"Subtitle", "Hello, World!",25,130,0,99999,"Arial",48)
```

```
# but since the frame range can be provided to Subtitle itself, this is the same as:
```

```
AviSource("E:\pdwork\DO-Heaven.avi").BicubicResize(320,240)
Subtitle("Hello, World!",25,130,0,48,"Arial",48)
```

```
$Date: 2009/09/12 15:10:22 $
```

### 13.20 AssumeSampleRate

AssumeSampleRate (*clip, int samplerate*)

AssumeSampleRate (exists starting from v2.07) changes the sample rate (playback speed) of the current sample.

If used alone, it will cause desync with the video, because the playback time is not changed.

```
# Let's play that this video is 25fps, 44100hz video clip.
AviSource("video_audio.avi")
```

```
# Play audio at half speed:
AssumeSampleRate(22050)
```

```
# Play video at half speed:
AssumeFPS(12.5)
```

```
# Video and audio is now in sync, and plays in slow-motion.
```

```
$Date: 2004/02/29 20:04:51 $
```

### 13.21 AudioDub / AudioDubEx

AudioDub (*video\_clip, audio\_clip*)

AudioDubEx (*video\_clip, audio\_clip*)

AudioDub takes the video stream from the first argument and the audio stream from the second argument and combines them into a single clip. If either track isn't available, it tries it the other way around, and if that doesn't work it returns an error.

```
# Load capture segments from patched AVICAP32 which puts
# video in multiple AVI segments and audio in a WAV file
video = AVISource("capture1.avi") + AVISource("capture2.avi")
audio = WAVSource("capture.wav")
AudioDub(video, audio)
```

AudioDubEx takes the video stream from the first argument if present, the audio stream from the second argument if present and combines them into a single clip. Thus if you feed it with two video clips, and the second one has no audio, the resulting clip will have the video of the first clip and no audio. If you feed it with

two audio clips, the resulting clip will have the audio of the second clip and no video.

\$Date: 2005/11/08 12:37:33 \$

## 13.22 AVISource / OpenDMLSource / AVIFileSource / WAVSource

`AVISource (string filename [, ...], bool "audio" = true, string "pixel_type" = YV12, [string fourCC])`

`OpenDMLSource (string filename [, ...], bool "audio" = true, string "pixel_type" = YV12, [string fourCC])`

`AVIFileSource (string filename [, ...], bool "audio" = true, string "pixel_type" = YV12, [string fourCC])`

`WAVSource (string filename [, ...])`

`AVISource` takes as argument one or more file name in quotes, and reads in the file(s) using either the Video-for-Windows "AVIFile" interface, or AviSynth's built-in OpenDML code (taken from VirtualDub). This filter can read any file for which there's an AVIFile handler. This includes not only AVI files but also WAV files, AVS (AviSynth script) files, and VDR (VirtualDub frameserver) files. If you give multiples filenames as arguments, the clips will be spliced together with [UnalignedSplice](#). The *bool* argument is optional and defaults to `true`.

The `AVISource` filter examines the file to determine its type and passes it to either the AVIFile handler or the OpenDML handler as appropriate. In case you have trouble with one or the other handler, you can also use the `OpenDMLSource` and `AVIFileSource` filters, which force the use of one or the other handler. Either handler can read ordinary (< 2GB) AVI files, but only the OpenDML handler can read larger AVI files, and only the AVIFile handler can read other file types like WAV, VDR and AVS. There is built-in support for ACM (Audio Compression Manager) audio (e.g. mp3-AVIs).

`WAVSource` can be used to open a WAV file, or the audio stream from an AVI file. This can be used, for example, if your video stream is damaged or its compression method is not supported on your system.

The *pixel\_type* parameter (default "YV12" allows you to choose the output format of the decompressor. Valid values are "YV12", "YV411", "YV16", "YV24", "YUY2", "Y8", "RGB32" and "RGB24". If omitted, AviSynth will use the first format supported by the decompressor (in the following order: YV12, YV411, YV16, YV24, YUY2, Y8, RGB32 and RGB24). This parameter has no effect if the video is in an uncompressed format, because no decompressor will be used in that case. To put it in different words: if you don't specify something it will try to output the AVI as YV12, if that isn't possible it tries YV411 and if that isn't possible it tries YV16, etc ...

Sometimes the colors will be distorted when loading a DivX clip in AviSynth v2.5 (the chroma channels U and V are swapped), due to a bug in DivX (5.02 and older). You can use [SwapUV](#) to correct it.

From v2.53 `AVISource` can also open DV type 1 video input (only video, not audio).

From v2.55, an option *fourCC* is added. FourCC, is a FOUR Character Code in the beginning of media file, mostly associated with avi, that tells what codec your system should use for decoding the file. You can use this to force AviSource to open the avi file using a different codec. A list of FOURCCs can be found [here](#). By default, the fourCC of the avi is used.

## Avisynth 2.5 Selected External Plugin Reference

Some MJPEG/DV codecs do not give correct CCIR 601 compliant output when using AVISource. The problem could arise if the input and output colorformat of the codec are different. For example if the input colorformat is YUY2, while the output colorformat is RGB, or vice versa. There are two ways to resolve it:

1) Force the same output as the input colorformat. Thus for example (if the input is RGB):

```
AVISource("file.avi", pixel_type="RGB32")
```

2) Correct it with the filter [ColorYUV](#):

```
AVISource("file.avi").ColorYUV(levels="PC->TV")
```

Some reference threads:

[MJPEG codecs](#)

[DV codecs](#)

### Examples:

```
# C programmers note: backslashes are not doubled; forward slashes work too
AVISource("d:\capture.avi")
AVISource("c:/capture/00.avi")
WAVSource("f:\soundtrack.wav")
WAVSource("f:/soundtrack.wav")

# the following is the same as AVISource("cap1.avi") + AVISource("cap2.avi"):
AVISource("cap1.avi", "cap2.avi")

# disables audio and request RGB32 decompression
AVISource("cap.avi", false, "RGB32")

# opens a DV using the Canopus DV Codec
Avisource("cap.avi", false, fourCC="CDVC")

# opens an avi (for example DivX3) using the XviD Codec
Avisource("cap.avi", false, fourCC="XVID")

# splicing two clips where one of them contains no audio.
# when splicing the clips must be compatible (have the same video and audio properties):
A = Avisource("FileA.avi")
B = Avisource("FileB.avi") # No audio stream
A ++ AudioDub(B, BlankClip(A))
```

Some compression formats impose a limit to the number of Avisource() calls that can be placed in a script. Some people have experienced this limit with fewer than 50 Avisource() statements. See [discussion](#).

### Changes:

v2.55	Added fourCC option.
-------	----------------------

\$Date: 2009/09/12 15:10:22 \$

## 13.23 BlankClip / Blackness

BlankClip(*clip clip*, *int "length"*, *int "width"*, *int "height"*, *string "pixel\_type"*,



## Avisynth 2.5 Selected External Plugin Reference

```
float "fps", int "fps_denominator", int "audio_rate", int "channels",  
string "sample_type", int "color", int "color_yuv")
```

```
BlankClip (clip clip, int "length", int "width", int "height", string "pixel_type",  
float "fps", int "fps_denominator", int "audio_rate", bool "stereo",  
bool "sixteen_bit", int "color", int "color_yuv")
```

Blackness ()

The `BlankClip` filter produces a solid color, silent video clip of the specified *length* (in frames). The clip passed as an argument is used as a template for frame rate, image size, and so on, but you can specify all clip properties without having to provide a template. *color* should be given as hexadecimal RGB values. Without any argument, `BlankClip` will produce a pitch-black 10 seconds clip (RGB32), 640x480, 24 fps, 16 bit 44100 Hz mono.

*clip*: if present, the resulting clip will have the clip-properties of the template, except for the properties you define explicitly.

*length*: length of the resulting clip (in frames).

*width, height*: width and height of the resulting clip.

*pixel\_type*: pixel type of the resulting clip, it can be "RGB24", "RGB32", "YUY2" or "YV12".

*fps*: the framerate of the resulting clip.

*fps\_denominator*: you can use this option if "fps" is not accurate enough. For example: fps = 30000, fps\_denominator = 1001 (ratio = 29.97) or fps = 24000, fps\_denominator = 1001 (ratio = 23.976). It is 1 by default.

*audio\_rate*: samplerate of the silent audio of the clip.

*channels*: specifies the number of audio channels of silent audio added to the blank clip ( added in v2.58).

*stereo*: (boolean) when set to true the silent audio is in stereo, when set to false a silent mono track is added. Deprecated! Use should the *channels* parameter instead.

*sample\_type*: specifies the audio sample type of the resulting clip. It can be "8bit", "16bit", "24bit", "32bit" or "float" ( added in v2.58).

*sixteen\_bit*: (boolean) true give 16 bit, false gives ieee float. Deprecated! Use the *sample\_type* parameter instead.

*color*: specifies the color of the clip, black (= \$000000) by default. See `ColorPresets` for preset colors. See [here](#) for more information on specifying colors.

*color\_yuv*: is added in v2.55, and it lets you specify the color of the clip using YUV values. It requires setting *pixel\_type* to "YUY2" or "YV12", otherwise it doesn't do anything.

`Blackness` is an alias for `BlankClip`, provided for backward compatibility.

**Examples:**

```
# produces a black clip (3000 frames, width 720, height 576, framerate 25), with a silent audio
BlankClip(length=3000, width=720, height=576, fps=25, color=$000000)

# produces a black clip (3000 frames) with the remaining clip properties of the avi:
AviSource("E:\pdwork\DO-Heaven.AVI")
BlankClip(length=3000, color=$000000)

# adds a silent audio stream (with a samplerate of 48 kHz) to a video clip:
video = AviSource("E:\pdwork\DO-Heaven.AVI")
audio = BlankClip(video, audio_rate=48000)
AudioDub(video, audio)
```

**Changes:**

v2.55	added color_yuv
v2.58	added channels and sample_type

\$Date: 2009/09/12 15:10:22 \$

## 13.24 Blur / Sharpen

```
Blur (clip, float amount)
Blur (clip, float amountH, float amountV, bool MMX)
Sharpen (clip, float amount)
Sharpen (clip, float amountH, float amountV, bool MMX)
```

This is a simple 3x3–kernel blurring filter. The largest allowable argument for `Blur` is about 1.58, which corresponds to a (1/3,1/3,1/3) kernel. A value of 1.0 gets you a (1/4,1/2,1/4) kernel. If you want a large–radius Gaussian blur, I recommend chaining several copies of `Blur(1.0)` together. (Anybody remember Pascal's triangle?)

Negative arguments to `Blur` actually sharpen the image, and in fact `Sharpen(n)` is just an alias for `Blur(-n)`. The smallest allowable argument to `Blur` is `-1.0` and the largest to `Sharpen` is `1.0`.

You can use 2 arguments to set independent Vertical and Horizontal amounts. Like this, you can use `Blur(0,1)` to filter only Vertically, for example to blend interlaced lines together. By default `amountV=amountH`.

A Known issue, with the MMX routines is the lack of full 8 bit precision in the calculations. This can lead to banding in the resultant image. Set the `MMX=False` option to use the slower but more accurate C++ routines if this is a concern.

\$Date: 2006/12/03 11:37:04 \$

## 13.25 Bob

```
Bob (clip, float "b", float "c", float "height")
```

Bob takes a clip and bob–deinterlaces it. This means that it enlarges each field into its own frame by

## Avisynth 2.5 Selected External Plugin Reference

interpolating between the lines. The top fields are nudged up a little bit compared with the bottom fields, so the picture will not actually appear to bob up and down. However, it will appear to "shimmer" in stationary scenes because the interpolation doesn't really reconstruct the other field very accurately.

This filter uses [BicubicResize](#) to do its dirty work. You can tweak the values of *b* and *c*. You can also take the opportunity to change the vertical resolution with the *height* parameter.

A bob filter doesn't really move the physical position of a field. It just puts it back where it started. If you just [SeparateFields\(\)](#) then you have 2 half height frames: line 0 becomes line 0 of frame 0 and line 1 becomes line 0 of frame 1. Thus line 0 and 1 are now in the same place! Bob now basically resizes each frame by a factor of two but in the first frame uses the original lines for the even lines and in the second frame uses the original lines for the odd lines, exactly as is supposed to be. If you just did a resize vertically by a factor of 2 on each frame after doing a [SeparateFields\(\)](#), then it wouldn't work right because the physical position of a field moves.

Schematic:

Suppose the lines 0o, 1o, 2o, 3o, ... are original lines and 0i, 1i, 2i, 3i, ... are the interpolated lines.

start with

line number	frame 0
0)	0o
1)	1o
2)	2o
3)	3o

separate fields

line number	frame 0	frame 1
0)	0o	1o
1)	2o	3o

double size

line number	frame 0	frame 1
0)	0o	1o
1)	1i	2i
2)	2o	3o
3)	3i	4i

but this is wrong, because the physical position of the field changed.

Bob does it right

line number	frame 0	frame 1
0)	0o	0i
1)	1i	1o
2)	2o	2i

3)	3i	3o
----	----	----

### 13.25.0.1 To strictly preserve the original fields and just fill in the missing lines.

```
bob(0.0, 1.0)
```

Bob(0.0, 1.0) preserves the original fields for RGB and YUY2 and preserves the Luma but not the Chroma for YV12.

The filter coefficients with b=0.0 and c=1.0 give you 0 at x=1.0/2.0 and 1 at x=0. Which with the +/-0.25 shift occurring on the original field locations, you get a very crisp cubic filter with  $-1/8 \ 5/8 \ 5/8 \ -1/8$  coefficients on the x=0.5/1.5 taps for the other field.

However, since the shift on the chroma planes is only 0.125 for YV12 the taps don't end up on exactly the same distances. More [discussion](#).

\$Date: 2007/03/10 22:35:42 \$

## 13.26 ColorBars

ColorBars (*int "width", int "height", string "pixel\_type"*)

The ColorBars filter produces a video clip containing SMPTE color bars scaled to any image size. The clip produced is 640x480, RGB32 [16,235], 29.97 fps, 1 hour long.

A test tone is also generated. Test tone is a 440Hz sine at 48KHz, 16 bit, stereo. The tone pulses in the RIGHT speaker, being turned on and off once every second.

In v2.56, *pixel\_type* = "YUY2" or "YV12" is added (default "RGB32"). Note, this is almost equivalent to

```
ColorBars().ConvertToYUY2(matrix="PC.601") # don't scale the luma range
```

When directly generating YUV format data the color transitions are arranged to occur on a chroma aligned boundary.

The lower part of ColorBars is called the pluge. From left to right it consists of: -I, white, +Q, black, -4/0/+4 IRE levels and black. The -4/0/4 IRE levels can be used to set the brightness correctly. The -4 IRE and 0 IRE should have the same brightness, and the +4 IRE should be a little brighter than -4/0 IRE. The -I/+Q levels are not really interesting, since they are not used anymore for NTSC (analog TV), but they were used to set the chrominance correctly. More information about the colorbars and the pluge can be found [here](#).

### Changelog:

v2.56	added pixel_type="YUY2", "YV12"
-------	---------------------------------

\$Date: 2008/03/25 21:50:24 \$

## 13.27 ColorYUV

`ColorYUV (clip, float "gain_y", float "off_y", float "gamma_y", float "cont_y", float "gain_u", float "off_u", float "gamma_u", float "cont_u", float "gain_v", float "off_v", float "gamma_v", float "cont_v", string "levels", string "opt", boolean "showyuv", boolean "analyze", boolean "autowhite", boolean "autogain")`

`ColorYUV` allows many different methods of changing the color and luminance of your images. `ColorYUV` is present in `AviSynth v2.5`. All settings for this filter are optional. All values are defaulting to "0".

*gain*, *off*, *gamma* and *cont* can be set independent on each channel.

*gain* is a multiplier for the value, and it stretches the signal up from the bottom. In order to confuse you, in the filter [Tweak](#) this setting is called *contrast*. That means that if the gain is set to 0, it preserves the values as they are. When gain is 256 all values are multiplied by 2 (twice as bright). If the gain is 512 all values are multiplied by 3. Thus if  $gain = k * 256$  for some integer  $k$  then  $Y$  becomes  $(k+1) * Y$  (idem for the chroma). Although it is possible, it doesn't make sense to apply this setting to the chroma of the signal.

*off* (offset) adds a value to the luma or chroma values. An offset set to 16 will add 16 to the pixel values. An offset of -32 will subtract 32 from all pixel values.

*gamma* adjusts gamma of the specified channel. A gamma value of 0 is the same as gamma 1.0. When gamma is set to 256 it is the same as gamma 2.0. Gamma is valid down to -256, where it is the same as gamma 0.0. Note: gamma for chroma is not implemented (*gamma\_u* and *gamma\_v* are dummy parameters).

*cont* (contrast) is also multiplier for the value, and it stretches the signal out from the center. That means that if the contrast is set to 0, it preserves the values as they are. When the contrast is 256 all values are multiplied by 2 (twice as bright). If the contrast is 512 all values are multiplied by 3. Thus if  $cont = k * 256$  for some integer  $k$  (and zero gain) then  $Y$  becomes  $Y + k * (Y - 128)$  (idem for the chroma). Although it is possible, it doesn't make sense to apply this setting to the luma of the signal.

*levels* can be set to either "TV->PC" or "PC->TV". This will perform a range conversion. Normally YUV values are not mapped from 0 to 255 (PC range), but a limited range (TV range). This performs conversion between the two formats. If nothing is entered as parameter, no conversion will be made (default operation).

*opt* can be either "coring" or "" (nothing, default setting). Specifying "coring" will clip your YUV values to the valid TV-ranges. Otherwise "invalid results" will be accepted.

*showYUV* can be true or false. This will overwrite your image with an image showing all chroma values along the two axes. This can be useful if you need to adjust the color of your image, but need to know how the colors are arranged. At the topleft of the image, the chroma values are '16'. At the right side of the image, U is at maximum. At the bottom of the screen V is at its maximum. In the middle both chroma is 128 (or grey).

*analyze* can be true or false. This will print out color statistics on the screen. There are maximum and minimum values for all channels. There is an average for all channels. There is a "loose maximum" and "loose minimum". The "loose" values are made to filter out very bright or very dark noise creating an artificially low or high minimum / maximum.

*autowhite* can be true or false. This setting will use the information from the analyzer, and attempt to center the color offsets. If you have recorded some material, where the colors are shifted toward one color, this filter

## Avisynth 2.5 Selected External Plugin Reference

may help. But be careful – it isn't very intelligent – if your material is a clear blue sky, autowhite will make it completely grey! If you add "off\_u" or "off\_v" parameters at the same time as autowhite, they will not be used!

*autogain* can be true or false. This setting will use the information from the analyzer, and attempt to create as good contrast as possible. That means, it will scale up the luma (y) values to match the minimum and maximum values. This will make it act much as an "autogain" setting on cameras, amplifying dark scenes very much, while leaving scenes with good contrast alone. Some places this is also referred to as "autolevels".

The quantities saturation, contrast and brightness (as in [Tweak](#) for example) are connected with quantities in this filter by the following equations:

```
cont_u = cont_v = (sat-1) * 256
gain_y = (cont-1) * 256
off_y = bright
```

A saturation of 0.8 gives for example:  $\text{cont}_u = \text{cont}_v = -0.2 * 256 = -51.2$ . Note that in Tweak your YUV values will always be clipped to valid TV-ranges, but here you have to specify `opt="coring"`.

```
# This will adjust gamma for luma, while making luma smaller and chroma U greater:
ColorYUV(gamma_y=128, off_y=-16, off_u=5)
```

```
# Shows all colors. Frame 0 luma is 16, frame 1 luma is 17 and so on.
ColorYUV(showyuv=true)
```

```
# Recovers visibility on very bad recordings.
ColorYUV(autogain=true, autowhite=true)
```

**\$Date: 2009/09/12 15:10:22 \$**

### 13.28 Compare

`Compare (clip_filtered, clip_original, string "channels", string "logfile", bool "show_graph")`

This filter compares the original clip *clip\_original* and its filtered version *clip\_filtered*. The filtered version will be returned with the results of the comparison. Per frame the Mean Absolute Difference, Mean Difference and Peak signal-to-noise ratio (PSNR) is given, as well as the min (minimum), avg (average) and max (maximum) PSNR up to that frame (calculated frame-wise). Starting from v2.53, the 'Overall PSNR' (calculated over all pixels in all frames) is also shown on the output clip.

The *channels* (default "") string is a combination of R,G,B [,A] or Y,U,V, depending on the source clips. If it is empty, it means either "YUV" when the input clips are YCbCr or "RGB" when in the input clips are RGB.

If *show\_graph* (default false) is true then Marc's PSNR graph is also drawn on it.

If a logfile is specified, the results will be written to a file by this name and not drawn on the clip. It is much faster if you need to compare a lot of frames.

#### Examples:

```
# Displays differences on screen
Compare(clip1, clip2)
```

## Avisynth 2.5 Selected External Plugin Reference

```
# for creating a log file:
Compare(clip1, clip2, "", "compare.log")
# will only compare chroma channels of YUY2 clips.
Compare(clip1, clip2, "UV")
```

The [PSNR](#) is measured in decibels. It's defined as

$$\text{PSNR}(I,K) = 20 * \log_{10} ( 255/\sqrt{\text{MSE}(I,K)} )$$

with

$$\text{MSE}(I,K) = 1/M * \sum_{\{j,k\}} | I(j,k) - K(j,k) |^2$$

and (j,k) runs over all pixels in a frame, and M is the number of pixels in a frame.

### Changes:

v2.58	YV12 support.
-------	---------------

\$Date: 2008/06/16 19:42:53 \$

## 13.29 ConditionalFilter

`ConditionalFilter` (*clip testclip, clip source1, clip source2, string expression1, string operator, string expression2, bool "show"*)

`ConditionalFilter` returns *source1* when the condition formed by "expression1+operator+expression2" is met for current frame, otherwise it returns *source2*. If any function in *expression1* or *expression2* is not explicitly applied to a clip, it will be applied on *testclip*. The audio is taken from *source1*.

An example. This will choose frames from `vid_blur` when the average luma value of a frame is less than 20. Otherwise frames from `vid` will be returned.

```
vid = AviSource("file")
vid_blur = vid.Blur(1.5)
ConditionalFilter(vid, vid_blur, vid, "AverageLuma()", "lessthan", "20")
```

Adding `show="true"` will display the actual values on the screen.

The strings *expression1* and *expression2* can be any numeric or boolean expressions, and may include internal or user functions, as well as some additional functions which are predefined ([the Runtime Functions](#)) and the special runtime variable *current\_frame* (the framenummer of the requested frame). The string *operator* can be "equals", "greaterthan" or "lessthan". Or "=", ">" or "<" respectively.

## 13.30 ScriptClip

`ScriptClip` (*clip, string filter, bool "show", bool "after\_frame"*)

`ScriptClip` returns the clip returned by the *filter* evaluated on every frame. The string *filter* can be any expression returning a clip, including internal or user clip functions, and may include line breaks (allowing a

## Avisynth 2.5 Selected External Plugin Reference

sequence of statements to be evaluated). Also, also some functions which are predefined ([the Runtime Functions](#)) and the special runtime variable *current\_frame* (the framenummer of the requested frame) can be used in the *filter* expression. Adding *show="true"* will display the actual values on the screen.

Some examples:

```
# This will print the difference from the previous frame onto the current one:
clip = AviSource("c:\file.avi")
ScriptClip(clip, "Subtitle(String(YDifferenceFromPrevious))")

# This will apply blur on each frame based on the difference from the previous.
# This will also show how errors are reported on some frames :)
clip = AviSource("c:\file.avi")
ScriptClip(clip, "Blur(YDifferenceFromPrevious/20.0)")

# This will apply temporalsoften to very static scenes, and apply a _variable_ blur on moving s
# Blur is now capped properly. We also assign a variable - and this is why a line break is inse
function fmin(float f1, float f2) {
    return (f1<f2) ? f1 : f2
}
clip = AviSource("c:\file.avi")
T = clip.TemporalSoftten(2, 7, 7, 3, 2)
ScriptClip(clip, "diff = YDifferenceToNext()+"chr(13)+"diff>2.5 ? Blur(fmin(diff/20, 1.5)) : T")

# Shows the frame-number in a clip:
ScriptClip("subtitle(string(current_frame))")

# Shows 'frame = the frame-number' in a clip:
ScriptClip(" " "subtitle("frame = " + string(current_frame))" " " " )
```

In v2.55 an *after\_frame=true/false* option to is added. This determines if the script should be evaluated before (default operation) or after the frame has been fetched from the filters above.

"Restrictions": the output of the script MUST be exactly like the clip delivered to `ScriptClip` (same colorspace, width and height). Your returned clip is allowed to have different length – but the length from "clip" is always used. Audio from "clip" is passed through untouched. For two very different sources (MPEG2DEC3 and AviSource) – you might run into colorspace mismatches. This is known quirk.

### 13.31 FrameEvaluate

`FrameEvaluate (clip clip, script filter, bool "after_frame")`

Similar to `ScriptClip`, except the output of the *filter* is ignored. This can be used for assigning variables, etc. Frames are passed directly through from the supplied clip.

In v2.53 an *after\_frame=true/false* option to is added. This determines if the script should be evaluated before (default operation) or after the frame has been fetched from the filters above.

### 13.32 ConditionalReader

This filter allows you to import arbitrary information into a selectable variable.

See the dedicated [ConditionalReader](#) page.



## 13.33 Runtime Functions

These are the internal functions which are evaluated every frame.

These will return the average pixel value of a plane (require YV12, ISSE):

AverageLuma (*clip*)

AverageChromaU (*clip*)

AverageChromaV (*clip*)

These return a float value between 0 and 255 of the absolute difference between two planes (require YV12, ISSE):

RGBDifference (*clip1*, *clip2*)

LumaDifference (*clip1*, *clip2*)

ChromaUDifference (*clip1*, *clip2*)

ChromaVDifference (*clip1*, *clip2*)

When using these functions there is an "implicit last" clip (first parameter doesn't have to be specified), so the first parameter is replaced by the testclip.

These should be quite handy for detecting scene change transitions:

RGBDifferenceFromPrevious (*clip*)

YDifferenceFromPrevious (*clip*)

UDifferenceFromPrevious (*clip*)

VDifferenceFromPrevious (*clip*)

RGBDifferenceToNext (*clip*)

YDifferenceToNext (*clip*)

UDifferenceToNext (*clip*)

VDifferenceToNext (*clip*)

```
# This will replace the last frame before a scenechange
```

```
# with the first frame after the scenechange:
```

```
ConditionalFilter(last, last, last.trim(1,0), "YDifferenceToNext()", ">", "10", true)
```

### 13.33.0.1 Other internal functions:

YPlaneMax (*clip*, *float threshold*)

UPlaneMax (*clip*, *float threshold*)

VPlaneMax (*clip*, *float threshold*)

YPlaneMin (*clip*, *float threshold*)

UPlaneMin (*clip*, *float threshold*)

VPlaneMin (*clip*, *float threshold*)

YPlaneMedian (*clip*)

UPlaneMedian (*clip*)

VPlaneMedian (*clip*)

YPlaneMinMaxDifference (*clip*, *float threshold*)

UPlaneMinMaxDifference (*clip*, *float threshold*)

VPlaneMinMaxDifference (*clip*, *float threshold*)

*Threshold* is a percentage, on how many percent of the pixels are allowed above or below minimum. The threshold is optional and defaults to 0.

If you understand the stuff above, you can proceed with "advanced conditional filtering", which tells you a little bit more about conditional filtering.

### 13.34 Advanced conditional filtering: part I

You will have to know a few things about the functionality of Avisynth to understand this section: Scripts are parsed from top to bottom, but when a frame is requested the last filter is actually being invoked first, requesting frames upwards in the filter chain. For example:

```
AviSource("myfile.avi")
ColorYUV(analyze=true)
Histogram()
```

When opening the script in Vdub the following happens

- When Vdub requests a frame, Avisynth requests the frame from Histogram.
- Histogram requests a frame from ColorYUV,
- ColorYUV requests a frame from AviSource, which produces the frame, and delivers it to ColorYUV.
- ColorYUV processes the image and sends it on to Histogram, which returns it to Vdub.

So the filter chain basically works backwards (the output is 'pulled' from below rather than 'pushed' from above), which gives each filter the possibility to request several frames from the source above. Conditional filters however, need to evaluate scripts before they request frames from the filter above, because they need to know which filter to call. Another important issue is that run-time scripts are evaluated in the same context as the main script. Hence only global defined variables in the conditional filter 'environment' can be used inside a function (and vice versa). Have a look at the following script:

```
v = AviSource("E:\Temp\Test3\atomic_kitten.avi").ConvertToYV12

function g(clip c)
{
  global w = c
  c2 = ScriptClip(c, "subtitle(t)")
  c3 = FrameEvaluate(c2, "t = String(text)")
  c4 = FrameEvaluate(c3, "text = YDifferenceFromPrevious(w)")
  return c4
}

g(v)
```

This filter chain works like this:

- When Vdub requests a frame, Avisynth requests a frame from the second FrameEvaluate, the last filter in the chain generated by g().
- The second FrameEvaluate evaluates YDifferenceFromPrevious(w), which leads to the following actions:
  - ◆ YDifferenceFromPrevious requests a frame from ConvertToYV12;
  - ◆ ConvertToYV12 requests a frame from AviSource, which produces the frame, and delivers it to ConvertToYV12;
  - ◆ ConvertToYV12 processes the image and returns it to YDifferenceFromPrevious;
  - ◆ YDifferenceFromPrevious requests a second frame from ConvertToYV12, which is obtained in a similar way to the first;
  - ◆ It then compares the two frames to calculate its result which it delivers to FrameEvaluate.

## Avisynth 2.5 Selected External Plugin Reference

- FrameEvaluate assigns this value to the variable `text`.
- After this a frame is requested from the first FrameEvaluate.
- The first FrameEvaluate, after evaluating `String(text)` and assigning this value to the variable `t`, requests a frame from ScriptClip.
- ScriptClip sets `last` to the result of `ConvertToYV12()`, evaluates `Subtitle(t)` (creating a new, temporary, filter chain), and requests a frame from it.
  - ◆ Subtitle requests a frame from `ConvertToYV12`;
  - ◆ `ConvertToYV12` requests a frame from `AviSource`, which produces the frame, and delivers it to `ConvertToYV12`;
  - ◆ `ConvertToYV12` processes the image and returns it to `Subtitle`;
  - ◆ `Subtitle` adds the specified text to the frame and delivers the result to `ScriptClip`.
- ScriptClip returns the subtitled frame to the first FrameEvaluate.
- In turn this frame is returned to the second FrameEvaluate, and hence to Avisynth which returns it to VDub.

Notice how the addition of run-time filters and run-time functions makes the interactions between different parts of the filter chain more complex. This added complexity is managed internally by Avisynth, so you needn't worry about it. However, care is required when setting and using variables, as the order of events can be less obvious to the script writer (you!).

As can be seen, `w` is defined as a global variable. This way we can use it later in the script in the conditional environment. If we want to use the variables `t` and `text` in a different function (inside or outside the conditional environment), they must also be defined as global variables. Thus for example:

```
v = AviSource("E:\Temp\Test3\atomic_kitten.avi").ConvertToYV12

function g(clip c)
{
    global w = c
    c2 = ScriptClip(c, "subtitle(t)")
    c3 = FrameEvaluate(c2, "me()")
    c4 = FrameEvaluate(c3, "global text = YDifferenceFromPrevious(w)")
    return c4
}

function me()
{
    global t = String(text)
}

g(v)
```

This is just an illustration to demonstrate the various features. Much of the script above is redundant, and can be removed. The following two scripts give the same output

```
v = AviSource("c:\clip.avi")
# ScriptClip accepts multi-line scripts:
Scriptclip(v, "
    text = YDifferenceFromPrevious()
    t = string(text)
    subtitle(t)
")

v = AviSource("c:\clip.avi")
ScriptClip(v, "Subtitle(String(YDifferenceFromPrevious))")
```

In the following section some frame dependent info will be written to a text-file.

## 13.35 Advanced conditional filtering: part II

In the following example, some frame dependent info will be written to a text-file. The first variable "a" indicates whether the frame is combed (for a certain threshold). Note that IsCombed is a filter from the Decomb plugin. The second variable "b" indicates whether there is "much" movement in the frame.

```
global sep="."
global combedthreshold=25

function IsMoving()
{
global b = (diff < 1.0) ? false : true
}

function CombingInfo(clip c)
{
file = "F:\interlace.log"
global clip = c
c = WriteFile(c, file, "a", "sep", "b")
c = FrameEvaluate(c, "global a = IsCombed(clip, combedthreshold)")
c = FrameEvaluate(c, "IsMoving")
c = FrameEvaluate(c, "global diff = 0.50*YDifferenceFromPrevious(clip) + 0.25*UDifferenceFromPrevious(clip)")
return c
}

v = mpeg2source("F:\From_hell\from_hell.d2v").trim(100,124)
CombingInfo(v)
```

We can tidy up the two functions, and remove global variables, by writing them as follows:

```
function IsMoving(float diff)
{
return (diff >= 1.0)
}

function CombingInfo(clip c)
{
file = "F:\interlace.log"

c = WriteFile(c, file, "a", "sep", "b")
c = FrameEvaluate(c, "
diff = 0.50*YDifferenceFromPrevious() + 0.25*UDifferenceFromPrevious() + 0.25*VDifferenceFromPrevious()
b = IsMoving(diff)
a = IsCombed(combedthreshold)
")

return c
}
```

In the following section an example of "adaptive motion/resizing filter" will be considered.

## 13.36 Advanced conditional filtering: part III

Some adaptive motion/resizing filters appeared on the forums. These filters discriminate between low,

## Avisynth 2.5 Selected External Plugin Reference

medium and high motion in a clip (on frame basis). By doing that, different filters can be used for different kind of motion in the clip. In general, one should use temporal smoothing in low motion scenes, spatial smoothing in high motion scenes and use spatio-temporal smoothing in medium motion scenes.

Below, a simplified version of QUANTIFIED MOTION FILTER v1.5 b1 (10/07/2003) by HomiE FR, is given:

```
-----
# QUANTIFIED MOTION FILTER v1.3
# LOADING AVISYNTH PLUGINS
LoadPlugin("C:\PROGRA~1\GORDIA~1\mpeg2dec3.dll")
LoadPlugin("C:\PROGRA~1\GORDIA~1\TemporalCleaner.dll")
LoadPlugin("C:\PROGRA~1\GORDIA~1\FluxSmooth.dll")
LoadPlugin("C:\PROGRA~1\GORDIA~1\UnFilter.dll")

# LOADING QUANTIFIED MOTION FILTER SCRIPT

Import("E:\temp\QMF\qmf.avs")

# LOW MOTION FILTER FUNCTION
# -> SHARP RESIZING + TEMPORAL ONLY
function Low_Motion_Filter(clip c)
{
    c = TemporalCleaner(c, 5, 10)
    c = LanczosResize(c, 512, 272)
    return c
}

# MEDIUM MOTION FILTER FUNCTION
# -> NEUTRAL BICUBIC RESIZING + TEMPORAL & SPATIAL
function Medium_Motion_Filter(clip c)
{
    c = FluxSmooth(c, 7, 7)
    c = BicubicResize(c, 512, 272, 0.00, 0.50)
    return c
}

# HIGH MOTION FILTER FUNCTION
# -> SOFT RESIZING + SPATIAL ONLY
function High_Motion_Filter(clip c)
{
    c = FluxSmooth(c, -1, 14)
    c = UnFilter(c, -30, -30)
    c = BilinearResize(c, 512, 272)
    return c
}

# OPENING VIDEO SOURCE
AviSource("E:\temp\QMF\britney-I_love_rock_'n_roll.avi")
ConvertToYV12(interlaced=true)
Telecide(0)

# APPLYING ADAPTATIVE RESIZING FILTER (USING QMF)
QMF()
-----

# QUANTIFIED MOTION FILTER (17/08/2003) by HomiE FR (homie.fr@wanadoo.fr)
# MOTION ESTIMATION FUNCTION
function ME()
{
    # SETTING MOTION LEVEL ACCORDING TO AVERAGE DIFFERENCE [1]
```

## Avisynth 2.5 Selected External Plugin Reference

```
global motion_level = (diff < threshold_lm) ? 0 : motion_level
global motion_level = (diff >= threshold_lm && diff <= threshold_hm) ? 1 : motion_level
global motion_level = (diff > threshold_hm) ? 2 : motion_level
}

# QUANTIFIED MOTION FILTER FUNCTION
function QMF(clip c, float "threshold_lm", float "threshold_hm", bool "debug")
{
  # SETTING MOTION LEVELS THRESHOLDS [2]
  threshold_lm = default(threshold_lm, 4.0)
  threshold_hm = default(threshold_hm, 12.0)
  global threshold_lm = threshold_lm
  global threshold_hm = threshold_hm

  # ENABLING/DISABLING DEBUG INFORMATION [3]
  debug = default(debug, false)

  # INITIALIZING MOTION LEVEL
  global motion_level = 0

  # SETTING PRESENT CLIP [4]
  global clip = c

  # GETTING OUTPUT RESOLUTION [5]
  width = Width(Low_Motion_Filter(c))
  height = Height(Low_Motion_Filter(c))
  global c_resized = PointResize(c, width, height)

  # APPLYING MOTION FILTER ACCORDING TO MOTION LEVEL [6]
  c = ConditionalFilter(c, Low_Motion_Filter(c), c_resized, "motion_level", "=", "0") # [6a]
  c = ConditionalFilter(c, Medium_Motion_Filter(c), c, "motion_level", "=", "1") # [6b]
  c = ConditionalFilter(c, High_Motion_Filter(c), c, "motion_level", "=", "2") # [6c]

  # PRINTING DEBUG INFORMATION [7]
  c = (debug == true) ? ScriptClip(c, "Debug()") : c

  # GETTING MOTION LEVEL THROUGH MOTION ESTIMATION [8]
  c = FrameEvaluate(c, "ME()")

  # GETTING DIFFERENCES BETWEEN PAST/PRESENT FRAMES [9]
  c = FrameEvaluate(c, "global diff = 0.50*YDifferenceFromPrevious(clip) + 0.25*UDifferenceFromPrevious(clip)")
  return c
}

# DEBUG INFORMATION FUNCTION
function Debug(clip c)
{
  # PRINTING VERSION INFORMATION [10]
  c = Subtitle(c, "Quantified Motion Filter", x=20, y=30, font="lucida console", size=18, text_color=$)
  c = Subtitle(c, "by HomiE FR (homie.fr@wanadoo.fr)", x=20, y=45, font="lucida console", size=18, text_color=$)

  # PRINTING MOTION ESTIMATION INFORMATION [11]
  c = Subtitle(c, "motion estimation", x=20, y=85, font="lucida console", size=18, text_color=$)
  c = Subtitle(c, "diff = "+string(diff), x=20, y=110, font="lucida console", size=16, text_color=$)

  # PRINTING QUANTIFIED MOTION FILTER INFORMATION [12]
  c = Subtitle(c, "quantified motion filter", x=20, y=135, font="lucida console", size=18, text_color=$)
  c = (motion_level == 0) ? Subtitle(c, "scene type = low motion", x=20, y=160, font="lucida console", size=18, text_color=$) : c
  c = (motion_level == 1) ? Subtitle(c, "scene type = medium motion", x=20, y=160, font="lucida console", size=18, text_color=$) : c
  c = (motion_level == 2) ? Subtitle(c, "scene type = high motion", x=20, y=160, font="lucida console", size=18, text_color=$) : c
  return c
}
```

}

-----  
 This filter chain works like this:

- When Vdub requests a frame, AviSynth requests a frame from QMF.
  - ◆ QMF request a frame from FrameEvaluate [9].
  - ◆ After doing this the script [9] is evaluated, and the global variable *diff* is assigned after requesting a frame from AviSource. FrameEvaluate [9] requests a frame from FrameEvaluate [8].
  - ◆ Once again the script [8] is evaluated:
    - ◇ when evaluating *me()*, the global variable *motion\_level* is assigned for that frame [1]
  - ◆ If *debug=true*, a frame is requested from ScriptClip [7], and thus from Debug().
  - ◆ After that (and also when *debug* was set to false) a frame is requested from the last ConditionalFilter [6c], which requests a frame from [6b], which in turn requests a frame from [6a].
    - ◇ Note that in the end, a frame of *High\_Motion\_filter*, *Medium\_Motion\_filter*, or *Low\_Motion\_filter* is requested depending on the value of *motion\_level*.
- QMF request a frame from Telecide, Telecide from ConvertToYV12 and finally ConvertToYV12 from AviSource.
- AviSource produces the frame and sends it to ConvertToYV12, etc.

A few details were omitted, but this is how the script basically works.

**\$Date: 2009/10/11 11:43:31 \$**

## 13.37 ConditionalReader

ConditionalReader (*clip, string filename, string variablename, bool "show"*)

ConditionalReader allows you to import information from a text file, with different values for each frame – or a range of frames.

### 13.37.1 Parameters

Parameter	Description	Default
clip	The control clip. It is not touched, unless you specify <i>show=true</i> .	Not optional
filename	The file with the variables you want to set.	Not optional
variablename	The name of the variable you want the information inserted into.	Not optional
show	When set to true, the value given at this frame will be overlayed on the image.	false

### 13.37.2 File format

The file is plain text. All separation is done by spaces, and newline indicates a new data set. It is not case sensitive!

**TYPE (int|float|bool)**

You can only have one type of data in each file. Currently it is possible to have *float*, *int* or *bool* values. You specify this by using the **TYPE** keyword. You should always start out by specifying the type of data, as nothing is saved until this keyword has been found. It is not possible to change type once it has been set!

**DEFAULT <value>**

This specifies the default value of all frames. You should do this right after specifying the type, as it overwrites all defined frames. You can omit this setting, you must be sure to specify a setting for all frames, as it will lead to unexpected results otherwise.

**OFFSET <value>**

When specified, this will offset all framenumbers by a constant offset. This is done for all framenumbers which are set after this keyword.

**<framenumber> <value>**

This will set the value only for frame <framenumber>.

**R <startframe> <endframe> <value>**

This will apply a value to a range of frames. You should note that both start-frame AND end-frame are included.

**I <startframe> <endframe> <startvalue> <stopvalue>**

This will interpolate between two values over a range of frames. This only works on int and float values. You should note that both start-frame AND end-frame are included.

### 13.37.3 Types

As mentioned, the types can be either *float*, *int* or *bool*.

*Int* numbers is a number optionally preceeded with a sign.

*Float* is a decimal number containing a decimal point, optionally preceeded by a sign and optionally followed by the e or E character and a decimal number. Valid entries are *-732.103* or *7.12e4*.

*Bool* can either be *true*, *T*, *yes*, *false*, *F* or *no*.

### 13.37.4 Examples

#### 13.37.4.1 Basic usage

File.txt:

```
Type float
Default 3.45567

R 45 300 76.5654
```



## Avisynth 2.5 Selected External Plugin Reference

```
2 -671.454
72 -671.454
```

The file above will return float values. It will by default return 3.45567. However frames 45 to 300 it will return 76.5654. And frame 2 and 72 will return -671.454.

As you might notice – later changes overrule settings done earlier in the file. This is illustrated by frame '72' – even though it is inside the range of 45–300, the later value will be returned. On the other hand – if the range was specified AFTER '72 -671.454' – it would return 76.5654.

A script to invoke this file could be:

```
colorbars(512,512)
trim(0,500)
ScriptClip("subtitle(string(myvar))")
ConditionalReader("file.txt", "myvar", false)
```

This will put the values into the variable called "myvar", which is used by [Subtitle](#), invoked by [ScriptClip](#) to display the conditional value.

**Note!** The ConditionalReader() line comes **after** any use of "myvar" in your script.

### 13.37.4.2 Adjusting Overlay

#### AviSynth script:

```
colorbars(512,256)
a1 = trim(0,600)
a2 = MessageClip("Text clip")
overlay(a1,a2, y = 100, x = 110, mode="subtract", opacity=0, pc_range=true)
ConditionalReader("opacity.txt", "ol_opacity_offset", false)
ConditionalReader("xoffset.txt", "ol_x_offset", false)
```

#### xoffset.txt:

```
Type int
Default -50

I 25 50 -50 100
R 50 250 100
I 250 275 100 250
```

#### opacity.txt:

```
Type float
Default 0.0

I 25 50 0.0 1.0
R 50 250 1.0
I 250 275 1.0 0.0
```

Basically it defines keyframes for an x–offset and the opacity. Frame 25–>50 the opacity is scaled from 0.0 to 1.0, while the text is moving from left to right. The text is then kept steady from frame 50 to 250, whereafter it moves further to the right, while fading out.

It is easier to watch the clip above than completely describe what it does.

### 13.37.4.3 Complicated ApplyRange

As you may have noticed using a large number of [ApplyRange\(\)](#) calls in a script can lead to resource issue. Using [ConditionalReader](#) together with [ConditionalFilter](#) can lead to an efficient solution:

File.txt:

```
Type Bool
Default False

2 True
R 45 60 True
72 True
R 200 220 True
210 False
315 True
```

The file above will return boolean values. It will by default return False. However frames 2, 45 to 60, 72, 200 to 220 and 315 except for 210 it will return True. As you might notice, later changes overrule settings done earlier in the file. This is illustrated by frame '210' – even though it is inside the range of 200–220, the later value, False, will be returned.

A script to make use of this file could be:

```
colorbars(512,512)
trim(0,500)
A=Last
FlipHorizontal() # Add a complex filter chain
B=Last
ConditionalFilter(A, B, "MyVar", "==", "False", false)
ConditionalReader("File.txt", "MyVar", false)
```

This will put the values into the variable called "MyVar", which is used by [ConditionalFilter](#) to select between the unprocessed and flipped version of the source.

**Note!** The `ConditionalReader()` line comes **after** any use of "MyVar" in your script.

#### Changelog:

v2.60	Added OFFSET
-------	--------------

\$Date: 2009/09/12 15:10:22 \$

## 13.38 ConvertBackToYUY2 / ConvertToRGB / ConvertToRGB24 / ConvertToRGB32 / ConvertToY8 / ConvertToYUY2 / ConvertToYV12 / ConvertToYV16 / ConvertToYV24 / ConvertToYV411

```
ConvertToRGB (clip [, string "matrix"] [, bool "interlaced"] [, string "chromaplacement"] [, string "chromaresample"])
ConvertToRGB24 (clip [, string "matrix"] [, bool "interlaced"] [, string "chromaplacement"] [, string
```

## Avisynth 2.5 Selected External Plugin Reference

*"chromaresample")*

ConvertToRGB32 (*clip* [, *string "matrix"*] [, *bool "interlaced"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertToY8 (*clip* [, *string "matrix"*])

ConvertToYUY2 (*clip* [, *bool "interlaced"*] [, *string "matrix"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertToYV411 (*clip* [, *bool "interlaced"*] [, *string "matrix"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertToYV12 (*clip* [, *bool "interlaced"*] [, *string "matrix"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertToYV16 (*clip* [, *bool "interlaced"*] [, *string "matrix"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertToYV24 (*clip* [, *bool "interlaced"*] [, *string "matrix"*] [, *string "chromaplacement"*] [, *string "chromaresample"*])

ConvertBackToYUY2 (*clip* [, *string "matrix"*])

colorformats	planar/interleaved	chroma resolution
RGB	interleaved	full chroma – 4:4:4
RGB24	interleaved	full chroma – 4:4:4
RGB32	interleaved	full chroma – 4:4:4
YUY2	planar	chroma shared between 2 pixels – 4:2:2
Y8	planar/interleaved	no chroma – 4:0:0
YV411	planar	chroma shared between 4 pixels – 4:1:1
YV12	planar	chroma shared between 2x2 pixels – 4:2:0
YV16	planar	chroma shared between 2 pixels – 4:2:2
YV24	planar	full chroma – 4:4:4

*matrix*: Default Rec601. Controls the colour coefficients and scaling factors used in RGB – YUV conversions.

- "Rec601" : Use Rec.601 coefficients, scaled to TV range [16,235].
- "PC.601" : Use Rec.601 coefficients, keep full range [0,255].
- "Rec709" : Use Rec.709 coefficients, scaled to TV range.
- "PC.709" : Use Rec.709 coefficients, keep full range.

*interlaced*: Default false. Use interlaced layout for YV12 – YUV/RGB chroma conversions.

*chromaplacement* (added in v2.60): This determines the chroma placement when converting to or from YV12. It can be "MPEG2" (default), "MPEG1" and "DV".

*chromaresample* (added in v2.60): This determines which resizer is used in the conversion. It is used when the chroma resolution of the source and target is different. It can be all resamplers, default is "bicubic".

AviSynth prior to v2.50 can deal internally with two color formats, RGB and YUY2. Starting from v2.50 Avisynth can also deal with a third color format, YV12. These filters convert between them. If the video is

## Avisynth 2.5 Selected External Plugin Reference

already in the specified format, it will be passed through unchanged. RGB is assumed throughout this doc to mean RGBA = RGB32. `ConvertToRGB` converts to RGB32 unless your clip is RGB24. If you need 24-bit RGB for some reason, use `ConvertToRGB24` explicitly and `ConvertToRGB32` to do the reverse.

In v2.60 the following additional formats are supported: Y8 greyscale (it is both planar and interleaved since it contains no chroma; 4:0:0), YV411 (planar; YUV 4:1:1), YV16 (a planar version of YUY2; 4:2:2) and YV24 (planar; YUV 4:4:4).

Syntax and operation of `ConvertToRGB24` is identical to `ConvertToRGB`, except that the output format is 24-bit; if the source is RGB32, the alpha channel will be stripped.

Since v2.51/v2.52 an optional *interlaced* parameter is added (*interlaced*=false is the default operation). When set to false it is assumed that *clip* is progressive, when set to true it is assumed that *clip* is interlaced. This option is added because for example (assuming clip is interlaced YV12):

```
SeparateFields(clip)
ConvertToYV12
Weave
```

is upsampled incorrectly. Instead it is better to use:

```
ConvertToYV12(clip, interlaced=true)
```

Note, the *interlaced*=true setting only does something if the conversion YV12 <-> YUY2/RGB is requested, otherwise it's simply ignored. More about it can be found here "[Color conversions and interlaced / field-based video](#)".

Contrary to what one might expect, there is no unique way of converting YUV to RGB. In AviSynth the two most common ones are implemented: Rec.601 and Rec.709 (named after their official specifications). Although it will not be correct in all cases, the following should be correct in most cases:

The first one (Rec.601) should be used when your source is DivX/XviD or some analogue capture:

```
ConvertToRGB(clip)
```

The second one (Rec.709) should be used when your source is DVD or HDTV:

```
ConvertToRGB(clip, matrix="rec709")
```

In v2.56, the reverse is also available, that is

```
ConvertToYUY2(clip, matrix="rec709") or ConvertToYV12(clip, matrix="rec709")
```

In v2.56, *matrix*="pc.601" (and *matrix*="pc.709") enables you to do the RGB <-> YUV conversion while keeping the luma range, thus RGB [0,255] <-> YUV [0,255] (instead of the usual/default RGB [0,255] <-> YUV [16,235]).

All VirtualDub filters (loaded with `LoadVirtualDubPlugin`, see [Plugins](#)) support only RGB32 input.

**RGB24, RGB32:** The colors are stored as values of red, green and blue. In RGB32 there is an extra alpha channel for opacity. The image dimensions can have any values.

## Avisynth 2.5 Selected External Plugin Reference

**YUY2:** The picture is stored as a luma value Y and two color values U, V. For two horizontal pixels there is only one chroma value and two luma values (two Y's, one U, one V). Therefore the width has to be a multiple of two.

**YV8:** Greyscale. Thus the same as YV24 without the chroma planes.

**YV411:** Similar as YV12 but with only one chroma value for 4 pixels (a 1x4 square). The horizontal image dimension has to be a multiple of four.

**YV12:** The same as YUY2 but there is only one chroma value for 4 pixels (a 2x2 square). Both image dimensions have to be a multiple of two, if the video is interlaced the height has to be a multiple of four because the 2x2 square is taken from a field, not from a frame.

**YV16:** The same as YUY2 but planar instead of interleaved.

**YV24:** The same as YV12/YV16, but with full chroma.

Some functions check for the dimension rules, some round the parameters, there still can be some where an picture distortion or an error occurs.

Working in YUY2 is faster than in RGB. YV12 is even faster and is the native MPEG format, so there are fewer colorspace conversions.

Conversion back and forth is not lossless, so use as few conversions as possible. If multiple conversions are necessary, use `ConvertBackToYUY2` to convert to YUY2, if your source already has already once been YUY2. This will reduce colorblurring, but there is still some precision lost.

In most cases, the `ConvertToRGB` filter should not be necessary. If Avisynth's output is in YUY2 format and an application expects RGB, the system will use the installed YUY2 codec to make the conversion. However, if there's no installed YUY2 codec, or if (as is the case with ATI's and some other YUY2 codec) the codec converts from YUY2 to RGB incorrectly, you can use Avisynth's built-in filter to convert instead.

[Huffyuv](#) will act as the system YUY2 codec if there's no other codec installed, so if you install Huffyuv *and uninstall all other YUY2 codecs*, then you'll never need `ConvertToRGB`.

`ConvertToRGB24` and `ConvertToRGB32` can be used to force Avisynth to use a specific store method for RGB data. RGB24 data is often much slower to process than RGB32 data, so if your source is RGB24, you may get a speed gain by converting to RGB32. There are no known advantages of using RGB24 except that TMPGEnc and VFApi requires RGB24 input.

### Examples:

```
# There is a slight distortion caused by the conversion between YUV and RGB.
# Let's see if we can see it.
control = ConvertToYUY2()
test = ConvertToYUY2(ConvertToRGB(ConvertToYUY2(ConvertToRGB(control))))
test = ConvertToYUY2(ConvertToRGB(test))
return Subtract(test,control)
```

### Changes:

v2.60
-------

	Added: ConvertToY8, ConvertToYV411, ConvertToYV16, ConvertToYV24, chromaplacement and chromaresample
v2.50	ConvertToYV12

\$Date: 2009/09/12 15:10:22 \$

### 13.39 ConvertAudioTo8bit / ConvertAudioTo16bit / ConvertAudioTo24bit ConvertAudioTo32bit / ConvertAudioToFloat

ConvertAudioTo8bit (*clip*)  
 ConvertAudioTo16bit (*clip*)  
 ConvertAudioTo24bit (*clip*)  
 ConvertAudioTo32bit (*clip*)  
 ConvertAudioToFloat (*clip*)

The first four filters convert the audio samples to 8, 16, 24 and 32 bits, and ConvertAudioToFloat converts the audio samples to float. ConvertAudioTo8bit, ConvertAudioTo24bit, ConvertAudioTo32bit and ConvertAudioToFloat are available starting from v2.5.

Starting from v2.5 the audio samples will be automatically converted if any filters requires a special type of sample. This means that most filters will accept several types of input, but if a filter doesn't support the type of sample it is given, it will automatically convert the samples to something it supports. The internal formats supported in each filter is listed in the colorspace column. A specific sample type can be forced by using the ConvertAudio functions.

\$Date: 2004/12/23 22:00:52 \$

### 13.40 ConvertToMono

ConvertToMono (*clip*)

Prior to v2.5 it converts a stereo signal to mono, and starting from v2.5 it converts a multichannel signal to mono by averaging all channels with equal weights. If the signal is already in mono, it is returned untouched.

\$Date: 2004/03/07 22:44:06 \$

### 13.41 GeneralConvolution

GeneralConvolution (*clip*, *int "bias"*, *string "matrix"*, *float "divisor"*, *bool "auto"*)

This filter performs a matrix convolution.

<i>clip</i>	RGB32 clip
<i>bias</i> (default 0)	additive bias to adjust the total output intensity

## Avisynth 2.5 Selected External Plugin Reference

<i>matrix</i> (default "0 0 0 0 1 0 0 0 0")	can be a 3x3 or 5x5 matrix with 9 or 25 integer numbers between -256 and 256
<i>divisor</i> (default 1.0)	divides the output of the convolution (calculated before adding bias)
<i>auto</i> (default true)	Enables the auto scaling functionality. This divides the result by the sum of the elements of the matrix. The value of <i>divisor</i> is applied in addition to this auto scaling factor. If the sum of elements is zero, auto is disabled

The *divisor* is usually the sum of the elements of the matrix. But when the sum is zero, you must use *divisor* and the *bias* setting to correct the pixel values. The *bias* could be useful if the pixel values are negative due to the convolution. After adding a bias, the pixels are just clipped to zero (and 255 if they are larger than 255).

Around the borders the edge pixels are simply repeated to service the matrix.

Some examples:

# Blur:

```
GeneralConvolution(0, "
  10 10 10 10 10
  10 10 10 10 10
  10 10 16 10 10
  10 10 10 10 10
  10 10 10 10 10 ", 256, False)
```

# Horizontal (Sobel) edge detection:

```
GeneralConvolution(128, "
  1 2 1
  0 0 0
 -1 -2 -1 ", 8)
```

# Vertical (Sobel) Edge Detection:

```
GeneralConvolution(128, "
  1 0 -1
  2 0 -2
  1 0 -1 ", 8)
```

# Displacement (simply move the position  
# of the "1" for left, right, up, down)

```
GeneralConvolution(0, "
  0 1 0
  0 0 0
  0 0 0 ")
```

# Displacement by half pixel up (auto scaling):

```
GeneralConvolution(0, "
  0 1 0
  0 1 0
  0 0 0 ")
```

## Avisynth 2.5 Selected External Plugin Reference

# Displacement by half pixel right (manual scaling):

```
GeneralConvolution(0,"
  0  0  0
  0 128 128
  0  0  0 ", 256, False)
```

# Sharpness filter:

```
GeneralConvolution(0,"
  0  -1  0
 -1   5  -1
  0  -1  0 ", 1, True)
```

In this case, the new pixel values  $y(m,n)$  are given by

$$y(m,n) = (-1*x(m-1,n) - 1*x(m,n-1) + 5*x(m,n) - 1*x(m,n+1) - 1*x(m+1,n))/(-1-1+5-1-1)/1.0 + 0$$

# Slight blur filter with black level clipping and 25% brightening:

```
GeneralConvolution(-16,"
  0  12  0
 12 256 12
  0  12  0 ", 0.75 ,True)
```

In this case, the new pixel values  $y(m,n)$  are given by

$$y(m,n) = ( 12*x(m-1,n) + 12*x(m,n-1) + 256*x(m,n) + 12*x(m,n+1) + 12*x(m+1,n) )/(12+12+256+12+12)/0.75 - 16$$

Some other examples can be found [here](#).

### Changelog:

v2	Initial Release
v2.55	added divisor, auto

\$Date: 2008/05/28 21:24:49 \$

## 13.42 Crop / CropBottom

`Crop` (*clip, int left, int top, int width, int height, bool "align"*)  
`Crop` (*clip, int left, int top, int -right, int -bottom, bool "align"*)  
`CropBottom` (*clip, int count, bool align*)

`Crop` crops excess pixels off of each frame.

If your source video has 720x480 resolution, and you want to reduce it to 352x240 for VideoCD, here's the correct way to do it:

```
# Convert CCIR601 to VCD, preserving the correct aspect ratio
ReduceBy2
Crop(4,0,352,240)
```

See [colorspace conversion filters](#) for limitations when using different color formats.



## Avisynth 2.5 Selected External Plugin Reference

If a negative value is entered in the *width* and *height* these are also treated as offsets. For example:

```
# Crop 16 pixels all the way around the picture, regardless of image size:  
Crop(16,16,-16,-16)
```

In v2.53 an option *align* (false by default) is added:

Cropping an YUY2/RGB32 image is always mod4 (four bytes). However, when reading x bytes (an int), it is faster when the read is aligned to a modx placement in memory. MMX/SSE likes 8-byte alignment and SSE2 likes 16-byte alignment. If the data is NOT aligned, each read/write operation will be delayed at least 4 cycles. So images are always aligned to mod16 when they are created by AviSynth.

If an image has been cropped, they will sometimes be placed unaligned in memory – "*align* = true" will copy the entire frame from the unaligned memory placement to an aligned one. So if the penalty of the following filter is larger than the penalty of a complete image copy, using "*align* = true" will be faster. Especially when it is followed by smoothers.

The alternative CropBottom syntax is useful for cropping garbage off the bottom of a clip captured from VHS tape. It removes *count* lines from the bottom of each frame.

### 13.43 Memory alignment

In v2.53 an option *align* (false by default) is added:

Cropping an YUY2/RGB32 image is always mod4 (four bytes). However, when reading x bytes (an int), it is faster when the read is aligned to a modx placement in memory. MMX/SSE likes 8-byte alignment and SSE2 likes 16-byte alignment. If the data is NOT aligned, each read/write operation will be delayed at least 4 cycles. So images are always aligned to mod16 when they are created by AviSynth.

If an image has been cropped, they will sometimes be placed unaligned in memory – "*align* = true" will copy the entire frame from the unaligned memory placement to an aligned one. So if the penalty of the following filter is larger than the penalty of a complete image copy, using "*align*=true" will be faster. Especially when it is followed by smoothers.

### 13.44 Crop restrictions

In order to preserve the data structure of the different colorspace, the following mods should be used. You will not get an error message if they are not obeyed, but it may create strange artifacts. For a complete discussion on this, see [DataStorageInAviSynth ...](#)

Colorspace	Width	Height	
		progressive video	interlaced video
RGB	<i>no restriction</i>	<i>no restriction</i>	mod-2
YUY2	mod-2	<i>no restriction</i>	mod-2
YV12	mod-2	mod-2	mod-4

NOTE: The [resize functions](#) optionally allow fractional pixel cropping of the input frame, this results in a

weighting being applied to the edge pixels being resized. These options may be used if the mod-n format dimension restriction of crop are inconvenient. In sum — "For cropping off hard artifacts like VHS head noise or letterbox borders always use Crop. For extracting a portion of an image and to maintain accurate edge resampling use the resize cropping parameters." ([Doom9 thread](#))

\$Date: 2009/09/12 15:10:22 \$

### 13.45 DelayAudio

DelayAudio (*clip, float seconds*)

DelayAudio delays the audio track by *seconds* seconds. *Seconds* can be negative and/or have a fractional part.

```
# Play audio half a second earlier
DelayAudio(-0.5)
```

\$Date: 2004/03/07 22:44:06 \$

### 13.46 DeleteFrame

DeleteFrame (*clip, int frame [, ...]*)

DeleteFrame deletes a set of frames, given as a number of arguments. The sound track is not modified, so if you use this filter to delete many frames you may get noticeable desynchronization.

#### Examples:

```
DeleteFrame(3, 9, 21, 42) # delete frames 3, 9, 21 and 42
```

If you want to delete a range of frames (*a* to *b*, say) along with the corresponding portion of the soundtrack, you can do it with [Trim](#) like this:

```
Trim(0,a-1) ++ Trim(b+1,0)
```

Or like this:

```
Loop(0,a,b)
```

#### Changelog:

v2.58	added support for multiple arguments
-------	--------------------------------------

\$Date: 2008/06/16 19:42:53 \$

### 13.47 DirectShowSource

DirectShowSource (*string filename, float "fps", bool "seek", bool "audio", bool "video", bool "convertfps", bool "seekzero", int "timeout", string "pixel\_type", int "framecount", string "logfile", int "logmask"*)

## Avisynth 2.5 Selected External Plugin Reference

`DirectShowSource` reads *filename* using DirectShow, the same multimedia playback system which Windows Media Player uses. It can read most formats which Media Player can play, including MPEG, MP3, and QuickTime, as well as AVI files that `AVISource` doesn't support (like DV type 1, or files using DirectShow-only codecs). Try reading AVI files with `AVISource` first, and if that doesn't work then try this filter instead.

There are some caveats:

- Some decoders (notably MS MPEG-4) will produce upside-down video. You'll have to use [FlipVertical](#).
- DirectShow video decoders are not required to support frame-accurate seeking. In most cases seeking will work, but on some it might not.
- DirectShow video decoders are not even required to tell you the frame rate of the incoming video. Most do, but the ASF decoder doesn't. You have to specify the frame rate using the `fps` parameter, like this: `DirectShowSource ("video.asf", fps=15)`.
- This version automatically detects the Microsoft DV codec and sets it to decode at full (instead of half) resolution. I guess this isn't a caveat. :-)
- Also this version attempts to disable any decoder based deinterlacing.

*fps*: This is sometimes needed to specify the framerate of the video. If the framerate or the number of frames is incorrect (this can happen with asf or mov clips), use this option to force the correct framerate.

*seek* = true (in v2.53): There is full seeking support (available on most file formats). If problems occur try enabling the *seekzero* option first, if seeking still cause problems completely disable seeking. With seeking disabled the audio stream returns silence and the video stream the last rendered frame when trying to seek backwards. Note the Avisynth cache may provide limited access to the previous few frames, beyond that the last frame rendered will be returned.

*audio* = true (in v2.53): There is audio support in DirectShowSource. DirectShowSource is able to open formats like WAV/DTS/AC3/MP3, provided you can play them in WMP for example (more exact: provided they are rendered correctly in graphedit). The channel ordering is the same as in the [\[wave-format-extensible format\]](#), because the input is always decompressed to WAV. For more information, see also `GetChannel`. Avisynth loads 8, 16, 24 and 32 bit int PCM samples, and float PCM format, and any number of channels.

*video* = true (in v2.52): When setting it to false, it lets you open the audio only.

*convertfps* = false (in v2.56): When setting it to true, it turns variable framerate video (vfr) into constant framerate video (cfr) by duplicating or skipping frames. This is useful when you want to open vfr video (for example mkv, rmvb, mp4, asf or wmv with hybrid video) in Avisynth. It is most useful when the *fps* parameter is set to the least common multiple of the component vfr rates, e.g. 120 or 119.880.

*seekzero* = false (in v2.56): An option to restrict seeking only back to the beginning. It allows limited seeking with files like unindexed ASF. Seeking forwards is of course done the hard way (by reading all samples).

*timeout* = 60000 (in milliseconds; 60000 ms = 1 min) (in v2.56): To set time to wait when DirectShow refuses to render. Positive values cause the return of blank frames for video and silence for audio streams. Negative values cause a runtime Avisynth exception to be thrown.

*pixel\_type* (in v2.56): The pixel type of the resulting clip, it can be "YV12", "YUY2", "ARGB", "RGB32", "RGB24", "YUV", "RGB" or "AUTO". By default, upstream DirectShow filters are free to bid all of their

## Avisynth 2.5 Selected External Plugin Reference

supported media types in the order of their choice. A few DirectShow filters get this wrong. The **pixel\_type** argument limits the acceptable video stream subformats for the IPin negotiation. Note the graph builder may add a format converter to satisfy your request, so make sure the codec in use can actually decode to your chosen format. The M\$ format converter is just adequate. The "YUV" and "RGB" pseudo-types restrict the negotiation to all supported YUV or RGB formats respectively. The "AUTO" pseudo-type permits the negotiation to use all relevant formats in the order of preference YV12, YUY2, ARGB, RGB32, RGB24. Many DirectShow filters get this wrong, which is why it is not enabled by default. The option exists so you have enough control to encourage the maximum range of filters to serve your media. (See [discussion](#).)

*framecount* (in v2.57): This is sometimes needed to specify the framecount of the video. If the framerate or the number of frames is incorrect (this can happen with asf or mov clips), use this option to force the correct number of frames. If *fps* is also specified the length of the audio stream is also adjusted.

*logfile* (in v2.57): Use this option to specify the name of a debugging logfile.

*logmask* = 35 (in v2.57): When a logfile is specified, use this option to select which information is logged.

Value	Data
1	Format Negotiation
2	Receive samples
4	GetFrame/GetAudio calls
8	Directshow callbacks
16	Requests to Directshow
32	Errors
64	COM object use count
128	New objects
256	Extra info
512	Wait events

Add the values together of the data you need logged. Specify -1 to log everything. The default, 35, logs Format Negotiation, Received samples and Errors. i.e 1+2+32

### 13.47.1 Examples

Opens an avi with the first available RGB format (without audio):

```
DirectShowSource("F:\TestStreams\xvid.avi",  
\      fps=25, audio=false, pixel_type="RGB")
```

Opens a DV clip with the MS DV decoder:

```
DirectShowSource("F:\DVCodecs\Analysis\Ced_dv.avi") # MS-DV
```

Opens a variable framerate mkv as 119.88 by adding frames (ensuring sync):

```
DirectShowSource("F:\Guides\Hybrid\vfr_startrek.mkv",  
\      fps=119.88, convertfps=true)
```

Opens a realmedia \*rmvb clip:

## Avisynth 2.5 Selected External Plugin Reference

```
DirectShowSource("F:\test.rmvb", fps=24, convertfps=true)
```

Opens a GraphEdit file:

```
V=DirectShowSource("F:\vid_graph.grf", audio=False) # video only (audio renderer removed)
A=DirectShowSource("F:\aud_graph.grf", video=False) # audio only (video renderer removed)
AudioDub(V, A)
```

See below for some audio examples.

### 13.47.2 Troubleshooting video and audio problems

AviSynth will by default try to open only the media it can open without any problems. If one component cannot be opened it will simply not be added to the output. This will also mean that if there is a problem, you will not see the error. To get the error message to the missing component, use `audio=false` or `video=false` and disable the component that is actually working. This way AviSynth will print out the error message of the component that doesn't work.

#### 13.47.2.1 RenderFile, the filter graph manager won't talk to me

This is a common error that occurs when DirectShow isn't able to deliver any format that is readable to AviSynth. Try creating a filter graph manually and see if you are able to construct a filter graph that delivers any output AviSynth can open. If not, you might need to download additional DirectShow filters that can deliver correct material.

#### 13.47.2.2 The samplerate is wrong

Some filters might have problems reporting the right samplerate, and then correct this when the file is actually playing. Unfortunately there is no way for AviSynth to correct this once the file has been opened. Use [AssumeSampleRate](#) and set the correct samplerate to fix this problem.

#### 13.47.2.3 My sound is choppy

Unfortunately Directshow is not required to support sample exact seeking. Open the sound another way, or demux your video file and serve it to AviSynth another way. Otherwise you can specify `"seekzero = true"` or `"seek = false"` as parameters or use the [EnsureVBRMP3Sync](#) filter to enforce linear access to the Directshow audio stream.

#### 13.47.2.4 My sound is out of sync

This can happen especially with WMV, apparently due to variable frame rate video being returned. Determine what the fps should be and set it explicitly, and also `"ConvertFPS"` to force it to remain constant. And [EnsureVBRMP3Sync](#) reduces problems with variable rate audio.

```
DirectShowSource("video.wmv", fps=25, ConvertFPS=True)
EnsureVBRMP3Sync()
```

#### 13.47.2.5 My ASF renders start fast and finish slow

Microsoft in their infinite wisdom chose to implement ASF stream timing in the ASF demuxer. As a result it is not possible to strip ASF format files any faster than realtime. This is most apparent when you first start to process the streams, usually after opening the Avisynth script it takes you a while to configure your video

editor, all this time the muxer is accumulating *credit* time. When you then start to process your stream it races away at maximum speed until you catch up to realtime at which point it slows down to the realtime rate of the source material. This feature makes it impossible to use Avisynth to reclock 24fps ASF material upto 25fps for direct PAL playback.

### 13.47.3 Common tasks

This section will describe various tasks that might not be 100% obvious. :)

#### 13.47.3.1 Opening GRF files

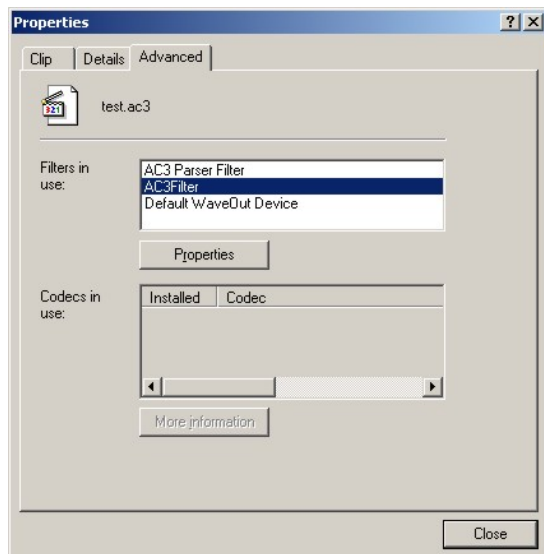
GraphEdit GRF-files are automatically detected by a .grf filename extension and directly loaded by DirectShowSource. For AviSynth to be able to connect to it, you must leave a pin open in GraphEdit of a media types that AviSynth is able to connect to. AviSynth will not attempt to disconnect any filters, so it is important that the output type is correct. DirectShowSource only accepts YV12, YUY2, ARGB, RGB32 and RGB24 video formats and 32, 24, 16 and 8 bit PCM and IEEE FLOAT audio formats.

A given GRF-file should only target one of an audio or video stream to avoid confusion when directshowsource attempts the connection to your open pin(s). From version 2.57 this single stream restriction is enforced.

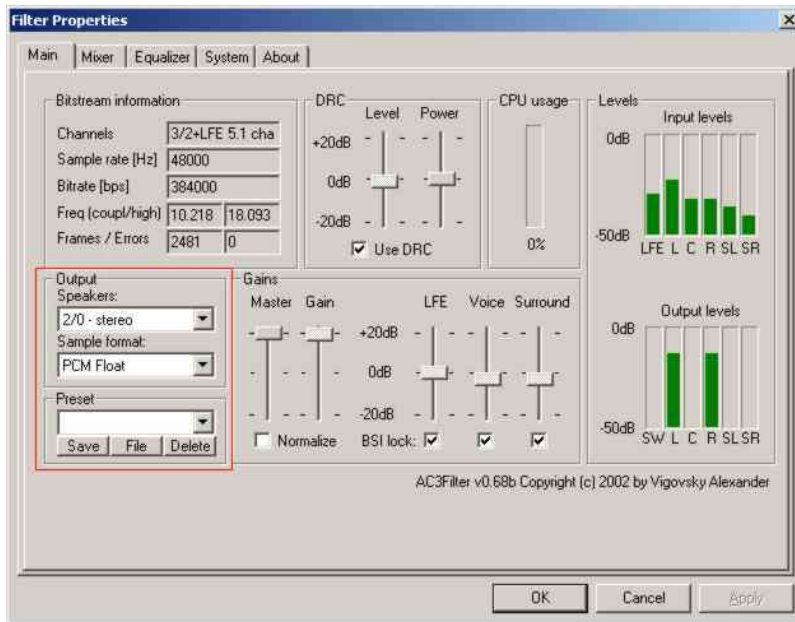
#### 13.47.3.2 Downmixing AC3 to stereo

There are essentially two ways to do this. The first is to set the downmixing in the configuration of your AC3 decoder itself, and the second one is to use the external downmixer of "Trombettworks":

1) Install AC3filter. Open the AC3 file in WMP6.4 and select the file properties. Set the output of AC3Filter on **2/0 – stereo**. If you want the best possible quality, select PCM Float as Sample format.



## Avisynth 2.5 Selected External Plugin Reference



Make the following script:

```
v = Mpeg2Source("e:\movie.d2v")
a = DirectShowSource("e:\Temp\Test2\test.ac3")
AudioDub(v,a)
```

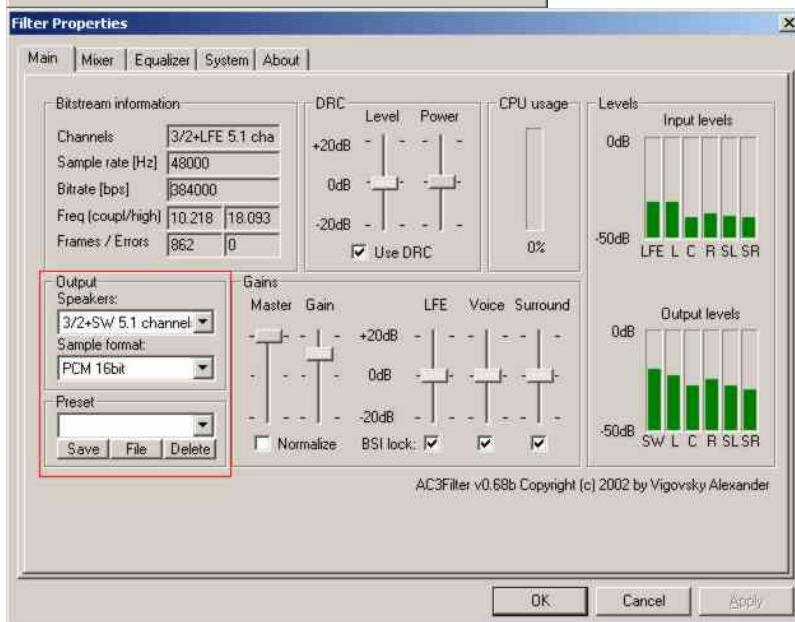
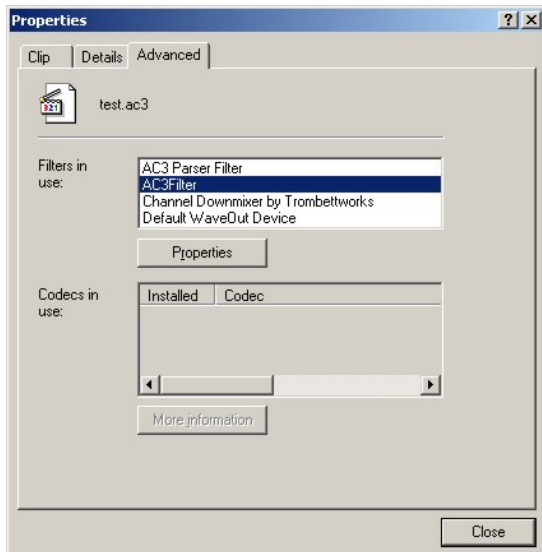
Finally, open the script in vdup and convert the audio stream to MP3 (of course you can also demux the downmixed WAV stream if needed).

2) Register the directshow filter [Channel Downmixer by Trombettworks](#) (under start -> run):

```
regsvr32 ChannelDownmixer.ax
```

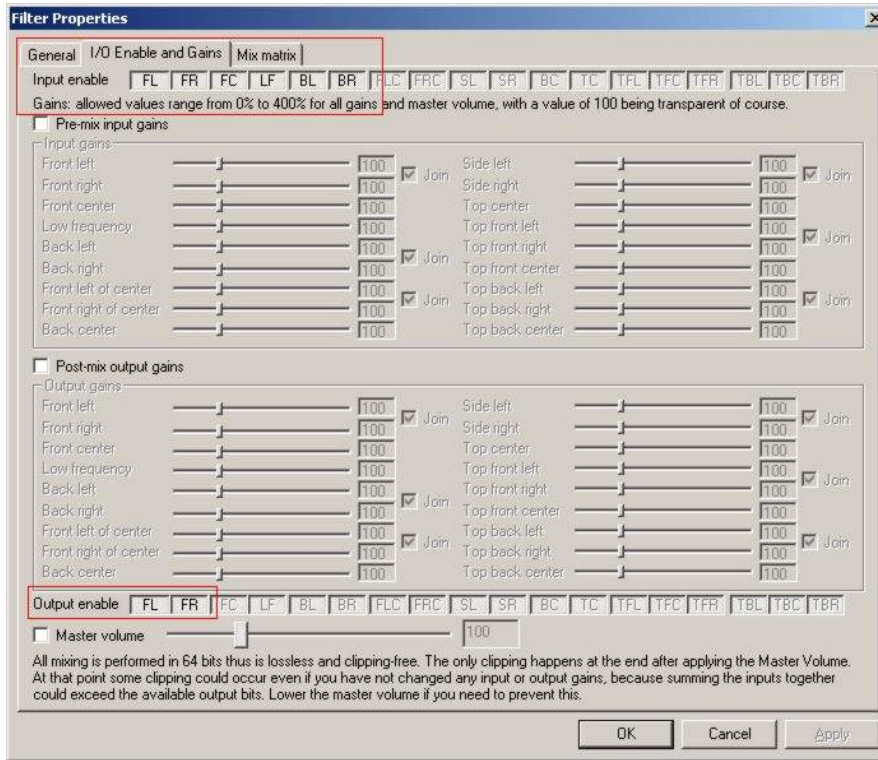
Open the AC3 file in WMP6.4 and select the file properties. Set the output of AC3Filter on **3/2+SW 5.1 channels** (this downmixer can't handle PCM Float, thus PCM 16 bit is selected here). In the properties of the downmixer, the number of input and output channels should be detected automatically. Check whether this is indeed correct.

## Avisynth 2.5 Selected External Plugin Reference





## Avisynth 2.5 Selected External Plugin Reference



Make the following script:

```
v = Mpeg2Source("e:\movie.d2v")
a = DirectShowSource("e:\Temp\Test2\test.ac3")
AudioDub(v,a)
```

Finally, open the script in vdup and convert the audio stream to MP3 (of course you can also demux the downmixed WAV stream if needed).

For some reason, I can't get this to work with DTS streams :(

### Changes

v2.56	convertfps turns vfr into constant cfr by adding frames
	seekzero restricts seeking to beginning only
	timeout controls response to recalcitrant graphs
v2.57	pixel_type specifies/restricts output video pixel format
	framecount overrides the length of the

streams.
logfile and logmask specify debug logging.

\$Date: 2009/09/12 15:10:22 \$

## 13.48 Dissolve

Dissolve (*clip1, clip2 [, ...], int overlap, float "fps"*)

Dissolve is like [AlignedSplice](#), except that the clips are combined with some overlap. The last *overlap* frames of the first video stream are blended progressively with the first *overlap* frames of the second video stream so that the streams fade into each other. The audio streams are blended similarly.

With audio only clips, by default, overlap is in units of 1/24th seconds. Set Fps=AudioRate() if sample exact audio positioning is required.

The term "dissolve" is sometimes used for a different effect in which the transition is pointwise rather than gradated. This filter won't do that.

Also see [here](#) for the resulting clip properties.

\$Date: 2005/01/21 07:48:43 \$

## 13.49 DoubleWeave

DoubleWeave (*clip*)

If the input clip is field-based, the DoubleWeave filter operates like [Weave](#), except that it produces double the number of frames: instead of combining fields 0 and 1 into frame 0, fields 2 and 3 into frame 1, and so on, it combines fields 0 and 1 into frame 0, fields 1 and 2 into frame 1, and so on. It does not change the frame rate or frame count.

If the input clip is frame-based, this filter acts just as though you'd separated it into fields with [SeparateFields](#) first, only faster!

[Weave](#) is actually just a shorthand for DoubleWeave followed by [SelectEven](#).

Most likely you will want to use a filter like [SelectOdd](#) or [PullDown](#) after using this filter, unless you really want a 50fps or 60fps video. It may seem inefficient to interlace every pair of fields only to immediately throw away half of the resulting frames. But actually, because Avisynth only generates frames on demand, frames that are not needed will never be generated in the first place.

If you're processing field-based video, like video-camera footage, you probably won't need this filter. But if you're processing NTSC video converted from film and you plan to use the PullDown filter, you need to use DoubleWeave first. See the PullDown filter for an explanation.

If you're processing PAL video converted from film, you don't need PullDown, but you might want to use DoubleWeave in the following situation:

## Avisynth 2.5 Selected External Plugin Reference

```
# Duplicate the functionality of the VirtualDub "PAL deinterlace" filter
DoubleWeave
SelectOdd
```

**\$Date: 2005/01/21 07:47:22 \$**

### 13.50 DuplicateFrame

DuplicateFrame (*clip*, *int frame* [, ...])

DuplicateFrame is the opposite of [DeleteFrame](#). It duplicates a set of frames given as a number of arguments. As with DeleteFrame, the sound track is not modified.

#### Examples:

```
DuplicateFrame(3, 3, 21, 42) # Add 4 frames
```

#### Changelog:

v2.58	added support for multiple arguments
-------	--------------------------------------

**\$Date: 2008/06/16 19:42:53 \$**

### 13.51 EnsureVBRMP3Sync

EnsureVBRMP3Sync (*clip*)

EnsureVBRMP3Sync will ensure synchronization of video with variable bitrate audio (MP3-AVI's for example) during seeking or trimming. It does so by buffering the audio. The name of the filter is a bit misleading, since it is useful for every stream with variable bitrate audio and not just for MP3.

It will slow seeking down considerably, but is very useful when using Trim for instance. Always use it before trimming.

```
# Ensures that soundtrack is in sync after trimming
Avisource("movie.avi")
EnsureVBRMP3Sync()
Trim(250,2500)
```

**\$Date: 2007/05/05 09:39:34 \$**

### 13.52 FadeIn / FadeIn0 / FadeIn2 / FadeIO0 / FadeIO / FadeIO2 / FadeOut / FadeOut0 / FadeOut2

```
FadeIn (clip clip, int num_frames, int "color", float "fps")
FadeIO (clip clip, int num_frames, int "color", float "fps")
FadeOut (clip clip, int num_frames, int "color", float "fps")
```

```
FadeIn0 (clip clip, int num_frames, int "color", float "fps")
FadeIO0 (clip clip, int num_frames, int "color", float "fps")
```

## Avisynth 2.5 Selected External Plugin Reference

FadeOut0 (*clip clip, int num\_frames, int "color", float "fps"*)

FadeIn2 (*clip clip, int num\_frames, int "color", float "fps"*)

FadeIO2 (*clip clip, int num\_frames, int "color", float "fps"*)

FadeOut2 (*clip clip, int num\_frames, int "color", float "fps"*)

FadeOut cause the video stream to fade linearly to black or the specified RGB color at the end. Similarly FadeIn cause the video stream to fade linearly from black or the specified RGB color at the start. FadeIO is a combination of the respective FadeIn and FadeOut functions. The sound track (if present) also fades linearly to or from silence. The fading affects only the last *num\_frames* frames of the video. The last frame of the video becomes almost-but-not-quite black (or the specified color). An additional perfectly black (or the specified color) frame is added at the end, thus increasing the total frame count by one.

FadeIn0 / FadeOut0 do not include the extra frame. It is useful when processing Audio only clips or chaining two or more fades to get a square law or a cube law fading effects. e.g Clip.FadeOut0(60).FadeOut0(60).FadeOut(60) gives a much sharper attack and gentler tailoff. The 50% point is at frame 12 of the fade, at frame 30 the fade is 12.5%, at frame 45, 1.6% the effectiveness is more pronounced with audio.

FadeIn2 / FadeOut2 works similarly, except that two black (or color) frames are added at the end instead of one. The main purpose of this is to work around a bug in Windows Media Player. All the WMP versions that I've tested fail to play the last frame of an MPEG file – instead, they stop on the next-to-last frame when playback ends. This leaves an unsightly almost-but-not-quite-black frame showing on the screen when the movie ends if you use FadeOut. FadeOut2 avoids this problem.

The *color* parameter is optional, default=0 <black>, and is specified as an RGB value regardless of whether the clip format is RGB or YUV based. See [here](#) for more information on specifying colors.

The *fps* parameter is optional, default=24.0, and provides a reference for *num\_frames* in audio only clips. It is ignored if a video stream is present. Set fps=AudioRate() if sample exact audio positioning is required.

FadeOut(clip, n) is just a shorthand for [Dissolve](#)(clip, [Blackness](#) (clip, n+1, color=\$000000), n) (or instead of n+1, n+2 for FadeOut2 and n for FadeOut0).

### Changelog:

Until v2.06	the FadeIn / FadeIn2 commands do not exist, but you can get the same effect by reversing the arguments to Dissolve: Dissolve(Blackness(clip, n+1, color=\$000000), clip, n).
v2.07	FadeIO / FadeIO2 commands are added and the <i>color</i> parameter is added to all fade functions.
v2.56	FadeIn0 / FadeIO0 / FadeOut0 commands are added and the <i>fps</i> parameter is added to all fade functions.

\$Date: 2009/10/11 11:43:32 \$

## 13.53 FixBrokenChromaUpsampling

`FixBrokenChromaUpsampling` (*clip*)

The free Canopus DV Codec v1.00 upsamples the chroma channels incorrectly (although newer non-free versions appear to work fine). Chroma is [duplicated from the other field](#), resulting in the famous [Chroma Upsampling Error](#).

`FixBrokenChromaUpsampling` filter compensates for it. You should put this after `AviSource` if you're using the above Canopus DV codec. Old versions of the DirectShow based MS DV codec also might have this problem (the one that comes with DirectX7 (but i need to check this), the one that comes with DirectX8/9 works fine).

The Canopus DV codec swaps the chroma of the middle 2 for each group of 4 lines:

frame_correct	frame_Canopus
line 1	line 1
line 2	line 3
line 3	line 2
line 4	line 4
line 5	line 5
line 6	line 7
line 7	line 6
line 8	line 8

For each group of 4 lines `FixBrokenChromaUpsampling` corrects this by swapping the chroma of the middle 2 back:

**\$Date: 2005/11/08 12:37:33 \$**

# 14 FixLuminance

FixLuminance (*clip, int intercept, int slope*)

My VCR has the annoying habit of making the top of each frame brighter than the bottom. The purpose of this filter is to progressively darken the top of the image to compensate for this. If you've noticed a similar problem with your VCR, email me and I'll finish this explanation of how the filter works. Otherwise, you can pretend it doesn't exist.

This filter works only with YUY2 input.

⌘Date: 2004/03/07 22:44:06 ⌘

## 14.1 FlipHorizontal / FlipVertical

FlipHorizontal (*clip*)

FlipVertical (*clip*)

FlipVertical flips the video upside-down. Useful for dealing with broken video codecs. Likewise FlipHorizontal (which is present in v2.5) flips the video from left to right.

⌘Date: 2004/03/07 22:44:06 ⌘

## 14.2 AssumeFPS

AssumeFPS (*clip, float fps, bool "sync\_audio"*)

AssumeFPS (*clip, int numerator [, int denominator], bool "sync\_audio"*)

AssumeFPS (*clip1, clip2, bool "sync\_audio"*)

AssumeFPS (*clip, string preset, bool "sync\_audio"*)

The AssumeFPS filter changes the frame rate without changing the frame count (causing the video to play faster or slower). It only sets the framerate-parameter

If *sync\_audio* is true, it also changes the audio sample rate by the same ratio, the pitch of the resulting audio gets shifted.

This is also a method to change only the sample rate of the audio alone.

In v2.55, if *clip2* is present, the framerate of *clip1* will be adjusted to match the one of *clip2*. This is useful when you want to add two clips with slightly different framerate.

In v2.57, the behaviour with respect to the framerate is a bit changed. The main issue is that users are allowed to specify the framerate as float, but the NTSC (FILM and Video) and PAL standards require ratios as framerate. Besides this AviSynth exports the framerate as a ratio, so when specifying a float, it will be converted to a ratio. The ratios of the standards are given by 24000/1001 for 23.976 (FILM) and 30000/1001 for 29.97 (Video). **When specifying these floats, they are exported by AviSynth as ratios, but not as the standard ratios.** One of the reasons for this is, that those floats are approximations (remember that  $24000/1001 = 23.9760239760\dots$ ), so how should AviSynth know how to choose the correct ratio? In order to overcome this issue, the user can use AssumeFPS(24000,1001) or simply AssumeFPS("ntsc\_film").

## Avisynth 2.5 Selected External Plugin Reference

Another problem is that the converted floats were (in v2.56 and older) exported with 64 bit precision, resulting in very large numerators and denominators, making some players crash. To overcome this, a smart float-ratio is added internally, and the framerates are approximated accurately by ratios of small numbers. For example, AssumeFPS(23.976) is converted to AssumeFPS(2997,125) as can be checked with [Info](#).

### Presets:

standard	numerator	denominator
"ntsc_film"	24000	1001
"ntsc_video"	30000	1001
"ntsc_double"	60000	1001
"ntsc_quad"	120000	1001
"ntsc_round_film"	2997	125
"ntsc_round_video"	2997	100
"ntsc_round_double"	2997	50
"ntsc_round_quad"	2997	25
"film"	24	1
"pal_film"	25	1
"pal_video"	25	1
"pal_double"	50	1
"pal_quad"	100	1

### Examples PAL +4% Telecine conversion:

```
AVISource("FILM_clip.avi")           # Get 24fps clip
LanczosResize(768,576)                # Resize to PAL square-pixel frame size.
AssumeFPS(25, 1, true)                # Convert frame rate to PAL, also adjust audio.
SSRC(44100)                            # Restore audio sample rate to a standard rate.
```

The +4% speed up is conventionally used for displaying 24fps film on PAL television. The slight increase in pitch and tempo is readily accepted by viewers of PAL material.

## 14.3 AssumeScaledFPS

AssumeScaledFPS (*clip*, *int "multiplier"*, *int "divisor"*, *bool "sync\_audio"*)

The AssumeScaledFPS filter scales the frame rate without changing the frame count. The numerator is multiplied by the multiplier, the denominator is multiplied by the divisor, the resulting rational FPS fraction is normalized, if either the resulting numerator or denominator exceed 31 bits the result is rounded and scaled. This allows exact rational scaling to be applied to the FPS property of a clip.

If *sync\_audio* is true, it also changes the audio sample rate by the same ratio, the pitch of the resulting audio gets shifted.

Available in v2.56.

## 14.4 ChangeFPS

ChangeFPS (*clip*, *float fps*, *bool "linear"*)

ChangeFPS (*clip*, *int numerator* [, *int denominator*], *bool "linear"*)

ChangeFPS (*clip1*, *clip2*, *bool "linear"*)

ChangeFPS (*clip*, *string preset*, *bool "linear"*)

ChangeFPS changes the frame rate by deleting or duplicating frames.

Up to v2.05, the video gets truncated or filled up to preserve playback speed and play time (the number of frames was not changed). In later versions, the behaviour has been changed and the number of frames is increased or decreased like in ConvertFPS.

In v2.54, an option *linear* = true/false is added to ChangeFPS. This will make AviSynth request frames in a linear fashion, when skipping frames. Default is true.

In v2.56, if *clip2* is present, the framerate of *clip1* will be adjusted to match that of *clip2*.

In v2.57, the behaviour with respect to the framerate is a bit changed. See AssumeFPS.

### Examples PAL→NTSC conversion:

```
AVISource("PAL_clip.avi")           # Get clip
Bob(height=480)                     # Separate fields and interpolate them to full height.
BicubicResize(640,480)              # Resize to NTSC square-pixel frame size.
ChangeFPS(60000, 1001)              # Convert field rate to NTSC, by duplicating fields.
SeparateFields.SelectEvery(4,0,3)   # Undo Bob, even field first. Use SelectEvery(4,1,2) for
Weave                                # Finish undoing Bob.
```

The effect is similar to 3–2 telecine pull down. Regular viewers of PAL material may notice a motion stutter that viewers of NTSC material readily ignore as for telecined film.

## 14.5 ConvertFPS

ConvertFPS (*clip*, *float new\_rate*, *int "zone"*, *int "vbi"*)

ConvertFPS (*clip*, *int numerator* [, *int denominator*], *int "zone"*, *int "vbi"*)

ConvertFPS (*clip1*, *clip2*, *int "zone"*, *int "vbi"*)

ConvertFPS (*clip*, *string preset*, *int "zone"*, *int "vbi"*)

The filter attempts to convert the frame rate of *clip* to *new\_rate* without dropping or inserting frames, providing a smooth conversion with results similar to those of standalone converter boxes. The output will have (almost) the same duration as *clip*, but the number of frames will change proportional to the ratio of target and source frame rates.

In v2.56, if *clip2* is present, the framerate of *clip1* will be adjusted to match that of *clip2*.

In v2.57, the behaviour with respect to the framerate is a bit changed. See AssumeFPS.

The filter has two operating modes. If the optional argument *zone* is not present, it will blend adjacent video frames, weighted by a blend factor proportional to the frames' relative timing ("Blend Mode"). If *zone* is present, it will switch from one video frame to the next ("Switch Mode") whenever a new source frame



## Avisynth 2.5 Selected External Plugin Reference

begins, that is, usually somewhere in the middle of a target frame. Switch Mode assumes that the output will be shown on a TV where each frame is scanned from top to bottom. The parameter *zone* specifies the height of the transition region in which the current frame will be blended into the next.

Blend Mode will cause visible, although slight, blurring of motion. This is a typical artifact of frame rate conversion and can be seen on commercial video tapes and TV programs as well. When working with interlaced video, it is important to let the filter operate on individual fields, not on the interlaced frames. (See examples below.)

Switch Mode is an attempt to avoid motion blurring, but comes at the expense of slight flicker and motion artifacts. Horizontal and vertical pans may show a slight wobble. Still frames from this conversion show "broken" or "bent" vertical lines in moving scenes. Scene transitions may occur in the middle of a frame. Nevertheless, the results do look less blurry than in "Blend Mode".

Neither mode is perfect. Which one to choose depends on personal preference and on the footage to be converted. Switch Mode is probably only suitable if the output will be shown on a TV, not on a computer screen.

Frame rate conversion is inherently difficult. This filter implements two common methods used by commercial Prosumer-level converter systems. The results are typically quite good. More sophisticated systems employ motion interpolation algorithms, which are difficult to get right, but, if done right, do yield superior results.

Footage converted with this filter should not be converted again. Blurriness builds up quickly in subsequent generations.

The audio data are not touched by this filter. Audio will remain synchronized, although the length of the audio data may slightly differ from that of the video data after the conversion. This is because the output can only contain an integer number of frames. This effect will be more pronounced for shorter clips. The difference in length should be ignored.

Parameters:

- new\_rate* Target frame rate. Can be integer or floating point number. In Blend Mode, *new\_rate* must be at least 2/3 (66.7%) of the source frame rate, or an error will occur. This is to prevent frame skipping. If you need to slow down the frame rate more than that, use Switch Mode.
- zone* (Optional) If specified, puts the filter into Switch Mode. Integer number greater or equal to zero. If zero, the filter will perform a hard switch, that is, it will immediately display the next frame below the switch line. If greater than zero, specifies the height (in lines) of the transition zone, where one frame is gradually blended into the next. *zone*=80 yields good results for full-size video (480/576 active lines). The transition is done in the same way as in PeculiarBlend(). *zone* must be less or equal than the number of lines of the target frame that correspond to the duration of the source frame. This is typically 5/6 or 6/5 of the target frame height, that is, a few hundred lines. An error occurs if a larger value is chosen.
- vbi* (Optional) In Switch Mode, specifies that the filter should apply a timing correction for the vertical blanking interval (VBI). Integer number greater than zero, indicating the height of the VBI of the target frames, in lines. Typically *vbi*=49 for PAL and *vbi*=45 for NTSC, but these values are not critical. Ignored in Blend Mode.

**Examples NTSC→PAL conversion:**

## Avisynth 2.5 Selected External Plugin Reference

```
AVISource("NTSC_clip.avi")           # Get clip
Bob(height=576)                       # Separate fields and interpolate them to full height.
BicubicResize(768,576)                # Resize to PAL square-pixel frame size. (Use 720,576 for
ConvertFPS(50)                         # Convert field rate to PAL, using Blend Mode.
SeparateFields.SelectEvery(4,0,3)     # Undo Bob, even field first. Use SelectEvery(4,1,2) for
Weave                                  # Finish undoing Bob.
```

This example will also work with frame-based NTSC material, even with telecined film (movies). For film material, however, you will get better results by using an inverse-telecine filter and speeding up the frame rate from 23.976 to 25fps.

Not all parameter values are checked for sanity.

### Changes:

v2.57	added preset option; changed framerate behaviour; YV12 and RGB support for ConvertFPS, fixed blending ratio
v2.56	added clip2 option in ChangeFPS, added AssumeScaledFPS
v2.55	added clip2 option in AssumeFPS
v2.54	added linear=true/false to ChangeFPS

\$Date: 2006/12/06 20:33:16 \$

## 14.6 FreezeFrame

```
FreezeFrame(clip, int first_frame, int last_frame, int source_frame)
```

The FreezeFrame filter replaces all the frames between *first\_frame* and *last\_frame* with a copy of *source\_frame*. The sound track is not modified. This is useful for covering up glitches in a video in cases where you have a similar glitch-free frame available.

\$Date: 2004/03/07 22:44:06 \$

## 14.7 GetChannel

```
GetChannel (clip, int ch1 [, int ch2, ...])
GetChannels (clip, int ch1 [, int ch2, ...])
```

Prior to v2.5 GetLeftChannel returns the left and GetRightChannel the right channel from a stereo signal. GetChannel is present starting from v2.5 and it returns one or more channels of a multichannel signal. GetChannels is an alias to GetChannel.

## Avisynth 2.5 Selected External Plugin Reference

The ordering of the channels is determined by the ordering of the input file, because AviSynth doesn't assume any ordering. In case of stereo 2.0 WAV and 5.1 WAV files the ordering should be as follows:

### WAV 2 ch (stereo):

1	left channel
2	right channel

### WAV 5.1 ch:

1	front left channel
2	front right channel
3	front center channel
4	LFE (Subwoofer)
5	rear left channel
6	rear right channel

```
# Removes right channel information, and return as mono clip with only left channel:
video = AviSource("c:\filename.avi")
stereo = WavSource("c:\afx-ab3_t4.wav")
mono = GetLeftChannel(stereo)
return AudioDub(video, mono)
```

```
# Using v2.5 this becomes:
video = AviSource("c:\filename.avi")
stereo = WavSource("c:\afx-ab3_t4.wav")
mono = GetChannel(stereo, 1)
return AudioDub(video, mono)
```

```
# You could also obtain the channels from the avi file itself:
video = AviSource("c:\filename.avi")
return GetChannel(video, 1)
```

```
# Converts avi with "uncompressed 5.1 wav" audio to a stereo signal:
video = AviSource("c:\divx_wav.avi")
audio = WavSource(c:\divx_wav.avi)
stereo = GetChannel(audio, 1, 2)
return AudioDub(video, stereo)
```

### 14.7.0.1 Remark1:

Every file format has a different channel ordering. The following table gives this ordering for some formats (useful for plugin writers :))

reference:	channel 1:	channel 2:	channel 3:	channel 4:	channel 5:	channel 6:
<a href="#">5.1 WAV</a>	front left channel	front right channel	front center channel	LFE	rear left channel	rear right channel
<a href="#">5.1 AC3</a>	front left channel	front center channel	front right channel	rear left channel	rear right channel	LFE
<a href="#">5.1 DTS</a>	front center	front left	front right	rear left	rear right	LFE

## Avisynth 2.5 Selected External Plugin Reference

	channel	channel	channel	channel	channel	
<a href="#">5.1 AAC</a>	front center channel	front left channel	front right channel	rear left channel	rear right channel	LFE
<a href="#">5.1 AIFF</a>	front left channel	rear left channel	front center channel	front right channel	rear right channel	LFE

\* 5.1 DTS: the LFE is on a separate stream (much like on multichannel MPEG2).

\* AAC specifications are unavailable on the internet (a free version)?

### 14.7.0.2 Remark2:

At the time of writing, Besweet still has the [2GB barrier](#). So make sure that the size of the 5.1 WAV is below 2GB, otherwise encode to six separate wavs or use HeadAC3he.

\$Date: 2006/10/24 19:47:56 \$

## 14.8 Greyscale

`Greyscale (clip [, string "matrix"])`

Converts the input clip to greyscale (without changing the color format).

In YCbCr based formats, the chroma channels are set to 128. In RGB based formats, the conversion produces the luma using the coefficients given in the *matrix* parameter (rec601 by default, which reflects the behaviour in old AviSynth versions). This option is added in v2.56. When setting *matrix*="rec709", the clip is converted to greyscale using Rec.709 coefficients. When setting *matrix*="Average" the luma is calculated as  $(R+G+B)/3$ . See [ColorConversions](#) for an explanation of these coefficients.

\$Date: 2008/02/10 13:57:17 \$

## 14.9 Histogram

`Histogram (clip, string "mode")`

Adds a luminance histogram to the right side of the clip.

Starting from AviSynth v2.50 this filter will also show valid and invalid colors in YUV mode. Invalid values (below 16 and above 235) will be colored brown/yellow-ish.

Starting in v2.53 an optional *mode* parameter has been added to show additional information of a clip. *mode* can be "Classic" (default old mode), "Levels", "Color", "Luma" (v2.54), "Stereo" (v2.54), "StereoOverlay" (v2.54), "Color2" (v2.58) and "AudioLevels" (v2.58).

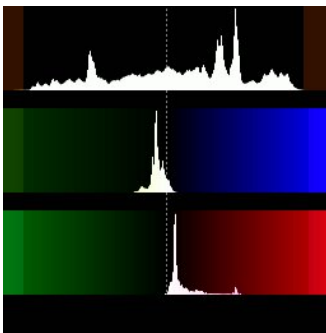
### 14.9.1 Classic mode



This will add a per-line luminance graph (which is actually called a Waveform Monitor) on the right side of the video. the left side of the graph represents luma = 0 and the right side represents luma = 255. The non-valid CCIR-601 ranges are shown in a brown/yellow-ish color, and a greenish line represents  $Y = 128$ .

Available in YUV mode.

### 14.9.2 Levels mode

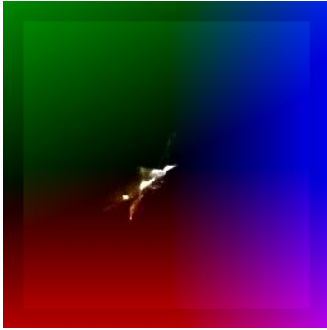


This mode will display three level-graphs on the right side of the video frame (which are called Histograms). This will show the distribution of the Y, U and V components in the current frame.

The top graph displays the luma (Y) distribution of the frame, where the left side represents  $Y = 0$  and the right side represents  $Y = 255$ . The valid CCIR601 range has been indicated by a slightly different color and  $Y = 128$  has been marked with a dotted line. The vertical axis shows the number of pixels for a given luma (Y) value. The middle graph displays the U component, and the bottom graph displays the V component.

Available in YV12 mode.

### 14.9.3 Color mode



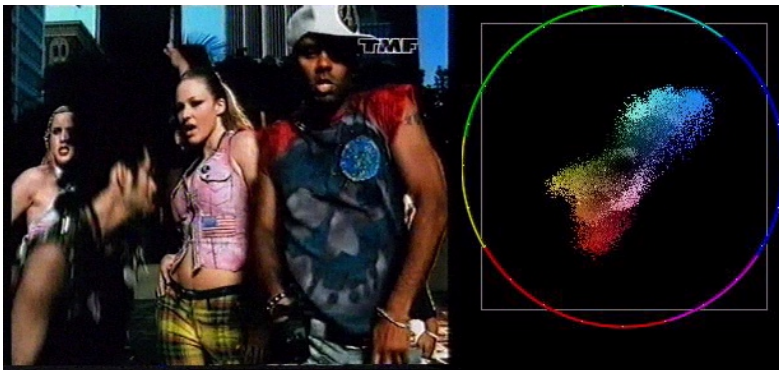
This mode will display the chroma values (U/V color placement) in a two dimensional graph (which is called a vectorscope) on the right side of the video frame. It can be used to read of the hue and saturation of a clip. At the same time it is a histogram. The whiter a pixel in the vectorscope, the more pixels of the input clip correspond to that pixel (that is the more pixels have this chroma value).

The U component is displayed on the horizontal (X) axis, with the leftmost side being  $U = 0$  and the rightmost side being  $U = 255$ . The V component is displayed on the vertical (Y) axis, with the top representing  $V = 0$  and the bottom representing  $V = 255$ .

The position of a white pixel in the graph corresponds to the chroma value of a pixel of the input clip. So the graph can be used to read of the hue (color flavor) and the saturation (the dominance of the hue in the color). As the hue of a color changes, it moves around the square. At the center of the square, the saturation is zero, which means that the corresponding pixel has no color. If you increase the amount of a specific color, while leaving the other colors unchanged, the saturation increases, and you move towards the edge of the square.

Available in YV12 mode.

#### 14.9.4 Color2 mode



This mode will display the pixels in a two dimensional graph (which is called a vectorscope) on the right side of the video frame. It can be used to read of the hue and saturation of a clip.

The U component is displayed on the horizontal (X) axis, with the leftmost side being  $U = 0$  and the rightmost side being  $U = 255$ . The V component is displayed on the vertical (Y) axis, with the top representing  $V = 0$  and the bottom representing  $V = 255$ . The grey square denotes the valid CCIR601 range.

The position of a pixel in the graph corresponds to the chroma value of a pixel of the input clip. So the graph can be used to read of the hue (color flavor) and the saturation (the dominance of the hue in the color). As the hue of a color changes, it moves around the circle. At the center of the circle, the saturation is zero, which

## Avisynth 2.5 Selected External Plugin Reference

means that the corresponding pixel has no color. If you increase the amount of a specific color, while leaving the other colors unchanged, the saturation increases, and you move towards the edge of the circle. A color wheel is plotted and divided into six hues (red, yellow, green, cyan, blue and magenta) to help you reading of the hue values. Also every 15 degrees a white dot is plotted.

At  $U=255, V=128$  the hue is zero (which corresponds to blue) and the saturation is maximal, that is,  $\text{sqrt}((U-128)^2 + (V-128)^2) = 127$ . When turning clock-wise, say 90 degrees, the chroma is given by  $U=128, V=255$  (which corresponds to red). Keeping the hue constant and decreasing the saturation, means that we move from the circle to the center of the vectorscope. Thus the color flavor remains the same (namely red), only it changes slowly to greyscale. Etc ...

Available in YV12 mode.

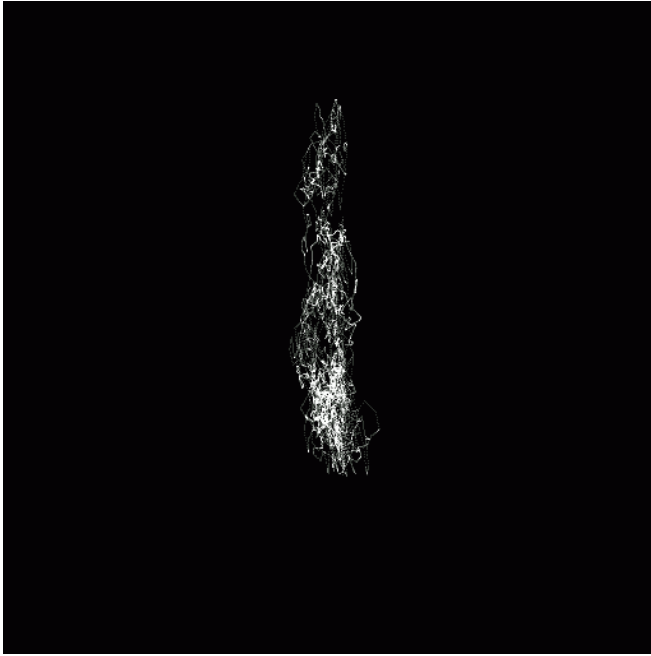
### 14.9.5 Luma mode



This mode will amplify luminance, and display very small luminance variations. This is good for detecting blocking and noise, and can be helpful at adjusting filter parameters. In this mode a 1 pixel luminance difference will show as a 16 pixel luminance pixel, thus seriously enhancing small flaws.

Available in YV12 mode.

### 14.9.6 Stereo and StereoOverlay mode



This mode shows a classic stereo graph, from the audio in the clip. Some may know these from recording studios. This can be used to see the left-right and phase distribution of the input signal. StereoOverlay will overlay the graph on top of the original. Each frame will contain only information from the current frame to the beginning of the next frame. The signal is linearly upsampled 8x, to provide clearer visuals.

This mode requires a stereo signal input and StereoOverlay requires YV12 video.

### 14.9.7 AudioLevels mode



This mode shows the audiolevels for each channel in decibels (multichannel is supported). More accurately it determines:

- the root mean square value of the samples belonging to each frame and converts this value to decibels using the following formula:

$$\text{RMS} = 20 * \log_{10}(\text{sqrt}(\text{sum}_j(\text{sample}(j)^2) / j) / 32768) \text{ \# for each channel}$$

- the maximum volume of the samples belonging to each frame and converts this value to decibels using the following formula:

$$\text{max} = 20 * \log_{10}(\text{max}_j(\text{sample}(j)) / 32768) \text{ \# for each channel}$$



## Avisynth 2.5 Selected External Plugin Reference

The bars corresponding to the root mean square value are green, and to the maximum are blue. The filter is available in planar mode and the audio is converted to 16 bit. Note that for 16 bit audio, the maximal volume could be

$$20 * \log_{10}(32768/32768) = 0 \text{ dB (since } 2^{16/2} = 32768)$$

and the minimal volume

$$20 * \log_{10}(1/32768) = - 90.31 \text{ dB}$$

### Changes:

v2.53	Added different modes.
v2.55	Added dots to mode = "stereo" to show bias/offsets.
v2.56	Added invalid colors in YUY2 mode.
v2.58	Color2 and AudioLevels modes added.
v2.60	Added planar support.

\$Date: 2007/09/12 07:54:58 \$

## 14.10 ImageReader / ImageSource

`ImageReader` (*string "file", int "start", int "end", float "fps", bool "use\_DeVIL", bool "info", string "pixel\_type"*)

`ImageSource` (*string "file", int "start", int "end", float "fps", bool "use\_DeVIL", bool "info", string "pixel\_type"*)

`ImageReader` is present in v2.52, it replaces WarpEnterprises' plugin, with some minor functionality changes. As of v2.55 `ImageSource` is equivalent, with some minor functionality changes. `ImageSource` is faster than `ImageReader` when importing one picture.

*file*: template for the image file(s), where frame number substitution can be specified using [sprintf syntax](#). For example, the files written by [ImageWriter](#)'s default parameters can be referenced with "c:\%06d.ebmp". As of v2.56 if the template points to a single file then that file is read once and subsequently returned for all requested frames.

*start* = 0, *end* = 1000: Specifies the starting and ending numbers used for filename generation. The file corresponding to *start* is always frame 0 in the clip, the file corresponding to *end* is frame (end-start). The resulting clip has (end-start+1) frames. '*end*=0' does NOT mean 'no upper bound' as with [ImageWriter](#). The first file in the sequence, i.e., corresponding to '*start*', MUST exist in order for clip parameters to be computed. Any missing files in the sequence are replaced with a blank frame.

*fps* = 24: frames per second of returned clip. An integer value prior to v2.55.

*use\_DeVIL* = false: When false, an attempt is made to parse (E)BMP files with the internal parser, upon failure DevIL processing is invoked. When true, execution skips directly to DevIL processing. You should only need to use this if you have BMP files you don't want read by `ImageReader`'s internal parser.

## Avisynth 2.5 Selected External Plugin Reference

**NOTE** : Devil version 1.6.6 as shipped with Avisynth does not correctly support DIB/BMP type files that use a palette, these include 8 bit RGB, Monochrome, RLE8 and RLE4. Because the failure is usually catastrophic, from revision 2.56, internal BMP processing does not automatically fail over to Devil processing. Forcing Devil processing for these file types is currently not recommended.

*info* = false: when true, the source filename text is overlaid on each video frame (added in v2.55).

*pixel\_type* = rgb24: Allow the output pixel format to be specified, both rgb24 and rgb32 are supported. The alpha channel is loaded only for rgb32 and only if Devil supports it for the loaded image format. (added in v2.56).

The resulting video clip colorspace is RGB if Devil is used, otherwise it is whatever colorspace an EBMP sequence was written from (all AviSynth formats are supported).

```
# Default parameters: read a 1000-frame native
# AviSynth EBMP sequence (at 24 fps)
ImageSource()

# Read files "100.jpeg" through "199.jpeg"
# into an NTSC clip.
ImageSource("D:\%d.jpeg", 100, 199, 29.97)
# Note: floating point fps is available from v2.56

# Read files "00.bmp" through "50.bmp" bypassing
# AviSynth's internal BMP reader.
ImageSource("D:\%02d.bmp", end=50, use_Devil=true)

# Read a single image, repeat 300 times
ImageSource("D:\static.png", end=300)
# Much, much faster from v2.56
```

\* "EBMP" is an AviSynth extension of the standard Microsoft RIFF image format that allows you to save raw YUY2 and YV12 image data. See [ImageWriter](#) for more details.

**\$Date: 2007/05/17 20:56:14 \$**

### 14.11 ImageWriter

`ImageWriter(clip, string "file", int "start", int "end", string "type", bool "info")`

`ImageWriter` (present in limited form in v2.51, full functionality in v2.52) writes frames from a clip as images to your harddisk.

*file* default "c:\": is the path + filename prefix of the saved images. The images have filenames such as: [path]000000.[type], [path]000001.[type], etc.

*start* and *end* are the start and end of the frame range that will be written. They both default to 0 (where "end"=0 means last frame). If *end* is negative (possible since v2.58), it specifies the number of frames that will be written.

*type* default "ebmp", is the filename extension and defines the format of the image.

The supported values for *type*, are:

(e)bmp, dds, ebmp, jpg/jpe/jpeg, pal, pcx, png,

## Avisynth 2.5 Selected External Plugin Reference

pbm/pgm/ppm, raw, sgi/bw/rgb/rgba, tga, tif/tiff

*info* default false: optionally overlay text progress information on the output video clip, showing whether a file is being written, and if so, the filename (added in v2.55).

Format "ebmp" supports all colorspace (RGB24, RGB32, YUY2, YV12). The "ebmp" files written from RGB colorspace are standard BMP files; those produced from YUV formats can probably only be read by AviSynth's [ImageReader/ImageSource](#). This pair of features allows you to save and reload raw video in any internal format.

For all other formats the input colorspace must be RGB24 or RGB32 (when the alpha channel is supported by the format and you want to include it).

### Examples:

```
# Export the entire clip in the current native AviSynth format
ImageWriter("D:\backup-stills\myvideo")

# Write frame 5 to "C:\000005.PNG"
ImageWriter("", 5, 5, "png")

# Write frames 100 till the end to "F:\pic-000100.JPEG", "F:\pic-000101.JPEG", etc.
# and display progress info
ImageWriter(file = "F:\pic", start = 100, type = "jpeg", info = true)
```

### Changelog:

v2.58	added end=-num_frames
-------	--------------------------

\$Date: 2008/06/06 16:36:46 \$

## 14.12 Import

`Import` (*string filename* [, ...])

`Import` evaluates the contents of other AviSynth scripts.

Functions, variables and loaded plugins declared inside the imported script are made available to the importing script. In addition, the return value of the function is the last value of the last filename script (a clip or whatever the filename script chooses to end with). The later can be assigned to a variable of the importing script and manipulated (this is most useful when the imported script ends with a clip).

The current working directory (CWD) is saved and set to the directory containing the script file before compiling the script. The CWD is then restored to the saved directory.

Some indicative uses of `Import` include:

- \* Storing multiple script-functions, variables and global variables for reuse by scripts (creation of script libraries).
- \* Retrieving pre-built streams.
- \* Retrieving dynamically configured pre-built streams (the core idea is: the importing script declares some

global variables which the imported script uses to configure the stream that will return).

\$Date: 2009/09/12 15:10:22 \$

## 14.13 Info

Info (*clip*)

Present in v2.5. It gives info of a clip printed in the left corner of the clip. The info consists of the duration, colorspace, size, fps, whether it is field (you applied [SeparateFields](#)) or frame based (you didn't apply [SeparateFields](#)), whether Avisynth thinks it is bottom (the default in case of [AviSource](#)) or top field first, whether there is audio present, the number of channels, sample type, number of samples and the samplerate. In v2.55 a CPU flag is added with supported optimizations.

Example:

```
AviSource("C:\filename.avi").Info
```

Results in a video with information in the top left corner:

```
Frame: 0 of 6035
Time: 00:00:00:000 of 00:04:01:400
ColorSpace: YUY2
Width: 720 pixels, Height: 576 pixels.
Frames per second: 25.0000 (25/1)
FieldBased (Separated) Video: NO
Parity: Bottom Field First
Video Pitch: 1440 bytes.
Has Audio: YES
Audio Channels: 2
Sample Type: Integer 16 bit
Samples Per Second: 44100
Audio length: 10650150 samples. 00:04:01:500
CPU detected: x87 MMX ISSE SSE 3DNOW 3DNOW_EXT
```

Changelog:

v2.57	Added time of current frame, total time, numerator and denominator of the framerate and audio length.
v2.55	Added supported CPU optimizations
v2.50	Initial Release

\$Date: 2009/09/12 15:10:22 \$

## 14.14 Interleave

Interleave (*clip1, clip2 [, ...]*)

Interleave interleaves frames from several clips on a frame-by-frame basis, so for example if you give three arguments, the first three frames of the output video are the first frames of the three source clips, the next three frames are the second frames of the source clips, and so on.

Also see [here](#) for the resulting clip properties.

\$Date: 2004/03/07 22:44:06 \$

## 14.15 Invert

`Invert (clip, string "channels")`

Inverts one or many color channels of a clip, available in v2.53.

### Parameters:

channels	Defines which channels should be inverted. The default is all channels of the current colorspace. Valid channels are R,G,B,A for RGB clips, and Y,U & V for YUY2 and YV12 clips.
----------	--

### Example:

```
AviSource("clip.avi")
ConvertToRGB32()
Invert("BG") # inverts the blue and green channels
```

### Changelog:

v2.53	Initial Release
v2.55	Added RGB24, YUY2 and YV12 mode.

\$Date: 2004/04/09 16:58:20 \$

## 14.16 KillAudio / KillVideo

`KillAudio (clip)`

`KillVideo (clip)`

Removes the audio or video from a clip completely. Can be used, if the destination does not accept an audio or video source, or if Avisynth crashes when processing audio or video from a clip.

\$Date: 2009/09/12 15:10:22 \$

## 14.17 Layer [yuy2][rgb32]

`Layer (base_clip, overlay_clip, string "op", int "level", int "x", int "y", int "threshold", bool "use_chroma")`

This filter can overlay two clips of different sizes (but with the same color format) using different operation modes.

For pixel-wise transparency information the 4th color channel of RGB32 (A- or alpha-channel) is used as a mask.

*Base\_clip*: the underlying clip which determines the size and all other video and audio properties of the result.

## Avisynth 2.5 Selected External Plugin Reference

*Overlay\_clip*: the clip which is merged onto clip. This clip can contain an alpha layer.

*op*: the performed merge operation, which can be: "add", "subtract", "lighten", "darken", "fast", "mul"

*level*: 0–257, the strength of the performed operation. 0: the base\_clip is returned unchanged, 257 (256 for YUY2): the maximal strength is used

*x, y*: offset position of the overlay\_clip

*threshold*: only implemented for "lighten" and "darken"

*use\_chroma*: use chroma of the overlay\_clip, default=true. When false only luma is used.

There are some differences in the behaviour and the allowed parameter depending on the color format and the operation, here are the details:

- There is no mask (alpha-channel) in YUY2, so the alpha-channel is assumed to be fully opaque everywhere.
- in RGB32 the alpha-channel of the *overlay\_clip* is multiplied with *level*, so the resulting alpha =  $(\text{alpha\_mask} * \text{level} + 1) / 256$ . This means for full strength of operation alpha has to be 255 and *level* has to be 257.

These operators behave equally for RGB32 or YUY2:

"fast": *use\_chroma* must be TRUE, *level* and *threshold* is not used.  
The result is simply the average of *base\_clip* and *overlay\_clip*.

"add": *threshold* is not used. The difference between *base\_clip* and *overlay\_clip* is multiplied with alpha and added to the *base\_clip*.  
alpha=0 -> only *base\_clip* visible,  
alpha=128 -> base and overlay equally blended,  
alpha=255 -> only overlay visible.  
Formula used :-  
RGB32 :: base += ((overlay-base)\*(alpha\*level+1)>>8)>>8  
YUY2 :: base += ((overlay-base)\*level)>>8

"subtract": the same as add, but the *overlay\_clip* is inverted before.

These operators seem to work correctly only in YUY2:

"mul": *threshold* is not used. The *base\_clip* is colored as *overlay\_clip*, so *use\_chroma* should be TRUE.  
alpha=0 -> only *base\_clip* visible, alpha=255 -> approx.  
the same Luminance as Base but with the colors of Overlay

"lighten": *use\_chroma* must be TRUE. Performs the same operation as "add", but only when the result is BRIGHTER than the base the new values are used. With ~~at threshold~~ *threshold* the operation is more likely, ~~with threshold=255~~ it's the same as "add", with *threshold*=0 the *base\_clip* is more likely passed unchanged, depending on the difference between *base\_clip* and *overlay\_clip*.

"darken": the same as "lighten", but it is performed only when the result is DARKER than the base.

Also see [here](#) for the resulting clip properties.

## 14.18 Mask [*rgb32*]

Mask (*clip*, *mask\_clip*)

Applies a defined alpha-mask to *clip*, for use with Layer, by converting *mask\_clip* to greyscale and using that for the mask (the alpha-channel) of RGB32. In this channel "black" means completely transparent, white means completely opaque).

For those of you who familiar with Photoshop masks, the concept is the same. In fact you can create a black and white photo in Photoshop, load it in your script and use it as a mask.

Here is an example; ss.jpg is derived from a snapshot from a video clip, which served as a guideline to create the mask just using Paint. We use Imagesource to load the image in the script and Mask to apply it.

```
bg = AviSource("0lgray.avi").ConvertToRGB32()           # here is the background clip
mk = Imagesource("ss.jpg").ConvertToRGB32()           # load the image
top = AviSource("k3.avi").ConvertToRGB32().Mask(mk)    # load the top layer clip and apply the
Layer(bg, top)                                         # layer the background and the top layer
```

## 14.19 ResetMask [*rgb32*]

ResetMask (*clip*)

Applies an "all-opaque" (that is white) alpha-mask to *clip*, for use with Layer.

The alpha-channel of a RGB32-clip is not always well-defined (depending on the source), this filter is the faster way to apply an all white mask:

```
clip = ResetMask(clip)
```

## 14.20 ColorKeyMask [*rgb32*]

ColorKeyMask (*clip*, *int color*[, *int tolB*, *int tolG*, *int tolR*])

Clears pixels in the alpha-channel by comparing the *color* (default black). Each pixel with a color differing less than (*tolB*, *tolR*, *tolG*) (default 10) is set to transparent (that is black), otherwise it is left unchanged i.e. It is NOT set to opaque (that it is not set to white, that's why you might need ResetMask before applying this filter), this allows a aggregate mask to be constructed with multiple calls. When *tolR* or *tolG* are not set, they use the value from *tolB* (which reflects the old behaviour). Normally you start with a ResetMask, then chain a few calls to ColorKeyMask to cause transparent holes where each color of interest occurs. See [Overlay](#) for examples.

For Avisynth versions older than v2.58, there were no separate tolerance levels for blue, green and red. There was only one tolerance level called *tolerance* and was used for blue, green and red simultaneously.

\$Date: 2009/09/12 15:10:22 \$

## 14.21 Letterbox

`Letterbox (clip, int top, int bottom, [int left], [int right], int "color")`

Letterbox simply blackens out the top *top* and the bottom *bottom*, and optionally *left* and *right* side of each frame. This has a couple of uses: one, it can eliminate stray video noise from the existing black bands in an image that's already letterboxed; two, it can eliminate the garbage lines that often appear at the bottom of the frame in captures from VHS tape.

The functionality of Letterbox can be duplicated with a combination of [Crop](#) and [AddBorders](#), but Letterbox is faster and easier.

Generally, it's better to crop this stuff off using Crop or [CropBottom](#) than to hide it with Letterbox. However, in some cases, particularly if you're compressing to MPEG, it's better to use Letterbox because it lets you keep a standard frame size like 352x288 or 320x240. Some MPEG players get confused when the source video has a strange frame size.

The color parameter is optional, default=0 <black>, and is specified as an RGB value regardless of whether the clip format is RGB or YUV based. See [here](#) for more information on specifying colors.

Another use could also be to clear out overscan areas in VCD or SVCD encodings.

\$Date: 2008/06/06 11:37:04 \$

## 14.22 Levels

`Levels (clip, int input_low, float gamma, int input_high, int output_low, int output_high, bool "coring")`

The Levels filter adjusts brightness, contrast, and gamma (which must be > 0). The *input\_low* and *input\_high* parameters determine what input pixel values are treated as pure black and pure white, the *output\_low* and *output\_high* parameters determine the output values corresponding to pure black and white and the *gamma* parameter controls the degree of nonlinearity in the conversion. To be precise, the conversion function is:

$$\text{output} = [(\text{input} - \text{input\_low}) / (\text{input\_high} - \text{input\_low})]^{1/\text{gamma}} (\text{output\_high} - \text{output\_low}) + \text{output\_low}$$

This is one of those filters for which it would really be nice to have a GUI. Since I can't offer a GUI (at least not in AviSynth's current form), I decided I could at least make this filter compatible with VirtualDub's when the clip is RGB. In that case you should be able to take the numbers from VirtualDub's Levels dialog and pass them as parameters to the Levels filter and get the same results. However, the input and output parameters can be larger than 255.

When processing data in YUV mode, Levels only gamma-corrects the luma information, not the chroma. Gamma correction is really an RGB concept, and I don't know how to do it properly in YUV. However, if *gamma* = 1.0, the filter should have the same effect in RGB and YUV modes. For adjusting brightness or contrast it is better to use [Tweak](#) or [ColorYUV](#), because Levels also changes the chroma of the clip.



## Avisynth 2.5 Selected External Plugin Reference

In v2.53 an optional *coring* = *true/false* (true by default, which reflects the behaviour in older versions) is added.

*coring* = true: input luma is clamped to [16,235] (and the chroma to [16,240]), result is *\*scaled\** from [16,235] to [0,255], the conversion takes place according to the formula above, and output is *\*scaled\** back from [0,255] to [16,235].

*coring* = false: conversion takes place according to the formula above.

```
# does nothing on a [16,235] clip, but it clamps (or rounds) a [0,255] clip to [16,235]:
Levels(0, 1, 255, 0, 255)
```

```
# the input is scaled from [16,235] to [0,255], the conversion [0,255]->[16,235] takes place (a
# and the output is scaled back from [0,255] to [16,235]: (for example: the luma values in [0,1
Levels(0, 1, 255, 16, 235)
```

```
# gamma-correct image for display in a brighter environment:
# example: luma of 16 stays 16, 59 is converted to 79, etc.
Levels(0, 1.3, 255, 0, 255)
```

```
# invert the image (make a photo-negative):
# example: luma of 16 is converted to 235
Levels(0, 1, 255, 255, 0)
```

```
# does nothing on a [0,255] clip; does nothing on a [16,235]:
Levels(0, 1, 255, 0, 255, coring=false)
```

```
# scales a [0,255] clip to [16,235]:
Levels(0, 1, 255, 16, 235, coring=false) # this is the same as ColorYUV(levels="PC->TV")
```

```
# scales a [16,235] clip to [0,255]:
Levels(16, 1, 235, 0, 255, coring=false) # this is the same as ColorYUV(levels="TV->PC")
```

**\$Date: 2005/11/15 21:23:11 \$**

### 14.23 Limiter

Limiter (*clip*, *int "min\_luma"*, *int "max\_luma"*, *int "min\_chroma"*, *int "max\_chroma"*, *string "show"*)

This filter is present in v2.5. The standard known as CCIR-601 defines the range of pixel values considered legal for presenting on a TV. These ranges are 16-235 for the luma component and 16-240 for the chroma component.

Pixels outside this range are known to cause problems with some TV sets, and thus it is best to remove them before encoding if that is your intended display device. By default this filter clips (or "clamps") pixels under 16 to 16 and over 235 (or 240) to 235 (or 240).

Prior to v2.53 the (incorrect) default value was 236. Use Limiter(16, 235, 16, 240) for CCIR-601 compliant digital video.

In v2.56, an option *show* is added. If set, it colors the pixels outside the specified [min\_luma,max\_luma] or [min\_chroma,max\_chroma] range.

*show* can be "luma" (shows out of bounds luma in red/green), "luma\_grey" (shows out of bounds luma and makes the remaining pixels greyscale), "chroma" (shows out of bounds chroma in yellow), "chroma\_grey" (shows out of bounds chroma and makes the remaining pixels greyscale). The coloring is done as follows:

## Avisynth 2.5 Selected External Plugin Reference

YUY2 (chroma shared between two horizontal pixels p1 and p2: Y1UY2V):

YV12 (chroma shared between 2x2 pixels Y11uY12v;/Y21uY22v):

<b>j,k=1,2 or j,k=11,12,21,22</b>	<b>luma</b>	<b>luma_grey</b>
Yj < min_luma	red (pj)	red (pj)
Yj > max_luma	green (pj)	green (pj)
Yj < min_luma and Yk > max_luma	yellow (all)	puke (pj), olive (pk)
	<b>chroma</b>	<b>chroma_grey</b>
U < min_chroma	yellow	yellow
U > max_chroma	yellow	blue
V < min_chroma	yellow	cyan
V > max_chroma	yellow	red
U < min_chroma and V < min_chroma	yellow	green
U > max_chroma and V < min_chroma	yellow	teal
U < min_chroma and V > max_chroma	yellow	orange
U > max_chroma and V > max_chroma	yellow	magenta

### Changelog:

v2.56	added show to show out of bounds luma/chroma
-------	--

⌘Date: 2005/06/01 17:42:26 ⌘

## 14.24 Loop

Loop (*clip*, *int "times"*, *int "start"*, *int "end"*)

Loops the segment from frame *start* to frame *end* a given number of *times*.

*times* (default -1) is the number of times the loop is applied.

*start* (default 0) the frame of the clip where the loop starts.

*end* (default framecount(*clip*)) the frame of the clip where the loop ends.

```
# Loops frame 100 to 110 of the current clip 10 times
Loop(10,100,110)
```

```
Loop() # make the clip loop (almost) endlessly
```

```
Loop(10) # make the clip loop ten times
```

## Avisynth 2.5 Selected External Plugin Reference

```
Loop(10,20,29) # repeat frames 20 to 29 ten times before going on

# actual clip duration increased by 90 frames
Loop(0,20,29) # delete frames 20 to 29

# actual clip duration decreased by 10 frames
Loop(-1,20,29) # frame 20 to 29 is repeated (almost) infinite times

$Date: 2004/03/07 22:44:06 $
```

### 14.25 MaskHS

`MaskHS` (*clip*, *float "startHue"*, *float "endHue"*, *float "maxSat"*, *float "minSat"*, *bool "coring"*)

Added in v2.6. This filter returns a mask (as Y8) of clip using a given hue and saturation range.

*startHue* (default 0), *endHue* (default 360): (both from 0 to 360; given in degrees.). The hue and saturation will be adjusted for values in the range [startHue, endHue] when startHue<endHue. Note that the hue is periodic, thus a hue of 360 degrees corresponds with a hue of zero degrees. If endHue<startHue then the range [endHue, 360] and [0, startHue] will be selected (thus anti-clockwise). If you need to select a range of [350, 370] for example, you need to specify startHue=370 and endHue=350. Thus when using the default values all pixels will be processed.

*maxSat* (default 150), *minSat* (default 0): (both from 0 to 150 with minSat<maxSat; given in percentages). The hue and saturation will be adjusted for values in the range [minSat, maxSat]. Practically the saturation of a pixel will be in the range [0,100] (thus 0–100%), since these correspond to valid RGB pixels (100% corresponds to R=255, G=B=0, which has a saturation of 119). An overshoot (up to 150%) is allowed for non-valid RGB pixels (150% corresponds to U=V=255, which has a saturation of  $\sqrt{127^2+127^2} = 180$ ). Thus when using the default values all pixels will be processed.

*coring* = true/false (default false). When set to true, the luma (Y) is clipped to [16,235]; when set to false, the luma is left untouched.

Suppose we want to create a mask of the skin of the girl. The proper way to do this is to use look at the vectorscope of [Histogram](#):

```
clip = ...
Histogram(clip, mode="color2")
```

and estimate the hue range you want to select. As can be seen, the orange hue is between (about) 105 and 165 degrees.

lower the hue range till you found the correct hue range which should be processed. Use the values in `MaskHS` and make the interval smaller till the correct one is selected. You can also use [Tweak](#) for this (with sat=0). Using the example in `Tweak`, the following mask is obtained:

	
original	MaskHS(startHue=105, endHue=138)

Looking at the blue screen example in [Overlay](#) the following can be used

```
testcard = ColorBars()

# example subtitle file with blue background:
subs = ImageSource("F:\TestClips\blue.jpg").ConvertToYV24

# subs.Histogram(mode="color2").ConvertToRGB # blue in [345,359]
mask_hs = subs.MaskHS(startHue=340, endHue=359).Levels(0, 1, 255, 255, 0)

Overlay(testcard, subs, mask=mask_hs, mode="blend", opacity=1)
```

**Changelog:**

v2.60	Initial Release
-------	-----------------

\$Date: 2009/09/12 15:10:22 \$

## 14.26 Merge / MergeChroma / MergeLuma

Merge (*clip1, clip2, float weight=0.5*)  
 MergeChroma (*clip1, clip2, float weight=1.0*)  
 MergeLuma (*clip1, clip2, float weight=1.0*)

These filters make it possible to merge luma or chroma or both from a videoclip into another. There is an optional weighting, so a percentage between the two clips can be specified. Merge is present in v2.56.

*clip1* is the clip that has Luma or Chroma merged INTO (based on which filter you use), that means that the OTHER channel (chroma if MergeLuma is used, luma in MergeChroma) is left completely untouched.

*clip2* is the one from which the Luma or Chroma must be taken. In MergeChroma, this is where the Chroma will be taken from, and vice-versa for MergeLuma. It must be the same colorspace as clip1; i.e. you cannot merge from a YV12 clip into a YUY2 clip.

## Avisynth 2.5 Selected External Plugin Reference

The *weight* defines how much influence the new clip should have. Range is 0.0 to 1.0, where 0.0 is no influence and 1.0 will completely overwrite the specified channel (default). The filter will be slightly slower when a weight other than 0.0, 0.5 or 1.0 is specified.

Also see [here](#) for the resulting clip properties.

```
# Will only blur the Luma channel.
mpeg2source("c:\apps\avisynth\main.d2v")
lumavid = Blur(1.0)
MergeLuma(lumavid)

# This will do a Spatial Smooth on the Chroma channel
# that will be mixed 50/50 with the original image.
mpeg2source("c:\apps\avisynth\main.d2v")
chromavid = SpatialSmoother(2,3)
MergeChroma(chromavid,0.5)

# This will run a temporalsmoother and a soft spatial
# smoother on the Luma channel, and a more aggressive
# spatial smoother on the Chroma channel.
# The original luma channel is then added with the
# smoothed version at 75%. The chroma channel is
# fully replaced with the blurred version.
mpeg2source("c:\apps\avisynth\main.d2v")
luma = TemporalSmoother(2,3)
luma = luma.SpatialSmoother(3,10,10)
chroma = SpatialSmoother(3,40,40)
MergeLuma(luma,0.75)
MergeChroma(chroma)

# This will average two video sources.
avisource("c:\apps\avisynth\main.avi")
vid2 = avisource("c:\apps\avisynth\main2.avi")
Merge(vid2)
```

### Changelog:

v2.56	added Merge
-------	----------------

**\$Date: 2008/10/26 14:18:27 \$**

## 14.27 MergeChannels

MergeChannels (*clip1* , *clip2* [, *clip3*, ...])

Starting from v2.5 MergeChannels replaces [MonoToStereo](#), and can be used to merge the audio channels of two or more clips.

Don't confuse this with mixing of channels ([MixAudio](#) and [ConvertToMono](#) do this) – the sound of each channel is left untouched, the channels are only put into the new clip.

Before merging audio is converted to the sample type of *clip1*.

```
# Example, converts "uncompressed wav" audio to a 44.1 kHz stereo signal:
```

## Avisynth 2.5 Selected External Plugin Reference

```
video = AviSource("c:\divx_wav.avi")
audio = WavSource("c:\divx_wav.avi")
l_ch = GetChannel(audio, 1)
r_ch = GetChannel(audio, 2)
stereo = MergeChannels(l_ch, r_ch).ResampleAudio(44100)
return AudioDub(video, stereo)
```

```
# This is similar to:
video = AviSource("c:\divx_wav.avi")
audio = WavSource("c:\divx_wav.avi")
stereo = GetChannel(audio, 1, 2).ResampleAudio(44100)
return AudioDub(video, stereo)
```

**\$Date: 2004/03/09 21:28:07 \$**

### 14.28 MergeARGB / MergeRGB

MergeARGB (*clipA*, *clipR*, *clipG*, *clipB*)  
MergeRGB (*clipR*, *clipG*, *clipB* [, *string "pixel\_type"*])

These filters makes it possible to merge the alpha and color channels from the source video clips into the output video clip.

*ClipA* is the clip that provided the alpha data to merge into the output clip. For a YUV format input clip the data is taken from the Luma channel. For a RGB32 format input clip the data is taken from the Alpha channel. It may not be in RGB24 format.

*ClipR*, *ClipG* and *ClipB* are the clips that provided the R, G and B data respectively to merge into the output clip. For YUV format input clips the data is taken from the Luma channel. For RGB format input clips the data is taken from the respective source channel. i.e. R to R, G to G, B to B. The unused chroma or color channels of the input clips are ignored.

All YUV luma pixel data is assumed to be pc-range, [0..255], there is no tv-range, [16..235], scaling. Chroma data from YUV clips is ignored. Input clips may be a mixture of all formats. YV12 is the most efficient format for transporting single channels thru any required filter chains.

*pixel\_type* default RGB32, optionally RGB24, specifies the output pixel format.

Also see [here](#) for the resulting clip properties.

#### Examples:

```
# This will only blur the Green channel.
mpeg2source("c:\apps\avisynth\main.d2v")
ConvertToRGB24()
MergeRGB>Last, Blur(0.5), Last)

# This will swap the red and blue channels and
# load the alpha from a second video sources.
vid1 = avisource("c:\apps\avisynth\main.avi")
vid2 = avisource("c:\apps\avisynth\alpha.avi")
MergeARGB(vid2, vid1.ShowBlue("YV12"), vid1, vid1.ShowRed("YV12"))
AudioDub(vid1)
```

**Changelog:**

v2.56	added MergeARGB and MergeRGB
-------	------------------------------

\$Date: 2008/05/28 21:24:49 \$

**14.29 MessageClip**

`MessageClip` (*string message, int "width", int "height", bool "shrink", int "text\_color", int "halo\_color", int "bg\_color"*)

`MessageClip` produces a clip containing a text message; used internally for error reporting. Arial font is used, size between 24 points and 9 points chosen to fit, if possible, in the *width* by *height* clip. If *shrink* is true, the clip resolution is then reduced, if necessary, to fit the text.

\$Date: 2004/03/09 21:28:07 \$

**14.30 MixAudio**

`MixAudio` (*clip1, clip2, float "clip1\_factor", float "clip2\_factor"*)

Mixes audio from two clips. A volume for the two clips can be given, but is optional.

Volume is given as a factor, where 0.0 is no audio from the desired channel, and 1.0 is 100% from the desired channel. Default is 0.5/0.5 – if only one factor is given, the other channel will be 1.0–(factor). If factor1 + factor2 is more than 1.0, you risk clipping your signal.

The two clips must have audio at the same sample rates (use [ResampleAudio](#) if this is a problem). Your clips needs to have the same number of channels (stereo/mono) – use [ConvertToMono](#) or [MergeChannels](#) if this is a problem.

```
# Mixes two sources, with one source slightly lower than the other.
Soundtrack = WavSource("c:\soundtrack.wav")
Speak = WavSource("c:\speak.wav")
return MixAudio(Soundtrack, Speak, 0.75, 0.25)    # The Expert may notice that the last 0.25 is
```

\$Date: 2004/03/09 21:28:07 \$

**14.31 MonoToStereo**

`MonoToStereo` (*left\_channel\_clip, right\_channel\_clip*)

Converts two mono signals to one stereo signal. This can be used, if one or both channels has been modified seperately, and then has to be recombined.

## Avisynth 2.5 Selected External Plugin Reference

The two clips has to have audio at the same sample rates (Use [ResampleAudio](#) if this is a problem.) If either of the sources is in stereo, the signal will be taken from the corresponding channel (left channel from the *left\_channel\_clip*, and vice versa for right channel).

Before merging audio is converted to the sample type of the *left\_channel\_clip*.

In v2.5 this function is simply mapped to [MergeChannels](#).

```
#Combines two separate wave sources to a stereo signal:
left_channel = WavSource("c:\left_channel.wav")
right_channel = WavSource("c:\right_channel.wav")
return MonoToStereo(left_channel, right_channel)
```

**\$Date: 2004/03/09 21:28:07 \$**

### 14.32 Normalize

Normalize (*clip*, *float "volume"*, *bool "show"*)

Amplifies the entire waveform as much as possible, without clipping.

By default the clip is amplified to 1.0, that is maximum without clipping – higher values are sure to clip, and create distortions. If one volume is supplied, the other channel will be amplified the same amount.

The calculation of the peak value is done the first time the audio is requested, so there will be some seconds until AviSynth continues.

Starting from v2.08 there is an optional argument `show`, if set to `true`, it will show the maximum amplification possible without distortions.

Multichannels are never amplified separately by the filter, even if the level between them is very different. The volume is applied AFTER the maximum peak has been found, and works in effect as a separate [Amplify](#). That means if you have two channels that are very different the loudest channel will also be the peak for the lowest. If you want to normalize each channel separately, you must use [GetChannel](#) to split up the stereo source.

The audio sample type is converted to float or is left untouched if it is 16 bits.

Examples:

```
# normalizes signal to 98%
video = AviSource("C:\video.avi")
audio = WavSource("c:\autechre.wav")
audio = Normalize(audio, 0.98)
return AudioDub(video, audio)

# normalizes each channel separately
video = AviSource("C:\video.avi")
audio = WavSource("C:\bjoer7000.wav")
left_ch = GetLeftChannel(audio).Normalize
right_ch = GetRightChannel(audio).Normalize
audio = MonoToStereo(left_ch, right_ch)
```



## Avisynth 2.5 Selected External Plugin Reference

```
return AudioDub(video, audio)

# normalizes each channel seperately
clip = AviSource("D:\Video\rawstuff\stereo-test file_left(-6db).avi")
left_ch = GetChannel(clip,1).Normalize
right_ch = GetChannel(clip,2).Normalize
audio = MergeChannels(left_ch, right_ch)
AudioDub(clip, audio)
```

**\$Date: 2009/09/12 15:10:22 \$**

### 14.33 Overlay

*Overlay (clip, clip overlay, int "x", int "y", clip "mask", float "opacity", string "mode", bool "greymask", string "output", bool "ignore\_conditional", bool "pc\_range")*

Overlay puts two clips on top of each other with an optional displacement of the overlaying image, and using different overlay methods. Furthermore opacity can be adjusted for the overlay clip.

Input for overlay is any colorspace, and colorspaces of different clip doesn't matter! The input clips are internally converted to a general YUV (with no chroma subsampling) format, so it is possible for the filter to output another colorspace than the input. It is also possible to input video in different colorspaces, as they will be converted seamlessly. It is however not recommended to use overlay "only" for colorspace conversions, as it is both slower and with slightly worse quality.

In general all clips are treated as 0->255 values. This means that numbers will not be clipped at CCIR 601 range. Use [Limiter](#) for this task afterwards.

**Masks should also have values from 0->255.** You can use [Histogram](#) in Histogram("levels") mode to view the color distributions. If your mask is in CCIR 601 range, use [ColorYUV](#)(levels="TV->PC") to upscale the color levels.

It is not recommended to do overlays on interlaced material, unless you know what you are doing.

#### 14.33.0.1 Parameters:

##### **clip**

This clip will be the base, and the overlay picture will be placed on top of this.

##### **overlay**

This is the image that will be placed on top of the base clip. The colorspace or image dimensions do not have to match the base clip.

##### **x & y**

These two variables define the placement of the overlay image on the base clip in pixels. The variable can be positive or negative.

*Default values are 0.*

##### **mask**

This will be used as the transparency mask for the overlay image. The mask must be the same size as the overlay clip. By default only the greyscale (luma) components are used from the image. The darker the image is, the more transparent will the overlay image be.

*There is no default, but not specifying is equivalent to supplying a 255 clip.*

## Avisynth 2.5 Selected External Plugin Reference

### opacity

This will set how transparent your image will be. The value is from 0.0 to 1.0, where 0.0 is transparent and 1.0 is fully opaque (if no mask is used). When used together with a mask this value is multiplied by the mask value to form the final opacity.

*Default value is 1.0*

### mode

Mode defines how your clip should be overlaid on your image.

Mode	Description
Blend	This is the default mode. When opacity is 1.0 and there is no mask the overlay image will be copied on top of the original. Ordinary transparent blending is used otherwise.
Add	This will add the overlay video to the base video, making the video brighter. To make this as comparable to RGB, overbright luma areas are influencing chroma and making them more white.
Subtract	The opposite of Add. This will make the areas darker.
Multiply	This will also darken the image, but it works different than subtract.
Chroma	This will only overlay the color information of the overlay clip on to the base image.
Luma	This will only overlay the luminosity information of the overlay clip on to the base image.
Lighten	This will copy the light information from the overlay clip to the base clip, only if the overlay is lighter than the base image.
Darken	This will copy the light information from the overlay clip to the base clip, only if the overlay is darker than the base image.
SoftLight	This will lighten or darken the base clip, based on the light level of the overlay clip. If the overlay is darker than luma = 128, the base image will be darker. If the overlay is lighter than luma=128, the base image will be lighter. This is useful for adding shadows to an image. Painting with pure black or white produces a distinctly darker or lighter area but does not result in pure black or white.
HardLight	This will lighten or darken the base clip, based on the light level of the overlay clip. If the overlay is darker than luma = 128, the base image will be darker. If the overlay is lighter than luma=128, the base image will be lighter. This is useful for adding shadows to an image. Painting with pure black or white results in pure black or white.
Difference	This will display the difference between the clip and the overlay. Note that like <a href="#">Subtract</a> a difference of zero is displayed as grey, but with luma=128 instead of 126. If you want the pure difference, use mode="Subtract" or add <a href="#">ColorYUV(off_y=-128)</a> .
Exclusion	This will invert the image based on the luminosity of the overlay image. Blending with white inverts the base color values; blending with black produces no change.

*Default value is Blend*

### greymask

This option specifies whether chroma from the mask should be used for chroma transparency. For general purpose this mode shouldn't be disabled. External filters like mSharpen and Masktools are able to export proper chroma maps.

*Default value is true*

### output

It is possible to make Overlay return another colorspace. Possible output colorspace are "YV12", "RGB32" and "RGB24".

*Default is input colorpace*

### **ignore\_conditional**

This will make Overlay ignore any given conditional variables. See the "Conditional Variables" section for an overview over conditional variables.

*Default is false*

### **pc\_range**

When set to true, this will make all internal RGB → YUV → RGB conversions assume that luma range is 0 to 255 instead of default 16→235 range. It is only recommended to change this setting if you know what you are doing. See the section on "RGB considerations" below.

*Default is false*

## **14.33.1 RGB considerations**

This section will describe things that may give you an explanation of why Overlay behaves like it does when it is given one or more RGB sources.

One or more inputs for Overlay are allowed to be RGB-data. However, as Overlay is processing material in the YUV colorspace this will lead to an RGB to YUV conversion. There are two modes for this conversion, toggled by the "pc\_range" parameter. This parameter will extend the YUV range from 16–235 (this is the range used by all avisynth converters) to 0–255. There are some cases where enabling pc\_range is a good idea:

- When overlaying an RGB-clip using the "add", "subtract" or "multiply" modes, the range of the overlay clip is better, if it is 0–255, since this will enable completely dark areas not to influence the result (instead of adding 16 to every value).
- When NOT doing a colorspace conversion on output. If the output colorspace (RGB vs. YUV) is different from the input, the scale will be wrong. If pc\_range is true, and input is RGB, while output is YUY2 – the YUY2 will have an invalid range, and not CCIR 601 range.

### **Outputting RGB**

It might be a good idea to let Overlay output YUY2, even if your input colorspace is RGB, as this avoids a colorspace conversion back to RGB from YUV. You should however be aware that your material might be "overscaled", as mentioned above, if you use pc\_range = true. You can correct this by using "ColorYUV(levels="pc→tv")" to convert back to 16–235 range.

### **Inputting RGB for mask clip**

The mask clip from RGB may behave a bit different than it could be expected. If you always use a greyscale mask, and don't disable "greymask" you will get the result you'd expect. You should note that mask clip values are never scaled, so it will automatically be in 0→255 range, directly copied from the RGB values.

### *Using RGB32 alpha channel*

Overlay will never use the alpha channel given in an RGB32 clip. If you want to extract the alpha channel from an RGB32 clip you can use the [ShowAlpha](#) command to extract the alpha information. For maintaining maximum quality it is recommended to extract the alpha as RGB.

## **14.33.2 Conditional Variables**

The global variables "OL\_opacity\_offset", "OL\_x\_offset" and "OL\_y\_offset" are read each frame, and applied.

## Avisynth 2.5 Selected External Plugin Reference

It is possible to modify these variables using [FrameEvaluate](#). The values of these variables will be added to the original on each frame. So if you specify "x = 100" as a filter parameter, and the global variable "OL\_x\_offset" is set to 50, the overlay will be placed at x = 150.

If you are using multiple filters this can be disabled by using the "ignore\_conditional = true" parameter.

There is an example of conditional modification at the [ConditionalReader](#) page.

### 14.33.3 Examples

# Prepares some sources.

```
bg = colorbars(512,384).converttoyuy2()
text = blankclip(bg).subtitle("Colorbars", size=92, text_color=$ffffff).coloryuv(levels="tv->pc")
```

# This will overlay the text in three different versions.

```
overlay(bg, text, x=50, y=20, mode="subtract", opacity=0.25)
overlay(text, x=50, y=120, mode="add", opacity=0.5)
overlay(text, x=50, y=240, mode="blend", opacity=0.7)
```

# This will overlay yuy2clip with rgbclip using a yuy2-mask (note that the luma range of the mask is [0-255]).

```
Overlay(yuy2clip, rgbclip, mask = rgbclip.ShowAlpha("yuy2"))
```

# which is the same as

```
mask = rgbclip.ShowAlpha("rgb").ConvertToYUY2.ColorYUV(levels="TV->PC")
Overlay(yuy2clip, rgbclip, mask)
```

# which is the same as

```
mask = rgbclip.ShowAlpha("rgb")
Overlay(yuy2clip, rgbclip, mask)
```

# This will take the average of two clips. It can be used for example to combine two captures of different broadcastings for reducing noise. A discussion of this idea can be found [\[here\]](#). A sample script (of course you have to ensure that the frames of the two clips matches exactly, use [DeleteFrame](#) if necessary):

```
clip1 = AviSource("F:\shakira-underneath_your_clothes1.avi")
clip2 = AviSource("F:\shakira-underneath_your_clothes2.avi")
Overlay(clip1, clip2, mode="blend", opacity=0.5)
```

# Use a blue (or any other color) background (blue.jpg is a blue frame overlaid with subtitles in a black rectangle) as mask. The black rectangle containing the subtitles will be visible on the source clip (which is ColorBars here):

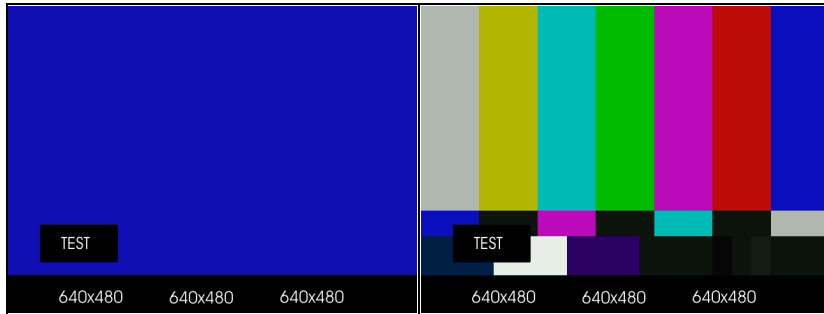
```
testcard = ColorBars()

# get a blue mask clip (the same blue as in ColorBars is used: R16 G16 B180)
maskclip = BlankClip(testcard, color=$0f0fb4)

# Example subtitle file with blue background as above
```

## Avisynth 2.5 Selected External Plugin Reference

```
subs = ImageSource("F:\TestClips\blue.jpg").ConvertToRGB32
maskclip = ColorKeyMask(subs, $0f0fb4, 60)
Overlay(testcard, subs, mask=ShowAlpha(maskclip), mode="blend", opacity=1)
```



A tolerance of 60 is used here because the blue is not entirely uniform. Near the black rectangles the blue is given by R23 G22 B124. Probably due to the compression of blue.jpg.

# Move a red (or any other color) dot on a clip using ConditionalReader (dot.bmp is a red dot on a black background):

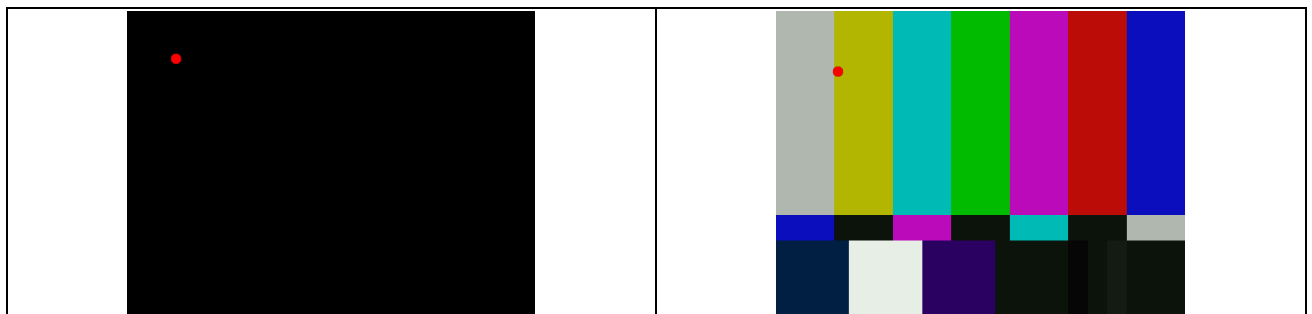
```
a1 = ColorBars().Trim(0,399)
a2 = ImageSource("F:\TestClips\dot.bmp").ConvertToRGB32

# a2.GreyScale returns a grey dot on a black background; Levels makes the dot white
mask_clip = Mask(a2, a2.GreyScale.Levels(0, 1, 75, 0, 255))
Overlay(a1, a2, mask=ShowAlpha(mask_clip), y=0, x=0, mode="blend", opacity=1)

ConditionalReader("xoffset.txt", "ol_x_offset", false)
ConditionalReader("yoffset.txt", "ol_y_offset", false)
```

Make xoffset.txt containing the x-positions and yoffset.txt containing the y-positions of the moving dot (see [ConditionalReader](#) for more info), and put it in the same folder as your script:

xoffset.txt	yoffset.txt
Type int	Type int
Default -50	Default -50
R 0 100 20	R 0 100 20
I 100 200 20 250	I 100 200 20 350
R 200 300 250	R 200 300 350
I 300 400 250 400	I 300 400 350 40



thus the dot moves in the following way: (20,20) → (250,350) → (400,40). Nb, it's also possible to do this with `Animate`.

### 14.33.4 Changelog:

v2.54	Initial Release
-------	-----------------

\$Date: 2009/09/12 15:10:22 \$

## 14.34 AssumeFrameBased / AssumeFieldBased

`AssumeFrameBased` (*clip*)  
`AssumeFieldBased` (*clip*)

AviSynth keeps track of whether a given clip is field-based or frame-based. If the clip is field-based it also keeps track of the parity of each field (that is, whether it's the top or the bottom field of a frame). If the clip is frame-based it keeps track of the dominant field in each frame (that is, which field in the frame comes first when they're separated).

However, this information isn't necessarily correct, because field information usually isn't stored in video files and Avisynth's source filters just guess at it. `AssumeFrameBased` and `AssumeFieldBased` let you tell AviSynth the correct type of a clip.

`AssumeFrameBased` throws away the existing information and assumes that the clip is frame-based, with the bottom (even) field dominant in each frame. (This happens to be what the source filters guess.) If you want the top field dominant, use `ComplementParity` afterwards.

`AssumeFieldBased` throws away the existing information and assumes that the clip is field-based, with the even-numbered fields being bottom fields and the odd-numbered fields being top fields. If you want it the other way around, use `ComplementParity` afterwards.

## 14.35 AssumeTFF / AssumeBFF

`AssumeTFF` (*clip*)  
`AssumeBFF` (*clip*)

Forcing the field order regardless of current value.

## 14.36 ComplementParity

`ComplementParity` (*clip*)

If the input clip is field-based, `ComplementParity` changes top fields to bottom fields and vice-versa. If the input clip is frame-based, it changes each frame's dominant field (bottom-dominant to top-dominant and vice-versa).

\$Date: 2005/10/13 21:41:11 \$

## 14.37 PeculiarBlend

PeculiarBlend (*clip, int cutoff*)

This filter blends each frame with the following frame in a peculiar way. The portion of the frame below the (*cutoff*)th scanline is unchanged. The portion above the (*cutoff*-30)th scanline is replaced with the corresponding portion of the following frame. The 30 scan lines in between are blended incrementally to disguise the switchover.

You're probably wondering why anyone would use this filter. Well, it's like this. Most videos which were originally shot on film use the 3:2 pulldown technique which is described in the description of the [PullDown](#) filter. But some use a much nastier system in which the crossover to the next frame occurs in the middle of a field—in other words, individual fields look like one movie frame on the top, and another on the bottom. This filter partially undoes this peculiar effect. It should be used after the [PullDown](#) filter. To determine *cutoff*, examine a frame which is blended in this way and set *cutoff* to the number of the first scanline in which you notice a blend.

This filter works only with YUY2 input.

\$Date: 2004/03/09 21:28:07 \$

## 14.38 Pulldown

Pulldown (*clip, int a, int b*)

The Pulldown filter simply selects two out of every five frames of the source video. The frame rate is reduced to two-fifths of its original value.

For example, Pulldown(0,2) selects frames 0, 2, 5, 7, 10, 12, and so on.

This filter is designed to be used after [DoubleWeave](#), and its purpose is to recover the original frames of a movie that was converted to video using the 3:2 pulldown process.

The reason you need to use [DoubleWeave](#) first is that capture cards combine fields in the wrong way. In terms of fields, the 3:2 pulldown sequence is simply "A A B B C C D D D ...", where "A" through "D" represent the original film frames. But the capture cards combine the fields into frames with no respect for the pulldown pattern, and you get this:

```
A  B  C  D  D      (30fps)
A  B  B  C  D
```

In this pattern frame C is never shown by itself. After [DoubleWeave](#) every pair of fields gets its own frame, so the video stream will begin like this:

```
A A B B C C D D D      (60fps)
A B B B B C C D D
*      *      *      *
```

Now each movie frame has at least one video frame to itself. At this point the Pulldown filter with arguments of 0, 3 will select the frames marked with a \*, and you'll get

```
A      B      C      D      (24fps)
```

A B C D

... which is what you really want.

This is all very complicated to describe, but in practice undoing the pulldown is just a matter of inserting some boilerplate code. See the example below under [ShowFiveVersions](#).

Pulldown(*a*, *b*) is implemented internally as [SelectEvery](#)(5, *a*, *b*) . [AssumeFrameBased](#).

\$Date: 2008/01/02 01:13:14 \$

## 14.39 HorizontalReduceBy2 / VerticalReduceBy2 / ReduceBy2

HorizontalReduceBy2 (*clip*)

VerticalReduceBy2 (*clip*)

ReduceBy2 (*clip*)

HorizontalReduceBy2 reduces the horizontal size of each frame by half, and VerticalReduceBy2 reduces the vertical size by half. Chain them together (in either order) to reduce the whole image by half. You can also use the shorthand ReduceBy2, which is the same as HorizontalReduceBy2 followed by VerticalReduceBy2.

The filter kernel used is (1/4,1/2,1/4), which is the same as in VirtualDub's "2:1 reduction (high quality)" filter. This avoids the aliasing problems that occur with a (1/2,1/2) kernel. VirtualDub's "resize" filter uses a third, fancier kernel for 2:1 reduction, but I experimented with it and found that it actually produced slightly worse-looking MPEG files—presumably because it sharpens edges slightly, and most codecs don't like sharp edges.

If your source video is interlaced, the VerticalReduceBy2 filter will deinterlace it (by field blending) as a side-effect. If you plan to produce output video in a size like 320x240, I recommend that you capture at full interlaced vertical resolution (320x480) and use VerticalReduceBy2. You will get much better-looking output. My Huffuyv utility will compress captured video about 2:1, losslessly, so you can capture 320x480 in about the same space as it used to take to capture 320x240. (If your disk has the capacity and throughput to support it, you can even capture at 640x480 and use both HorizontalReduceBy2 and VerticalReduceBy2. But this won't improve the quality as much, and if you have to go to MotionJPEG to achieve 640x480, you're probably better off with Huffuyv at 320x480.)

Note that, it's a quick and dirty filter (performance related compromise). Unlike the standard [resize](#) filters, the ReduceBy2 filters do not preserve the position of the image center. It shifts color planes by half of pixel. In fact, ReduceBy2( ) is equivalent to:

BilinearResize(Width/2, Height/2, 0.5, -0.5) for RGB,

MergeChroma(BilinearResize(Width/2,Height/2,0.5,0.5),BilinearResize(Width/2,Height/2,0.5,-0.5)) for YV12,

MergeChroma(BilinearResize(Width/2,Height/2,0.5,0.5),BilinearResize(Width/2,Height/2,0.5,-0.5)) for YUY2.

\$Date: 2008/12/24 19:19:07 \$



## 14.40 ResampleAudio

`ResampleAudio (clip, int new_rate_numerator[, int new_rate_denominator])`

`ResampleAudio` performs a high-quality change of audio samplerate. The conversion is skipped if the samplerate is already at the given rate.

When using fractional resampling the output audio samplerate is given by :

```
int(new_rate_numerator / new_rate_denominator + 0.5)
```

However the internally the resampling factor used is :

```
new_rate_numerator / (new_rate_denominator * old_sample_rate)
```

This causes the audio duration to vary slightly (which is generally what is desired).

Starting from v2.53 `ResampleAudio` accepts any number of channels.

Starting from v2.56 `ResampleAudio` process float samples directly. Support fractional resampling.

```
# resamples audio to 48 kHz
source = AviSource("c:\audio.wav")
return ResampleAudio(source, 48000)
```

```
# Exact 4% speed up for Pal telecine
Global Nfr_num=25
Global Nfr_den=1
AviSource("C:\Film.avi") # 23.976 fps, 44100Hz
Ar=Audiorate()
ResampleAudio(Ar*FramerateNumerator()*Nfr_den, FramerateDenominator()*Nfr_num)
AssumeSampleRate(Ar)
AssumeFPS(Nfr_num, Nfr_den, False)
```

For exact resampling the intermediate samplerate needs to be 42293.706293 which if rounded to 42294 would causes about 30ms per hour variation.

```
$Date: 2005/01/18 11:10:51 $
```

## 14.41 BicubicResize / BilinearResize / BlackmanResize / GaussResize / LanczosResize / Lanczos4Resize / PointResize / SincResize / Spline16Resize / Spline36Resize / Spline64Resize

`BicubicResize (clip, int target_width, int target_height, float "b=1./3.", float "c=1./3.", float "src_left", float "src_top", float "src_width", float "src_height")`

`BilinearResize (clip, int target_width, int target_height, float "src_left", float "src_top", float "src_width", float "src_height")`

`BlackmanResize (clip, int target_width, int target_height, float "src_left", float "src_top", float`

```
"src_width", float "src_height", int "taps=4")
GaussResize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height", float "p=30.0")
LanczosResize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height", int "taps=3")
Lanczos4Resize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height")
PointResize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height")
SincResize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height", int "taps=4")
Spline16Resize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height")
Spline36Resize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height")
Spline64Resize (clip, int target_width, int target_height, float "src_left", float "src_top", float
"src_width", float "src_height")
```

### 14.41.1 General information

From v2.56 you can use offsets (as in [Crop](#)) for all resizers:

i.e.

```
GaussResize (clip, int target_width, int target_height, float "src_left", float "src_top", float
-"src_right", float -"src_top")
```

For all resizers you can use an expanded syntax which crops before resizing. The same operations are performed as if you put a Crop before the Resize, there can be a slight speed difference.

Note the edge semantics are slightly different, Crop gives a hard absolute boundary, the Resizer filter lobes may extend into the cropped region but not beyond the physical edge of the image.

Use [Crop](#) to remove any hard borders or VHS head switching noise, using the Resizer cropping may propagate the noise into the adjacent output pixels. Use the Resizer cropping to maintain accurate edge rendering when excising a part of a complete image.

```
Crop(10,10,200,300).BilinearResize(100,150)
```

```
# which is nearly the same as
BilinearResize(100,150,10,10,200,300)
```

Important: AviSynth has completely separate vertical and horizontal resizers. If the input is the same as the output on one axis, that resizer will be skipped. Which one is called first, is determined by which one has the smallest downscale ratio. This is done to preserve maximum quality, so the second resizer has the best possible picture to work with. Data storing will have an impact on what modulus that "should" be used for sizes when resizing and cropping, see the [Crop](#) page.

### 14.41.2 BilinearResize

The `BilinearResize` filter rescales the input video frames to an arbitrary new resolution. If you supply the optional `SOURCE` arguments, the result is the same as if you had applied [Crop](#) with those arguments to the clip before `BilinearResize`.

## Avisynth 2.5 Selected External Plugin Reference

`BilinearResize` uses standard bilinear filtering and is almost identical to VirtualDub's "precise bilinear" resizing option. It's only "almost" because VirtualDub's filter seems to get the scaling factor slightly wrong, with the result that pixels at the top and right of the image get either clipped or duplicated. (This error is noticeable when expanding the frame size by a factor or two or more, but insignificant otherwise, so I wouldn't worry too much about it.)

Examples:

```
# Load a video file and resize it to 240x180 (from whatever it was before)
AVISource("video.avi").BilinearResize(240,180)
```

```
# Load a 720x480 (CCIR601) video and resize it to 352x240 (VCD),
# preserving the correct aspect ratio
AVISource("dv.avi").BilinearResize(352, 240, 8, 0, 704, 480)
```

```
# or what is the same
AVISource("dv.avi").BilinearResize(352, 240, 8, 0, -8, -0)
```

```
# Extract the upper-right quadrant of a 320x240 video and zoom it
# to fill the whole frame
BilinearResize(320,240,160,0,160,120)
```

### 14.41.3 BicubicResize

`BicubicResize` is similar to `BilinearResize`, except that instead of a linear filtering function it uses the Mitchell–Netravali two-part cubic. The parameters  $b$  and  $c$  can be used to adjust the properties of the cubic, they are sometimes referred to as 'blurring' and 'ringing,' respectively.

With  $b = 0$  and  $c = 0.75$  the filter is exactly the same as VirtualDub's "precise bicubic," and the results are identical except for the VirtualDub scaling problem mentioned above. The default values are  $b = 1./3.$  and  $c = 1./3.$ , which were the values recommended by Mitchell and Netravali as yielding the most visually pleasing results in subjective tests of human beings. Larger values of  $b$  and  $c$  can produce interesting op-art effects — for example, try  $b = 0$  and  $c = -5$ .

If you are magnifying your video, you will get much better-looking results with `BicubicResize` than with `BilinearResize`. However, if you are shrinking it, you are probably just as well off, or even better off, with `BilinearResize`. Although VirtualDub's bicubic filter does produce better-looking images than its bilinear filter, this is mainly because the bicubic filter sharpens the image, not because it samples it better. Sharp images are nice to look at—until you try to compress them, at which point they turn nasty on you very quickly. The `BicubicResize` default doesn't sharpen nearly as much as VirtualDub's bicubic, but it still sharpens more than the bilinear. If you plan to encode your video at a low bitrate, I wouldn't be at all surprised if `BilinearResize` yields a better overall final result.

For the most numerically accurate filter constrain  $b$  and  $c$  such that they satisfy :-

$$b + 2 * c = 1$$

This gives maximum value for  $c = 0.5$  when  $b = 0$ . This is the Catmull–Rom spline. Which is a good suggestion for sharpness.

From  $c > 0.6$  the filter starts to "ring". You won't get real sharpness, what you'll get is crispening like with a TV set sharpness control. Negative values are not allowed for  $b$ , use  $b = 0$  for values of  $c > 0.5$ .

### 14.41.4 BlackmanResize

`BlackmanResize` is a modification of `LanczosResize` that has better control of ringing artifacts for high numbers of taps. See `LanczosResize` of an explanation for the `taps` argument (default: `taps=4, 1<=taps<=100`). (added in v2.58)

### 14.41.5 GaussResize

`GaussResize` uses a gaussian resizer with adjustable sharpness parameter  $\rho$  (default 30).  $\rho$  has a range from about 1 to 100, with 1 being very blurry and 100 being very sharp. `GaussResize` uses 4 taps and has similar speed as `Lanczos4Resize`. (added in v2.56)

### 14.41.6 LanczosResize / Lanczos4Resize

`LanczosResize` is an alternative to `BicubicResize` with high values of  $c$  about 0.6 ... 0.75 which produces quite strong sharpening. It usually offers better quality (fewer artifacts) and a sharp image.

`Lanczos` was created for `AviSynth` because it retained so much detail, more so even than `BicubicResize(x,y,0,0.75)`. As you might know, the more detail a frame has, the more difficult it is to compress it. This means that `Lanczos` is NOT suited for low bitrate video, the various `Bicubic` flavours are much better for this. If however you have enough bitrate then using `Lanczos` will give you a better picture, but in general I do not recommend using it for 1 CD rips because the bitrate is usually too low (there are exceptions of course).

The input parameter `taps` (default 3, `1<=taps<=100`) is equal to the number of lobes (ignoring mirroring around the origin).

`Lanczos4Resize` (added in v2.55) is a short hand for `LanczosResize(taps=4)`. It produces sharper images than `LanczosResize` with the default `taps=3`, especially useful when upsizing a clip.

*Warning: the input argument named `taps` should really be lobes. When discussing resizers, `taps` has a different meaning, as described below (the first paragraph concerns `LanczosResize(taps=2)`):*

"For upsampling (making the image larger), the filter is sized such that the entire equation falls across 4 input samples, making it a 4-tap filter. It doesn't matter how big the output image is going to be – it's still just 4 taps. For downsampling (making the image smaller), the equation is sized so it will fall across  $4 * \text{destination}$  samples, which obviously are spaced at wider intervals than the source samples. So for downsampling by a factor of 2 (making the image half as big), the filter covers  $2*4=8$  input samples, and thus 8 taps. For 3x downsampling, you need  $3*4=12$  taps, and so forth.

Thus the effective number of taps you get for downsampling is the downsampling ratio times the number of filter input taps (thus  $T_x$  downsampling and `LanczoskResize` results in  $T*2*k$  taps), this is rounded up to the next even integer. For upsampling, it's always just  $2*k$  taps." Source: [[avsforum post](#)].

### 14.41.7 PointResize

`PointResize` is the simplest resizer possible. It uses a Point Sampler or Nearest Neighbour algorithm, which usually results in a very blocky image. So in general this filter should only be used, if you intend to have inferior quality, or you need the clear pixel drawings.

It is very useful for magnifying small areas of pixels for close examination.

### 14.41.8 Spline16Resize/Spline36Resize/Spline64Resize

Three Spline based resizers (added in v2.56/v2.58).

`Spline16Resize`, `Spline36Resize` and `Spline64Resize` are three Spline based resizers. They are the (cubic) spline based resizers from [Panorama tools](#) that fit a spline through the sample points and then derives the filter kernel from the resulting blending polynomials. See [this thread](#) for the details.

The rationale for Spline is to be as sharp as possible with less ringing artefacts than `LanczosResize` produces. `Spline16Resize` uses  $\sqrt{16}=4$  sample points, `Spline36Resize` uses 6 sample points, etc ... The more sample points that are used, the sharper your clip will be. A comparison between some resizers is given [here](#).

### 14.41.9 SincResize

`SincResize` is added in v2.6 and it uses the truncated sinc function as resizer. See `LanczosResize` for an explanation of the `taps` argument (default: `taps=4`;  $1 \leq \text{taps} \leq 20$ ).

#### Changelog:

v2.55	added <code>Lanczos4Resize</code>
v2.56	added <code>Spline16Resize</code> , <code>Spline36Resize</code> , <code>GaussResize</code> and <code>taps</code> parameter in <code>LanczosResize</code> ; added offsets in <code>Crop</code> part of <code>xxxResize</code>
v2.58	added <code>BlackmanResize</code> , <code>Spline64Resize</code>
v2.6	added <code>SincResize</code>

\$Date: 2009/09/12 15:10:22 \$

## 14.42 Reverse

`Reverse` (*clip*)

This filter makes a clip play in reverse. This is useful for watching people walk backwards while listening to hidden satanic messages.

Note: fields parity (Top Field First – Bottom Field First) of interlaced clip will be changed.

\$Date: 2006/12/11 20:14:59 \$

## 14.43 SegmentedAVISource / SegmentedDirectShowSource

`SegmentedAVISource` (*string base\_filename* [, ...], *bool "audio"*, *string "pixel\_type"*)  
`SegmentedDirectShowSource` (*string base\_filename* [, ...], *float "fps"*, *bool "seek"*, *bool "audio"*, *bool "video"*, *bool "convertfps"*, *bool "seekzero"*, *int "timeout"*, *string "pixel\_type"*)

The `SegmentedAVISource` filter automatically loads up to 100 avi files per argument (using [AVISource](#)) and splices them together (using [UnalignedSplice](#)). If "d:\filename.ext" is passed as an argument, the files d:\filename.00.ext, d:\filename.01.ext and so on through d:\filename.99.ext will be loaded. Any files in this sequence that don't exist will be skipped.

## Avisynth 2.5 Selected External Plugin Reference

If segments are spanned across multiple drives/folders, they can be loaded provided the folders are sorted in the correct order. For example

```
# D:\t1 contains cap.01.avi
D:\t1\cap.01.avi

# D:\t2 contains cap.02.avi - cap.03.avi
D:\t2\cap.02.avi
D:\t2\cap.03.avi

# F:\t3 contains cap.04.avi - cap.05.avi
F:\t3\cap.04.avi
F:\t3\cap.05.avi

# load all segments
SegmentedAVISource("D:\t1\cap.avi", "D:\t2\cap.avi", "F:\t3\cap.avi")
```

SegmentedDirectShowSource works the same way. Its arguments are described in [DirectShowSource](#).

From v2.04 up there is built-in support for ACM (Audio Compression Manager) audio (e.g. mp3-AVIs).

If you get an Unrecognized Exception in AviSynth 2.5 while reading a segmented avi generated by a VirtualDub capture, delete the small final .avi file.

**\$Date: 2008/02/10 13:57:17 \$**

### 14.44 SelectEven / SelectOdd

```
SelectEven(clip)
SelectOdd(clip)
```

SelectEven makes an output video stream using only the even-numbered frames from the input. SelectOdd is its odd counterpart.

Since frames are numbered starting from zero, SelectEven actually selects the first, third, fifth,... frames by human counting conventions.

**\$Date: 2004/03/09 21:28:07 \$**

### 14.45 SelectEvery

```
SelectEvery(clip, int step_size, int offset1 [, int offset2 [, ...]])
```

SelectEvery is a generalization of filters like [SelectEven](#) and [Pulldown](#). I think the easiest way to describe it is by example:

```
SelectEvery(clip,2,0)      # identical to SelectEven(clip)
SelectEvery(clip,2,1)      # identical to SelectOdd(clip)
SelectEvery(clip,10,3,6,7) # select frames 3, 6, 7, 13, 16, 17, ... from source clip
SelectEvery(clip,9,0)      # select frames 0, 9, 18, ... from source clip
```

And how about this:

## Avisynth 2.5 Selected External Plugin Reference

```
# Take a 24fps progressive input clip and apply 3:2 pulldown,  
# yielding a 30fps interlaced output clip  
AssumeFrameBased  
SeparateFields  
SelectEvery(8, 0,1, 2,3,2, 5,4, 7,6,7)  
Weave
```

\$Date: 2004/03/09 21:28:07 \$

### 14.46 SelectRangeEvery

SelectRangeEvery (*clip*, *int "every"*, *int "length"*, *int "offset"*, *bool "audio"*)

This filter is available starting from v2.5. The filter selects *length* number of frames *every* *n* frames, starting from frame *offset*. Default values are: Select 50 frames every 1500 frames, starting with first selection at frame 0. (*every* = 1500, *length* = 50, *offset* = 0).

Starting from v2.55, SelectRangeEvery will also process audio. To keep the original audio, use *audio* = false.

Examples:

```
# Selects the frames 0 to 13, 280 to 293, 560 to 573, etc.  
SelectRangeEvery(clip, 280, 14)
```

```
# Selects the frames 2 to 15, 282 to 295, 562 to 575, etc.  
SelectRangeEvery(clip, 280, 14, 2)
```

\$Date: 2004/07/21 18:54:28 \$

### 14.47 SeparateFields

SeparateFields (*clip*)

NTSC and PAL video signals are sequences of fields, but all capture cards that I'm aware of capture two fields at a time and interlace (or "weave") them into frames. So frame 0 in the capture file contains fields 0 and 1; frame 1 contains fields 2 and 3; and so on. SeparateFields takes a frame-based clip and splits each frame into its component fields, producing a new clip with twice the frame rate and twice the frame count. This is useful if you would like to use [Trim](#) and similar filters with single-field accuracy.

SeparateFields uses the field-dominance information in the source clip to decide which of each pair of fields to place first in the output. If it gets it wrong, use [ComplementParity](#), [AssumeTFF](#) or [AssumeBFF](#) before SeparateFields.

From version 2.5.6 this filter raises an exception if the clip is already field-based. You may want to use [AssumeFrameBased](#) to force separate a second time. Prior versions did a no-op for materials that was already field-based.

\$Date: 2009/09/12 15:10:22 \$

## 14.48 ShowAlpha, ShowRed, ShowGreen, ShowBlue

ShowAlpha (*clip*, *string pixel\_type*)

ShowBlue (*clip*, *string pixel\_type*)

ShowGreen (*clip*, *string pixel\_type*)

ShowRed (*clip*, *string pixel\_type*)

ShowAlpha shows the alpha channel of a RGB32 clip, available in v2.53. ShowBlue / ShowGreen / ShowRed shows the selected channel of a RGB clip, available in v2.56.

In v2.54 ShowAlpha now returns RGB, YUY2, or YV12 via the *pixel\_type* argument. The latter two can be used to layer an RGB clip with alpha transparency data onto a YUV clip using the 3–argument form of Overlay, because when setting *pixel\_type* to YUY2 or YV12 the luma range is [0,255].

In v2.56 ShowAlpha/Red/Green/Blue now returns RGB24, RGB32, YUY2, or YV12 via the *pixel\_type* argument. For RGB32 output the selected channel is copied to all R, G and B channels, but not the Alpha channel which is left untouched. For YUV output the selected channel is copied to the Luma channel, the chroma channels are set to grey (0x80).

### Examples:

```
# shows alpha channels of clip
AviSource("clip.avi")
ShowAlpha()

# swaps red and blue channels:
AviSource("clip.avi")
MergeRGB(ShowBlue("YV12"), Last, ShowRed("YV12"))
```

### Changelog:

v2.56	added ShowBlue, ShowGreen and ShowRed
-------	---

\$Date: 2005/07/08 22:53:16 \$

## 14.49 ShowFiveVersions

ShowFiveVersions (*clip1*, *clip2*, *clip3*, *clip4*, *clip5*)

ShowFiveVersions takes five video streams and combines them in a staggered arrangement from left to right. The only use for this (that I can think of) is to help find the NTSC pulldown pattern. You can do this using code like this:

```
# View all five pulldown patterns at once
DoubleWeave()
# put a resizing filter here if necessary (see below)
a = Pulldown(0,2).Subtitle("0,2")
b = Pulldown(1,3).Subtitle("1,3")
c = Pulldown(2,4).Subtitle("2,4")
d = Pulldown(0,3).Subtitle("0,3")
```



## Avisynth 2.5 Selected External Plugin Reference

```
e = Pulldown(1,4).Subtitle("1,4")
ShowFiveVersions(a,b,c,d,e)
```

This code displays the five pulldown patterns with some text identifying which is which. I then look through the movie and pick the pattern which avoids blending frames. (In ordinary pulldown, there will actually be two which work equally well. Look at the diagrams in the [Pulldown](#) filter section to see why.) If none of the five works, then you're dealing with one of the more perverse forms of pulldown and you might want to use [PeculiarBlend](#).

By the way, if you're planning to capture at a high resolution and then scale down, as I recommend elsewhere, you should probably place the [ReduceBy2](#) or [BilinearResize](#) or whatever just after the [DoubleWeave](#) statement in the code above. Before [DoubleWeave](#) it won't work correctly, and if you postpone it any further, ShowFiveVersions will produce a really big frame.

### Changelog:

v2.56	added YV12
-------	------------

\$Date: 2005/01/26 22:08:36 \$

## 14.50 ShowFrameNumber

ShowFrameNumber (*clip*, *bool "scroll"*, *int "offset"*, *int "x"*, *int "y"*, *string "font"*, *float "size"*, *int "text\_color"*, *int "halo\_color"*, *float "font\_width"*, *float "font\_angle"*)

ShowFrameNumber draws text on every frame indicating what number AviSynth thinks it is. This is sometimes useful when writing scripts. If you apply additional filters to the clip produced by ShowFrameNumber, they will treat the text on the frame just as they would treat an image, so the numbers may be distorted by the time you see them.

If *scroll* (default: false) is set to true the framenumbers will be drawn only once on the video and scroll from top to bottom, else it will be drawn on the right side as often as it fits. For top field first material the framenumbers will be drawn on the left side of the clip, for bottom field first material on the right side and for field-based material it will be drawn alternating on the left side and right side of the clip (depending whether the field is top or bottom).

Starting from v2.56 other options ..., *x*, *y*, *font*, *size*, *text\_color*, *halo\_color*, *font\_width*, *font\_angle*) are present, see [Subtitle](#) for an explanation of these options.

*offset* enables the user to add an offset to the shown framenumbers.

## 14.51 ShowSMPTE

ShowSMPTE (*clip*, *float "fps"*, *string "offset"*, *int "offset\_f"*, *int "x"*, *int "y"*, *string "font"*, *float "size"*, *int "text\_color"*, *int "halo\_color"*, *float "font\_width"*, *float "font\_angle"*)

ShowSMPTE is similar to ShowFrameNumber but displays SMPTE timecode (hours:minutes:seconds:frame). Starting from v2.53 the *fps* argument is not required, unless the current fps can't be used. Otherwise, the *fps* argument is required and must be 23.976, 24, 25, 29.97, or 30.

## Avisynth 2.5 Selected External Plugin Reference

Starting from v2.56 other options *...*, *x*, *y*, *font*, *size*, *text\_color*, *halo\_color*, *font\_width*, *font\_angle*) are present, see [Subtitle](#) for an explanation of these options.

*offset* enables the user to add an offset to the timecode, while *offset\_f* enables the user to add an offset to the timecode specifying the number of frames (*offset* takes precedence over *offset\_f*).

### drop-frame versus non-drop-frame timecode

If the framerate of the clip is between 29.969 and 29.971 [drop-frame timecode](#) is enabled. Originally, when the signal of the TV was black and white, NTSC run at 60 Hz (30 fps). When they added color, they changed it to 59.94 Hz (29.97 fps) due to technical reasons. They run 1000 frames, but count 1001 (they never actually drop a frame, just a frame number). The first two frames are dropped of every minute except the tenth, ie 00:00:00:00, 00:00:00:01, 00:00:00:02, ..., 00:00:59:29, 00:01:00:02, 00:01:00:03, ..., 00:01:59:29, 00:02:00:02, 00:02:00:03, ..., 00:08:59:29, 00:09:00:02, 00:09:00:03, ..., 00:09:59:29, 00:10:00:00, 00:10:00:01, etc ... Counting the dropped frames implies that 00:10:00:00 in drop-frame matches 00:10:00:00 in real time.

### Examples

```
ShowSMPTE(offset="00:00:59:29", x=360, y=576, font="georgia",  
/         size=24, text_color=$ff0000)
```

```
Mpeg2Source("clip.d2v") # is always top field first  
# will draw the framenummer on the left side of the clip using  
# an offset of 9 frames, scrolling from top to bottom  
ShowFrameNumber(scroll=true, offset=9, text_color=$ff0000)
```

## 14.52 ShowTime

*ShowTime* (*clip*, *int "offset\_f"*, *int "x"*, *int "y"*, *string "font"*, *float "size"*, *int "text\_color"*, *int "halo\_color"*, *float "font\_width"*, *float "font\_angle"*)

*ShowTime* is similar to *ShowSMPTE* but it displays the time duration (hours:minutes:seconds.ms). See *ShowSMPTE* for an explanations of the options.

Take care: these filters are quite slow due to the text-drawing.

### Changes

v2.58	Added ShowTime function Added font_width, font_angle args
v2.56	Added offset and other options.

\$Date: 2008/08/10 12:40:46 \$

## 14.53 SpatialSoften / TemporalSoften

*SpatialSoften* (*clip*, *int radius*, *int luma\_threshold*, *int chroma\_threshold*)

*TemporalSoften* (*clip*, *int radius*, *int luma\_threshold*, *int chroma\_threshold*, *int "scenechange"*, *int "mode"*)

## Avisynth 2.5 Selected External Plugin Reference

The `SpatialSoftten` and `TemporalSoftten` filters remove noise from a video clip by selectively blending pixels. These filters can work miracles, and I highly encourage you to try them. But they can also wipe out fine detail if set too high, so don't go overboard. And they are very slow, especially with a large value of *radius*, so don't turn them on until you've got everything else ready.

`SpatialSoftten` replaces each sample in a frame with the average of all nearby samples with differ from the central sample by no more than a certain threshold value. "Nearby" means no more than *radius* pixels away in the x and y directions. The threshold used is *luma\_threshold* for the Y (intensity) samples, and *chroma\_threshold* for the U and V (color) samples.

`TemporalSoftten` is similar, except that it looks at the same pixel in nearby frames, instead of nearby pixels in the same frame. All frames no more than *radius* away are examined. This filter doesn't seem to be as effective as `SpatialSoftten`.

I encourage you to play around with the parameters for these filters to get an idea of what they do—for example, try setting one of the three parameters to a very high value while leaving the others low, and see what happens. Note that setting any of the three parameters to zero will cause the filter to become a very slow no-op.

`TemporalSoftten` smoothes luma and chroma separately, but `SpatialSoftten` smoothes only if both luma and chroma have passed the threshold.

The `SpatialSoftten` filter work only with YUY2 input. You can use the [ConvertToYUY2](#) filter if your input is not in YUY2 format.

Note that if you use `AviSynth v2.04` or above, you don't need the `TemporalSoftten2` plugin anymore, the built-in `TemporalSoftten` is replaced with that implementation.

Starting from `v2.50`, two options are added to `TemporalSoftten`:

- An optional *mode=2* parameter: It has a new and better way of blending frame and provides better quality. It is also much faster. Requires ISSE. *mode=1* is default operation, and works as always.
- An optional *scenechange=n* parameter: Using this parameter will avoid blending across scene changes. 'n' defines the maximum average pixel change between frames. Good values for 'n' are between 5 and 30. Requires ISSE.

Good initial values: `TemporalSoftten(4,4,8,15,2)`

### Changes:

v2.56	TemporalSoftten working also with RGB32 input (as well as YV12, YUY2)
-------	---

⌘Date: 2007/07/14 18:06:23 ⌘

## 14.54 SoundOut

SoundOut is a GUI driven sound output module for AviSynth.

### 14.54.1 Installation and Usage

Copy "SoundOut.dll" and "libsndfile-1.dll" to your AviSynth plugin directory, usually "c:\program files\avisynth 2.5\plugins". If you want to have "SoundOut.dll" at another location, you should move "libsndfile-1.dll" to your system32 folder, usually "c:\windows\system32".

Add `SoundOut ( )` to your script, where you would like to export audio. If you have your video stored in a variable, use `SoundOut (variable)` to add SoundOut. A GUI should pop up, when you open your script. Here is a simple example of how to use it:

```
AviSource("myvideo.avi")
SoundOut()
```

If you need some sample processing, to change samplerate or otherwise edit your video, you must do it before calling the SoundOut module. Like this:

```
AviSource("myvideo.avi")
AmplifydB(3)
SSRC(44100)
SoundOut()
```

### 14.54.2 Output Modules

#### 14.54.2.1 WAV/AIF/CAF

This will allow you to export uncompressed audio to the following formats:

- Microsoft WAV format
- Apple/SGI AIFF format
- Sun/NeXT AU format
- RAW PCM data
- Sonic Foundry's 64 bit RIFF/WAV (WAVE64)
- Apple Core Audio File format
- Microsoft WAV format with Broadcast Wave Format chunk.

Note, that 8 bit samples are NOT supported in the Core Audio File and Sun/NeXT AU format.

#### 14.54.2.2 FLAC

This will allow you to export lossless compressed audio FLAC format.

FLAC supports 8,16 or 24 bit audio. Any other format is internally converted to 24 bit.

#### 14.54.2.3 APE

This will allow you to export lossless compressed audio to the Monkey Audio Codec (APE) format.

## Avisynth 2.5 Selected External Plugin Reference

APE does not support input sample sizes that are larger than 2GB. Use only for smaller files.

APE supports 8, 16 or 24 bit audio. Any other format is internally converted to 24 bit.

### 14.54.2.4 MP2

This will allow you to compress your audio to MPEG 1 Layer 2 (MP2).

TwoLame only supports 16 mono or stereo audio. If you attempt to compress more than two channels, an error will be shown. Any other format than 16 bit integer samples are internally converted to 16 bit.

### 14.54.2.5 MP3

This will allow you to compress your audio to MPEG 1 Layer 3 (MP3) using LAME v3.97 encoder.

LAME Supports up to two channels of audio and the following samplerates: 48000, 44100, 32000, 24000, 22050, 16000, 12000, 11025 and 8000 Hz.

### 14.54.2.6 AC3

This will allow you to compress your audio to A/52 (AC3). The encoding is done via libaften.

Aften supports 1 to 6 channel audio. Supported samplerates are 48000, 44100 or 32000 samples per second.

Channel mapping is:

Number of channels	Channel order
1	Center
2	Left, Right
3	Left, Center, Right
4	Left, Right, Surround Left, Surround Right
5	Left, Center, Right, Surround Left, Surround Right
6	Left, Center, Right, Surround Left, Surround Right, LFE

### 14.54.2.7 OGG

This will allow you to compress your audio to an Vorbis encoded OGG file. It is possible to give an average bitrate, or do the encode as CBR.

### 14.54.2.8 Commandline Output

This output module will allow you to output to any program that supports input from stdin. This gives you complete control of your encoding, if you have commandline tools for the job.

You can select the format SoundOut should deliver to the application you use. There are three WAV formats and RAW PCM data. This is sent to stdin of the application. The program builds the command line from 4 parts, *the executable*, *command line options* before the output file, the *output file* that is selected, and *command line options* after the output file name.

There are two ways of specifying the executable. Either give complete path to the executable, or simply enter the executable's filename, and place it in a subdirectory called SoundOut in your plugin directory.

### 14.54.3 Exporting from script

It is possible to use SoundOut as an ordinary filter, running inside the script and giving parameters for each output mode. The parameters consists of two things: General Parameters, which can be used for all filters, and filter specific parameters, which gives parameters to the active output module.

The **out** parameter determines whether the GUI will be shown, if it is properly set, the filter will begin exporting audio as soon as it is started.

If the **out** parameter is **not** set, it is still possible to set additional parameters. The defaults will however be retrieved from the registry, but specific parameters override

#### 14.54.3.1 General Parameters

Parameter name	Type	Values
output	string	Select output module to use. Possible values are: "WAV", "AC3", "MP2", "MP3", "OGG", "FLAC", "MAC" and "CMD". If none, or an invalid value is given, the ordinary GUI will be shown.
filename	string	Full path to the output filename, including extension. No extra quotes are required. If no filename is given a file selector will pop up.
showprogress	bool	Show the progress window? Default: true
overwritefile	string	"Yes": Always overwrite file. "No": Never Overwrite file "Ask": Ask if file should be overwritten.
autoclose	bool	Should the progress window close automatically 5 seconds after encoding has finished? This will also code the window, even though an error occurred Default: false
silentblock	bool	When processing, enabling this option will return silent samples instead of blocking the requesting application. If disabled, any application requesting audio will be blocking, while sound is being exported Default: true
addvideo	bool	When enabled, this will add video to the current output, if none is present. The video is a black 32x32 pixels at 25fps, with the length of the audio. Default: true
wait	integer	How many seconds should the output window be shown, if autoclose is on. Default: 5.

#### 14.54.3.2 WAV/AIF/CAF Script Parameters:

Parameter name	Type	Values
type	integer	Select WAVE format 0: Microsoft WAV (default), 1: WAV with WAVEFORMATEX, 2: Apple/SGI AIFF, 3: Sun/NeXT AU, 4: RAW PCM, 5: S.F. WAVE64, 6: Core Audio File,

## Avisynth 2.5 Selected External Plugin Reference

		7: Broadcast Wave.
format	integer	Sets the sample format number of bits per sample. 0: 16bit per sample, 1: 24bit per sample, 2: 32bit per sample, 3: 32bit float per sample, Default: Same as input.
peakchunck	bool	Add Peak chunk to WAV file? Default: false

Audio will be written in the format delivered to the SoundOut plugin. All internal sound formats are supported.

### 14.54.3.3 FLAC Script Parameters:

Parameter name	Type	Values
compressionlevel	integer	Sets the compression level. 1(fastest) to 8(slowest) Default: 6

### 14.54.3.4 APE Script Parameters:

Parameter name	Type	Values
compressionlevel	integer	Sets the compression level. 1(fastest) to 6(slowest) Default: 3

### 14.54.3.5 MP2 Script Parameters:

Parameter name	Type	Values
bitrate	integer	Sets Bitrate for CBR or maximum bitrate for VBR. Default: 192
stereomode	integer	-1: Automatic (default) 0: Separate Stereo 1: Separate Stereo 2: Joint Stereo 3: Dual Channel 4: Mono
psymodel	integer	-1: Fast &Dumb 0: Low complexity 1: ISO PAM 1 2: ISO PAM 2 3: PAM 1 Rewrite (default) 4: PAM 2 Rewrite
vbrquality	float	Sets VBR Quality. Useful range is about -10 to 10. Default is 0
vbr	bool	Encode as VBR? Default: false.
quick	bool	Quick Encode? Default: false.
dab	bool	Add DAB Extensions? Default: false. According to TwoLame documentation this might not be reliable.

## Avisynth 2.5 Selected External Plugin Reference

crc	bool	Add CRC Error checks? Default: false.
original	bool	Set Original Flag? Default: false.
copyright	bool	Set Copyright flag? Default: false.
emphasis	integer	Set Emphasis flag. 0: No Emphasis (default) 1: 50/15 ms 3: CCIT J.17

### 14.54.3.6 MP3 Script Parameters:

Parameter name	Type	Values
mode	integer	Sets Encoding mode: 0: VBR (default) 1: ABR 2: CBR
vbrpreset	integer	Sets quality preset, when using VBR mode. Standard = 1001 (default), extreme = 1002, insane = 1003, standard_fast = 1004, extreme_fast = 1005, medium = 1006, medium_fast = 1007
abrrate	integer	Sets Average bitrate for ABR encoding. Default: 128
cbrrate	integer	Sets Bitrate for CBR encoding. Default: 128

### 14.54.3.7 AC3 Script Parameters:

Parameter name	Type	Values
iscbr	bool	Encode at Constant Bitrate? Default: true.
cbrrate	integer	Sets Bitrate for CBR or maximum bitrate for VBR. Default: 384
vbrquality	integer	VBR Bitrate quality. Values between 1 and 1023 are accepted Default: 220.
drc	integer	Dynamic Range Compression 0: Film Light 1: Film Standard 2: Music Light 3: Music Standard 4: Speech 5: None (default)
acmod	integer	Set channel mapping 0 = 1+1 (Ch1,Ch2) 1 = 1/0 (C) 2 = 2/0 (L,R)



## Avisynth 2.5 Selected External Plugin Reference

		3 = 3/0 (L,R,C) 4 = 2/1 (L,R,S) 5 = 3/1 (L,R,C,S) 6 = 2/2 (L,R,SL,SR) 7 = 3/2 (L,R,C,SL,SR)
dialognormalization	integer	Dialog normalization. Values from 0 to 31 are accepted Default: 31.
islfe	bool	Is there LFE channel present? Default: false if less than 4 channels, true otherwise.
bandwidthfilter	bool	Use the bandwidth low-pass filter? Default: false.
lfe_lowpass	bool	Use the LFE low-pass filter Default: false.
dchighpass	bool	Use the DC high-pass filter Default: false.
dolbysurround	bool	Is the material Dolby Surround encoded? (only applies to stereo sound, otherwise ignored) Default: false.
blockswitch	bool	Selectively use 256-point MDCT? Default: false (Use only 512-point MDCT).
accuratealloc	bool	Do more accurate encoding? Default: true.

### 14.54.3.8 OGG Script Parameters:

Parameter name	Type	Values
vbrbitrate	integer	Selects the average bitrate to encode at Default: 128.
cbr	bool	Encode as CBR? Default: false.

### 14.54.3.9 Wavpack Script Parameters:

Parameter name	Type	Values
compressionlevel	integer	Sets the compression level. 0(Very Fast) to 5(Extremely Slow) Default: 2 (Normal)
format	integer	Sets the sample format number of bits per sample. 0: 8bit per sample, 1: 16bit per sample, 2: 24bit per sample, 3: 32bit per sample, 4: 32bit float per sample, Default: Same as input.

### 14.54.3.10 Commandline Output Script Parameters:

Parameter name	Type	Values
type	integer	Select WAVE format 0: Microsoft WAV (default), 1: WAV with WAVEFORMATEX, 2: RAW PCM,

## Avisynth 2.5 Selected External Plugin Reference

		3: S.F. WAVE64.
format	integer	Select Output Bits per sample 0: 16 Bit 1: 24 Bit 2: 32 Bit 3: 32 bit float Default is same as input.
executable	string	Executable to use Default: "aften.exe" (without quotes).
prefilename	string	Parameters that are placed before the output filename Default: "-b 384 -" (without quotes).
postfilename	string	Parameters that are placed after the output filename Default: "" (without quotes).
showoutput	bool	Show the output window? Default: true.
nofilename	bool	Encode without output filename, and don't use postfilename? Default: false.

### 14.54.4 Examples

```
SoundOut(output = "mp3", filename="c:\outputFile.mp3", autoclose = true, showprogress=true, mod
```

Engages mp3 output module with CBR at 192kbit/sec.

### 14.54.5 Implementation notes

SoundOut is Multithreaded, and uses one thread for requesting audio from the previous filters, and another thread for encoding. The threads are given a "below normal" priority.

Only attempt to run two exports at the same time at your own risk. It is most likely slower and could potentially crash. You can safely export sound while you encode, if your encode does not read audio from AviSynth.

### 14.54.6 Changelist

v2.60	Initial Release; based on v1.1.1
-------	----------------------------------

v1.1.1

- Downgraded FLAC to v1.2.0, to avoid backwards incompatible 24 bit files.
- Conversion tune-up.
- OverWriteFile set to "No" was not respected.
- Client sample requests shown in GUI.

v1.1.0

- Added WavPack output module.
- Added Sample type selection to WAV Output.
- Updated FLAC to v 1.2.1 – 24 bit/sample seems broken, so only 8 & 16 bit are enabled.
- Fixed bug in FLAC to enable files larger than 2GB.
- FLAC now uses the same GUI as other filters.

## Avisynth 2.5 Selected External Plugin Reference

- Aften updated.
- Re-enabled Aften multithreading.
- Faster 3DNOW! float to 24 bit conversion.

### v1.0.3

- Vorbis, AC3 and MP3 now checks if file can be created.
- Fixed hang in aften on multiprocessor machines.
- Added wait parameter, how many seconds should SoundOut wait on autoclose.
- Avoid lockup if encoder cannot be initialized and set for direct output.
- Fixed OverwriteFile was not always being respected.

### v1.0.2

- Updated libaften to rev534.
- Fixed overwriteFile not being recognized in script.
- Fixed crash if mp2 file could not be opened for writing.
- Exit blocked, even if filter is (almost) instantly destroyed, if script is set for output.
- AC3 is now reporting the actual samples encoded (including padding).

### v1.0.1

- Updated libaften to rev. 512.
- Added overwriteFile="yes"/"no"/"ask". Default is Ask.

### v1.0.0

- The application will not exit, as long as an encode window is open.
- Fixed "nofilename" not being recognized in script.
- LFE no longer overridden by registry, when using GUI.

### v0.9.9

- Added ReplayGain calculation to Analyze.
- Parent filters are now blocked, or silent samples are returned, if the filter is currently exporting sound.
- Video is automatically added, if none is present. (black 32x32 RGB32)
- Buttons for export are disabled when output window is open.
- Main window is now minimized when export module is selected.
- Fixed Analyze bug on 16 bit samples.
- Fixed WAVEFORMATEXTENSIBLE channel mapping in Commandline Output.
- AC3 output: LFE option disabled when not relevant.
- AC3 output: LFE option named properly.

### v0.9.8

- Added Analyze option to calculate average, maximum and RMS levels. Only available through GUI.
- WAVEFORMATEXTENSIBLE in commandline out attempts to set channel maps based on channel number.
- Fixed thread race issue on very fast encoders.
- Minor GUI tweaks.

### v0.9.7

- Added channelmapping to AC3 output.
- Added LFE channel indicator switch to AC3 output.
- GUI now spawned in a new thread, fixing GUI lockup in foobar2000 and similar.

## Avisynth 2.5 Selected External Plugin Reference

- Fixed general thread race issue, where a fast encoder might lead to incomplete output.
- Fixed WAVE\_FORMAT\_EXTENSIBLE header without info in CmdLine Output.
- Fixed "Format" not working on Commandline output.
- Fixed Filename dialog not appearing.
- Forced final samplereading to be correct.
- Removed "private" option from MP2 GUI and script, as there is no way to set it via twolame.
- Removed DAB Extensions from MP2 GUI, as TwoLame reports it as not functioning.

### v0.9.6

- Added complete script customization.
- Added possibility to set output file from script.
- Added window autoclose option to script.
- Added option to script to disable progress window.
- GUI creates message handle thread.
- Settings are now saved to registry if output filter initializes successfully.
- Updated documentation.

### v0.9.5

- Added Broadcast WAVE out.
- Fixed OGG Vorbis support.
- Fixed Text fields not being correctly read.
- Fixed AC3 settings not being restored properly.
- Added: MP2 settings are now saved.

### v0.9.4

- Added OGG Vorbis support.
- Added: Parameters stored (on save) and read to registry.
- Added: "No filename needed" option in commandline output, to disable output filename prompt.
- Fixed collision between libaften and libvorbis.
- Updated libaften to rev 257.
- Enabled SSE optimizations in libaften.
- Hopefully fixed issue with commandline executable filename becoming garbled.

### v 0.9.3

- Added Commandline piping output.
- Added MP3 / LAME output.
- Fixed AC3 VBR Error sometimes wrongly being displayed.
- Fixed AC3 DRC Setting not being respected.
- Various GUI bugfixes.

### v 0.9.2

- Updated AC3 GUI.
- Fixed crash in WAV output.
- More stats during conversion.

### v 0.9.1

- Added AC3 Output.
- Added new parameter handling.
- Fixed last block not being encoded.

**\$Date: 2009/09/12 15:10:22 \$**

## 14.55 AlignedSplice / UnalignedSplice

```
AlignedSplice (clip1, clip2 [, ...])
UnAlignedSplice (clip1, clip2 [, ...])
```

`AlignedSplice` and `UnalignedSplice` join two or more video clips end to end. The difference between the filters lies in the way they treat the sound track. `UnalignedSplice` simply concatenates the sound tracks without regard to synchronization with the video. `AlignedSplice` cuts off the first sound track or inserts silence as necessary to ensure that the second sound track remains synchronized with the video.

You should use `UnalignedSplice` when the soundtracks being joined were originally contiguous – for example, when you're joining files captured with `AVI_IO`. Slight timing errors may lead to glitches in the sound if you use `AlignedSplice` in these situations.

Avisynth's scripting language provides `+` and `++` operators as synonyms for `UnalignedSplice` and `AlignedSplice` respectively.

Also see [here](#) for the resulting clip properties.

```
# Join segmented capture files to produce a single clip
UnalignedSplice(AVISource("cap1.avi"),AVISource("cap2.avi"),AVISource("cap3.avi"))
# or: AVISource("cap1.avi") + AVISource("cap2.avi") + AVISource("cap3.avi")

# Extract three scenes from a clip and join them together in a new order
AVISource("video.avi")
edited_video = Trim(2000,2500) ++ Trim(3000,3500) ++ Trim(1000,1500)
```

**\$Date: 2004/03/09 21:28:07 \$**

## 14.56 SSRC

`SSRC` (*int samplerate, bool "fast"*)

`SSRC` Shibata Sample Rate Converter is a resampler. Audio is always converted to float. This filter will result in better audio quality than [ResampleAudio](#).

It uses `SSRC` by [Naoki Shibata](#), which offers the best resample quality available.

Sampling rates of 44.1kHz and 48kHz are popularly used, but the ratio of these two frequency is 147:160, and that are not small numbers. Therefore, sampling rate conversion without degradation of sound quality requires filters with very large order, and it's difficult to achieve both quality and speed. This program achieved relatively fast and high quality with two different kinds of filters combined skillfully.

### Parameters:

samplerate	Samplerate must be an integer.
fast	This will enable faster processing at slightly lower quality. Disable this if you are doing large samplerate conversions (more than a factor 2). Default: true.

## Avisynth 2.5 Selected External Plugin Reference

SSRC doesn't work for arbitrary ratios of the samplerate of the source and target clip. The following ratios are allowed (see SSRC.c):

```
sfrq = samplerate of source clip
dfrq = samplerate of destination clip
frqgcd = gcd(sfrq,dfrq)
fsl = dfrq * sfrq / frqgcd
Resampling is possible if: (fsl/dfrq == 1) or (fsl/dfrq % 2 == 0) or (fsl/dfrq % 3 == 0)
```

example for which resampling is possible:

```
sfrq = 44.1 kHz
dfrq = 48 kHz
frqgcd = gcd(44100,48000) = 300
fsl / dfrq = sfrq / frqgcd = sfrq / gcd(sfrq,dfrq) = 44100/300 = 147
and 147%3=0 since 147 / 3 = 49 = integer
```

The samplerate of your source clip can be found as follows

```
AviSource(...)
Subtitle(string(c.AssumeFPS(23.976,sync_audio=true).AudioRate))
```

### Example:

```
# Downsampling to 44,1 kHz:
SSRC(44100)
```

### Changelog:

v2.54	Initial Release
-------	-----------------

Some parts of SSRC is: Copyright © 2001–2003, Peter Pawlowski. All rights reserved.

**\$Date: 2009/09/12 15:10:22 \$**

## 14.57 StackHorizontal / StackVertical

```
StackHorizontal (clip1, clip2 [, ...])
StackVertical (clip1, clip2 [, ...])
```

`StackHorizontal` takes two or more video clips and displays them together in left-to-right order. The heights of the images and their color formats must be the same. Most other information (sound track, frame rate, etc) is taken from the first clip – see [here](#) for the resulting clip properties. `StackVertical` does the same, except from top to bottom.

```
# Compare frames with and without noise reduction
StackVertical(last, last.SpatialSoften(2,3,6))

#
# Show clips in variables a,b,c,d in a box like this:
StackVertical(StackHorizontal(a,b),StackHorizontal(c,d))
```

**\$Date: 2004/03/09 21:28:07 \$**

## 14.58 Subtitle

`Subtitle (clip, string text, int "x", int "y", int "first_frame", int "last_frame", string "font", float "size", int "text_color", int "halo_color", int "align", int "spc", int "lsp", float "font_width", float "font_angle", bool "interlaced")`

`Subtitle (clip, string text)`

The `Subtitle` filter adds anti-aliased text to a range of frames. If you want more than one subtitle, you have to chain several `Subtitle` filters together. All parameters after text are optional and can be omitted or specified out of order using the `name=value` syntax.

### 14.58.0.1 Parameters

`text` is the text which will be overlaid on the clip starting from frame `first_frame` and ending with frame `last_frame`.

`(x,y)` is the position of the text. The parameters `x` and `y` can be set to `-1` to automatically calculate and use the horizontal or vertical center coordinate. Other negative values of `x` and `y` can be used to give subtitles partially off the screen. **Caution:** If your script uses `Subtitle` with [Animate](#) and negative `x` or `y` values, `x` or `y` might momentarily become `-1`, causing a glitch in the video.

`font` is the font of the text (all installed fonts on the current machine are available, they are located in your `'windows\fonts'` folder).

`size` is the height of the text in pixels, and is rounded to the nearest 0.125 pixel.

The `text_color` and `halo_color` should be given as hexadecimal `$aarrggbb` values, similar to HTML—except that they start with `$` instead of `#` and the 4th octet specifies the alpha transparency. `$00rrggbb` is completely opaque, `$FF000000` is fully transparent. You can disable the halo by selecting this color. See [here](#) for more information on specifying colors.

The `align` parameter allows you to set where the text is placed relative to the `(x,y)` location and is based on the numeric keypad as follows:

<code>&lt;left&gt; 7</code> <code>&lt;top&gt;</code>	<code>&lt;center&gt; 8</code> <code>&lt;top&gt;</code>	<code>&lt;right&gt; 9</code> <code>&lt;top&gt;</code>	top of text aligned to y-location for align=7,8,9
<code>&lt;left&gt; 4</code> <code>&lt;baseline&gt;</code>	<code>&lt;center&gt; 5</code> <code>&lt;baseline&gt;</code>	<code>&lt;right&gt; 6</code> <code>&lt;baseline&gt;</code>	baseline of text aligned to y-location for align=4,5,6
<code>&lt;left&gt; 1</code> <code>&lt;bottom&gt;</code>	<code>&lt;center&gt; 2</code> <code>&lt;bottom&gt;</code>	<code>&lt;right&gt; 3</code> <code>&lt;bottom&gt;</code>	bottom of text aligned to y-location for align=1,2,3
start at x for align=1,4,7	center on x for align=2,5,8	end at x for align=3,6,9	

Note: There is no vertical center alignment setting.

## Avisynth 2.5 Selected External Plugin Reference

The *spc* parameter allows you to modify the character spacing (0=unchanged). The value can be positive or negative to widen or narrow the text. Per the Visual C++ documentation of the function `SetTextCharacterExtra()`, that performs this task, this value is in logical units and rounded to the nearest 0.125 pixel. This is helpful for trying to match typical fonts on the PC to fonts used in film and television credits which are usually wider for the same height or to just fit or fill in a space with a fixed per-character adjustment.

Multi-line text using "\n" is added in v2.57 and it is used if the *lsp* (line spacing) parameter is set. It sets the additional line space between two lines in 0.125 pixel units.

The *font\_width* parameter allows you to modify, in 0.125 units, the aspect ratio of character glyphs per the Visual C++ documentation of the function `CreateFont()`. It is related to the *size* parameter by the default GDI aspect ratio and the natural aspect ratio of the chosen font.

The *font\_angle* parameter allows you to modify the baseline angle of text in 0.1 degree increments anti-clockwise.

The *interlaced* parameter when enabled reduces flicker from sharp fine vertical transitions on interlaced displays. It does this by increasing the window for the anti-aliaser to include 0.5 of the pixel weight from the lines above and below, it effectively applies a mild vertical blur.

The short form (with all default parameters) form is useful when you don't really care what the subtitle looks like as long as you can see it—for example, when you're using [StackVertical](#) and its ilk to display several versions of a frame at once, and you want to label them to remember which is which.

This filter is used internally by AviSynth for the [Version](#) command and for reporting error messages, and the subtitling apparatus is also used by [ShowFrameNumber](#) and friends.

### Default Values

<i>clip</i>	last
<i>text</i>	no default, must be specified
<i>x</i>	8 if align=1,4,7 or none; -1 if align=2,5,8; or width-8 if align=3,6,9
<i>y</i>	height-1 if align=1,2,3; size if align=4,5,6 or none; or 0 if align=7,8,9
<i>first_frame</i>	0
<i>last_frame</i>	framecount(clip)-1
<i>font</i>	"Arial"
<i>size</i>	18.0
<i>text_color</i>	\$00FFFF00 <full opaque yellow>
<i>halo_color</i>	0 <full opaque black>
<i>align</i>	Normally 4 <left and baseline>; if x=-1, then 5 <horizontal center and baseline>
<i>spc</i>	0 <font spacing unchanged>



## Avisynth 2.5 Selected External Plugin Reference

<i>lsp</i>	<multiline is not enabled>
<i>font_width</i>	0 <system default>
<i>font_angle</i>	0.0 degrees
<i>interlaced</i>	false

### Examples

```
# Some text in the center of the clip:
Avisource("D:\clip.avi")
Subtitle("Hello world!", align=5)

# Some text in the upper right corner of the
# clip with specified font, size and color red:
Avisource("D:\clip.avi")
Subtitle("Hello world!", font="georgia", size=24, \
        text_color=$ff0000, align=9)

# Prints text on multiple lines
# without any text halo border.
BlankClip()
Subtitle( \
    "Some text on line 1\nMore text on line 1\n" + \
    "Some text on line 2", \
        lsp=10, halo_color=$ff000000)

# It results in:
Some text on line 1\nMore text on line 1
Some text on line 2
```

### Version Specific Information

v2.58	Added font_width, font_angle, interlaced and alpha color blending.
v2.57	Added multi-line text and line spacing parameter.
v2.07	Added align and spc parameters. Setting y=-1 calculates vertical center (alignment unaffected) Default x and y values dependent on alignment (previously x=8, y=size)
v1.00	Setting x=-1 uses horizontal center and center alignment (undocumented prior to v2.07)

⌘Date: 2009/10/11 11:43:32 ⌘

## 14.59 Subtract

Subtract (*clip1*, *clip2*)

Subtract produces an output clip in which every pixel is set according to the difference between the corresponding pixels in *clip1* and *clip2*. More specifically, it sets each pixel to (50% gray) + (*clip1* pixel) – (*clip2* pixel). You can use [Levels](#) afterwards if you want to increase the contrast.

Also see [here](#) for the resulting clip properties.

### Examples:

```
# Make the differences between clip1 and clip2 blatantly obvious
Subtract(clip1, clip2).Levels(127, 1, 129, 0, 255)
```

If you want to see the deltas between adjacent frames in a single clip, you can do it like this:

```
Subtract(clip.Trim(1,0), clip)
```

### About offset of luma range:

For YUV formats the valid Y range is from 16 to 235 inclusive and subtract takes this into account. This means that the following script

```
Subtract(any_clip, any_clip)
```

will result in a grey clip with luma = 126. For those that require a subtract function for pc\_range YUV data use [Overlay](#):

```
#Overlay(any_clip, any_clip, mode="Difference", pc_range=true) # grey clip with luma = 128
Overlay(clip1, clip2, mode="Difference", pc_range=true)
```

**\$Date: 2006/09/27 18:41:25 \$**

## 14.60 SuperEQ

SuperEQ (*clip*, *string filename*)

SuperEQ (*clip*, *float band1* [, *float band2*, ..., *float band18*])

Shibatch Super Equalizer is a graphic and parametric equalizer plugin for winamp. This plugin uses 16383th order FIR filter with FFT algorithm. Its equalization is very precise. Equalization setting can be done for each channel separately. SuperEQ is originally written by [Naoki Shibata](#).

SuperEQ requires a [foobar2k](#) equalizer setting file. The equalizer can be found in foobar's DSPManager, and settings are adjused and saved from there as well.

Some preset settings can be downloaded from [here](#).

In v2.60, a custom band setting (band1, band2, ...) is added to allow all 18 bands to be set within your script (instead of within your preset file). The values should be specified in decibels (float).

## Avisynth 2.5 Selected External Plugin Reference

This plugin is optimized for processors which have cache equal to or greater than 128k bytes (16383\*2\*sizeof(float) = 128k). This plugin won't work efficiently with K6 series processors (buy Athlon!!!).

### Parameter:

filename	The foobar2k equalizer preset file to apply.
band1–band18	Allows to set all bands within your script (in dB).

### Example:

Apply "Loudness" filter from the Equalizer Presets above:

```
SuperEq("C:\Equalizer Presets\Loudness.feq")
```

### Changelog:

v2.60	Added custom band setting to allow all 16 bands to be set from script.
v2.54	Initial Release

Some parts of SuperEQ are:

Copyright © Naoki Shibata

Other parts are:

Copyright © 2001–2003, Peter Pawlowski

All rights reserved.

**\$Date: 2009/09/12 15:10:22 \$**

## 14.61 SwapUV / UToY / VToY / YToUV

SwapUV (*clip*)

UToY (*clip*)

VToY (*clip*)

YToUV (*clipU*, *clipV* [, *clipY*])

These four filters are available starting from v2.5.

SwapUV swaps chroma channels (U and V) of a clip. Sometimes the colors are distorted (faces blue instead of red, etc) when loading a DivX or MJPEG clip in AviSynth v2.5, due to a bug in DivX (5.00–5.02). You can use this filter to correct it.

UToY copies chroma U plane to Y plane, the resulting image is now half as big. All color (chroma) information is removed, so the image is now greyscale.

Likewise VToY copies chroma V plane to Y plane, the resulting image is now half as big. All color (chroma) information is removed, so the image is now greyscale.

YToUV puts the luma channels of the two clips as U and V channels. Image is now twice as big, and luma is 50% grey. Use [MergeLuma](#), if you want to add luma values.

Starting from v2.51 there is an optional argument *clipY* which puts the luma channel of this clip as the Y

channel.

Starting from v2.53 they also work in YUY2.

```
# Blurs the U chroma channel
video = Colorbars(512, 512).ConvertToYV12
u_chroma = UToY(video).blur(1.5)
YtoUV(u_chroma, video.VToY)
MergeLuma(video)
```

\$Date: 2006/12/15 19:29:25 \$

## 14.62 SwapFields

SwapFields (*clip*)

The SwapFields filter swaps image line 0 with line 1, line 2 with line 3, and so on, thus effectively swapping the two fields in an interlaced frame. It's the same as [SeparateFields.ComplementParity.Weave](#) (and it's implemented that way).

\$Date: 2005/03/24 22:07:09 \$

## 14.63 TCPServer / TCPSource

TCPServer (*clip, int "port"*)

TCPSource (*string hostname, int "port", string "compression"*)

This filter will enable you to send clips over your network. You can connect several clients to the same machine.

### 14.63.1 Syntax

#### 14.63.1.1 Server:

TCPServer (*clip, int "port"*)

This will spawn a server thread on the current machine running on the specified port. Port default is 22050. You will get output in the application you open your script in, but the server will only be running as long as the application (vdub for instance) is open.

Example:

```
Colorbars(512, 256)
TCPServer()
```

will start a server.

### 14.63.1.2 Client:

`TCPSource` (*string hostname, int "port", string "compression"*)

This will connect to the machine with the given address (IP-number for instance) to a server running on the given port. Port default is also 22050 here.

Compression enable you to choose the compression used for the video:

Compression Type	Description
None	Use no compression. Fastest option – video will not be compressed before being sent over the net.
LZO	Use <a href="#">LZO</a> dictionary compression. Fairly fast, but only compresses well on artificial sources, like cartoons and anime with very uniform surfaces.
Huffman	Uses a fairly slow Huffman routine by <a href="#">Marcus Geelnard</a> . Compresses natural video better than LZO.
GZip	Uses a <a href="#">Gzip</a> Huffman only compression. Works much like Huffman setting, but seems faster.

If no compression is given, GZip is currently used by default. Interlaced material compresses worse than non-interlaced due to downwards deltaencoding. If network speed is a problem you might want to use [SeparateFields](#).

Example:

```
TCPSource("127.0.0.1")
Info()
```

This will connect to the local machine, if a server is running.

### 14.63.2 Examples

You can use this to run each/some filters on different PC's. For example:

```
# Clustermember 1:
AVISource
Deinterlacer
TCPServer

# Clustermember 2:
TCPSource
Sharpener
TCPServer

# Clustermember 3:
TCPSource
# client app -> video codec -> final file
```

### 14.63.3 Usability Notes

Once you have added a TCPServer, you cannot add more filters to the chain, or use the output from the filter. The server runs in a separate thread, but since AviSynth isn't completely thread-safe you cannot reliably run multiple servers. This should **not** be used:

```
AviSource("avi.avi")
TCPServer(1001)
TCPServer(1002) # This is NOT a good idea
```

So the basic rule is **never more than one TCPServer per script**.

Using commands after TCPServer is also a bad idea:

```
AviSource("avi.avi")
TCPServer(1001)
AviSource("avi2.avi") # Do not do this, this will disable the server.
```

AviSynth detects that the output of TCPServer isn't used, so it kills the Server filter. **TCPServer should always be the last filter**.

### 14.63.4 Changelog

v2.55	Initial Release
-------	-----------------

\$Date: 2006/01/02 14:51:17 \$

## 14.64 TimeStretch

TimeStretch (*clip*, float "*tempo*", float "*rate*", float "*pitch*", int "*sequence*", int "*seekwindow*", int "*overlap*", bool "*quickseek*", int "*aa*")

TimeStretch allows changing the sound *tempo*, *pitch* and playback *rate* parameters independently from each other, i.e.:

- Sound *tempo* can be increased or decreased while maintaining the original pitch.
- Sound *pitch* can be increased or decreased while maintaining the original tempo.
- Change playback *rate* that affects both tempo and pitch at the same time.
- Choose any combination of tempo/pitch/rate.

#### Parameters:

The speed parameters are percentages, and defaults to 100. If *tempo* is 200 it will play twice as fast, if it is 50, it will play at half the speed. Adjusting *rate* is equivalent to using [AssumeSampleRate](#) and [ResampleAudio](#).

The time-stretch algorithm has a few parameters that can be tuned to optimize sound quality for certain applications. The current default parameters have been chosen by iterative if-then analysis (read: "trial and error") to obtain the best subjective sound quality in pop/rock music processing, but in applications processing different kind of sound the default parameter set may result in a sub-optimal result.

The time-stretch algorithm default parameter values are

## Avisynth 2.5 Selected External Plugin Reference

Sequence	82
SeekWindow	28
Overlap	12

These parameters affect the time-stretch algorithm as follows:

- **Sequence:** This is the length of a single processing sequence in milliseconds which determines how the original sound is chopped in the time-stretch algorithm. Larger values mean fewer sequences are used in processing. In principle a larger value sounds better when slowing down the tempo, but worse when increasing the tempo and vice versa.
- **SeekWindow:** The seeking window length in milliseconds is for the algorithm that searches for the best possible overlap location. This determines from how wide a sample "window" the algorithm can use to find an optimal mixing location when the sound sequences are to be linked back together.

The bigger this window setting is, the higher the possibility of finding a better mixing position becomes, but at the same time large values may cause a "drifting" sound artifact because neighboring sequences may be chosen at more uneven intervals. If there's a disturbing artifact that sounds as if a constant frequency was drifting around, try reducing this setting.

- **Overlap:** The overlap length in milliseconds. When the sound sequences are mixed back together to form a continuous sound stream again, this parameter defines how much of the ends of the consecutive sequences will be overlapped.

This shouldn't be that critical parameter. If you reduce the *Sequence* setting by a large amount, you might wish to try a smaller value on this.

- **QuickSeek:** The time-stretch routine has a 'quick' mode that substantially speeds up the algorithm but may degrade the sound quality.
- **aa:** Controls the number of tap the Anti-alias filter uses for the rate changer. Set to 0 to disable the filter. The value must be a multiple of 4.

The table below summarizes how the parameters can be adjusted for different applications:

Parameter name	Default value magnitude	Larger value affects...	Smaller value affects...	Music	Speech	Effect in CPU burden
Sequence	Default value is relatively large, chosen for slowing down music tempo	Larger value is usually better for slowing down tempo. Growing the value decelerates the "echoing" artifact when slowing down the tempo.	Smaller value might be better for speeding up tempo. Reducing the value accelerates the "echoing" artifact when slowing down the tempo	Default value usually good	A smaller value than default might be better	Increasing the parameter value reduces computation burden
SeekWindow	Default value is relatively large, chosen for slowing down music tempo	Larger value eases finding a good mixing position, but may cause a "drifting" artifact	Smaller reduce possibility to find a good mixing position, but reduce the "drifting" artifact.	Default value usually good, unless a "drifting" artifact is disturbing.	Default value usually good	Increasing the parameter value increases computation burden
Overlap	Default value is relatively large, chosen to		If you reduce the "sequence ms" setting, you might			Increasing the parameter value increases

## Avisynth 2.5 Selected External Plugin Reference

	suit with above parameters.		wish to try a smaller value.			computation burden
--	-----------------------------	--	------------------------------	--	--	--------------------

### Notes:

- This is NOT a sample exact plugin. If you use it, slight inaccuracies might occur. Since we are dealing with float values rounding errors might occur, especially on large samples. In general however inaccuracies should not exceed a few 10's of milliseconds for movielength samples.
- Currently the SoundTouch library only supports 1 and 2 channels. When used with more than 2 channels, each channel is processed individually in 1 channel mode. This will destroy the phase relationship between the channels. See this thread for details :- [TimeStretch in AVISynth 2.5.5 Alpha - Strange stereo effects ?](#)
- SoundTouch is used in float sample mode.

### Examples:

```
TimeStretch(pitch = 200)
```

This will raise the *pitch* one octave, while preserving the length of the original sample.

```
TimeStretch(pitch = 100.0*pow(2.0, 1.0/12.0))
```

This will raise the *pitch* one semi-tone, while preserving the length of the original sample.

```
TimeStretch(tempo = (100.0*25.0*1001.0)/24000.0)
```

This will change the *tempo* from Film speed to PAL speed without changing the pitch.

### Credits:

This function uses:

SoundTouch library Copyright (c) Olli Parviainen 2002-2006

<http://www.iki.fi/oparvi/i/soundtouch>

<http://www.surina.net/soundtouch>

### Changelog:

v2.55	Initial Release
v2.57	Expose soundtouch parameters

\$Date: 2008/12/24 22:55:01 \$

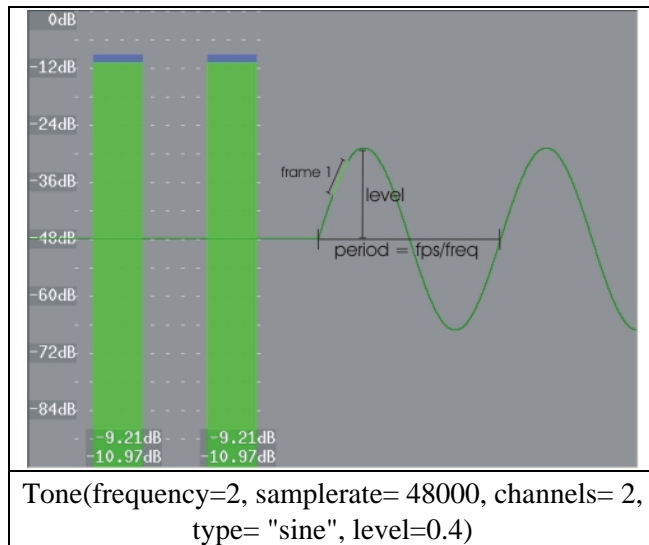


## 14.65 Tone

Tone (*float "length", float "frequency", int "samplerate", int "channels", string "type", float "level"*)

This will generate sound (a waveform) at a given *frequency* for a given *length* of time in seconds. *Type* can be "Silence", "Sine" (default), "Noise", "Square", "Triangle" or "Sawtooth". *level* is the amplitude of the waveform (which is maximal if level=1.0).

Defaults are Tone(10.0, 440, 48000, 2, "sine", 1.0).



In the figure above, a sinus is generated (on a grey clip with framerate 24 fps). The period of the waveform (in frames) is the framerate divided by *frequency* (or fps/freq, which is 24/2=12 frames in our example). The part of the graph which is light-green represents all samples of the frame under consideration (which is frame 1 here). The number of samples in a particular frame is given by the *samplerate* divided by the framerate (which is 48000/24 = 2000 samples in our example). (Note that the bars are made with [Histogram](#) and the graph with the [AudioGraph](#) plugin.)

More generally, the waveform above is described by

$$g(n,s) = \text{level} * \sin(2*\pi*(\text{frequency}*n/\text{framerate} + s*\text{frequency}/\text{samplerate}))$$

with "n" the frame and "s" the sample under consideration (note that s runs from 0 to samplerate/framerate – 1).

In the example above, this reduces to

$$g(n,s) = 0.4 * \sin(2*\pi*(2*n/24 + s*2/48000))$$

with "n" the frame and "s" the sample under consideration (note that s runs from 0 to 1999).

### Changes:

v2.54	Initial release.
v2.56	Added level.

\$Date: 2007/07/13 00:53:01 \$

## 14.66 Trim

Trim(*clip*, *int first\_frame*, *int last\_frame* [, *bool pad\_audio*])

Trim trims a video clip so that it includes only the frames *first\_frame* through *last\_frame*. The audio is similarly trimmed so that it stays synchronized. If you pass 0 for *last\_frame* it means "end of clip." A negative value of *last\_frame* returns the frames from *first\_frame* to *first\_frame* + (- *last\_frame*-1). This is the only way to get the very first frame!

*pad\_audio* (default true) causes the audio stream to be padded to align with the video stream. Otherwise the tail of a short audio stream is left so. When *last\_frame*=0 and *pad\_audio*=false the end of the two streams remains independent.

```

Trim(100,0)           # delete the first 100 frames, audio padded
                    # or trimmed to match the video length.
Trim(100,0,false)    # delete the first 100 frames of audio and video,
                    # the resulting stream lengths remain independent.
Trim(100,-100)       # is the same as trim(100,199)
Trim(100,199,false) # audio will be trimmed if longer but not
                    # padded if shorter to frame 199
Trim(0,-1)           # returns only the first frame

```

### Changelog:

v2.56	added pad audio
-------	-----------------

\$Date: 2008/10/26 14:18:27 \$

## 14.67 TurnLeft / TurnRight / Turn180

TurnLeft (*clip*)  
 TurnRight (*clip*)  
 Turn180 (*clip*)

TurnLeft rotates the clip 90 degrees counterclock wise, and  
 TurnRight rotates the clip 90 degrees clock wise.  
 Turn180 rotates the clip 180 degrees.

\$Date: 2004/07/06 04:52:52 \$

## 14.68 Tweak

Tweak(*clip*, *float "hue"*, *float "sat"*, *float "bright"*, *float "cont"*, *bool "coring"*, *bool "sse"*, *float "startHue"*, *float "endHue"*, *float "maxSat"*, *float "minSat"*, *float "interp"*)

This function provides the means to adjust the hue, saturation, brightness, and contrast of a video clip. In v2.58, both the saturation and hue can be adjusted for saturations in the range [*minSat*, *maxSat*] and hues in the range [*startHue*, *endHue*]. *interp* interpolates the adjusted saturation to prevent banding.

## Avisynth 2.5 Selected External Plugin Reference

*Hue*: (−180.0 to +180.0, default 0.0) is used to adjust the color hue of the image. Positive values shift the image towards red. Negative values shift it towards green.

*Sat*: (0.0 to 10.0, default 1.0) is used to adjust the color saturation of the image. Values above 1.0 increase the saturation. Values below 1.0 reduce the saturation. Use `sat=0` to convert to grayscale.

*Bright*: (−255.0 to 255.0, default 0.0) is used to change the brightness of the image. Positive values increase the brightness. Negative values decrease the brightness.

*Cont*: (0.0 to 10.0, default 1.0) is used to change the contrast of the image. Values above 1.0 increase the contrast. Values below 1.0 decrease the contrast.

*coring* = true/false (optional; true by default, which reflects the behaviour in older versions) is added. When setting to true is means that the luma (Y) is clipped to [16,235], and when setting to false it means that the luma is left untouched. [Added in v2.53.]

*sse* = true/false (optional; false by default) re-enables the SSE code if required (perhaps an AMD might run it faster). [Added in v2.56.]

*startHue* (default 0), *endHue* (default 360): (both from 0 to 360; given in degrees.). The hue and saturation will be adjusted for values in the range [*startHue*, *endHue*] when `startHue < endHue`. Note that the hue is periodic, thus a hue of 360 degrees corresponds with a hue of zero degrees. If `endHue < startHue` then the range [*endHue*, 360] and [0, *startHue*] will be selected (thus anti-clockwise). If you need to select a range of [350, 370] for example, you need to specify `startHue=370` and `endHue=350`. Thus when using the default values all pixels will be processed.

*maxSat* (default 150), *minSat* (default 0): (both from 0 to 150 with `minSat < maxSat`; given in percentages). The hue and saturation will be adjusted for values in the range [*minSat*, *maxSat*]. Practically the saturation of a pixel will be in the range [0,100] (thus 0–100%), since these correspond to valid RGB pixels (100% corresponds to R=255, G=B=0, which has a saturation of 119). An overshoot (up to 150%) is allowed for non-valid RGB pixels (150% corresponds to U=V=255, which has a saturation of  $\sqrt{127^2+127^2} = 180$ ). Thus when using the default values all pixels will be processed.

*interp*: (0 to 32, default 16) is used to interpolate the adjusted saturation. The interpolation is done in the range [*minSat*−*interp*, *minSat*] and [*maxSat*, *maxSat*+*interp*]. There is no interpolation for `interp=0`, which can be useful when a clip consists of uniform colors. The interpolation is linear.

### 14.68.1 Usage and examples: adjusting contrast and brightness



## Avisynth 2.5 Selected External Plugin Reference



original

There are two problems with this picture. It is too dark, and the contrast is too small (the details of the tree are not visible for example). First, we will increase the brightness to make the picture less dark (left picture below). Second, we will increase the contrast to make details in the dark areas more visible (right picture below). Make sure that the bright parts don't get too bright though.

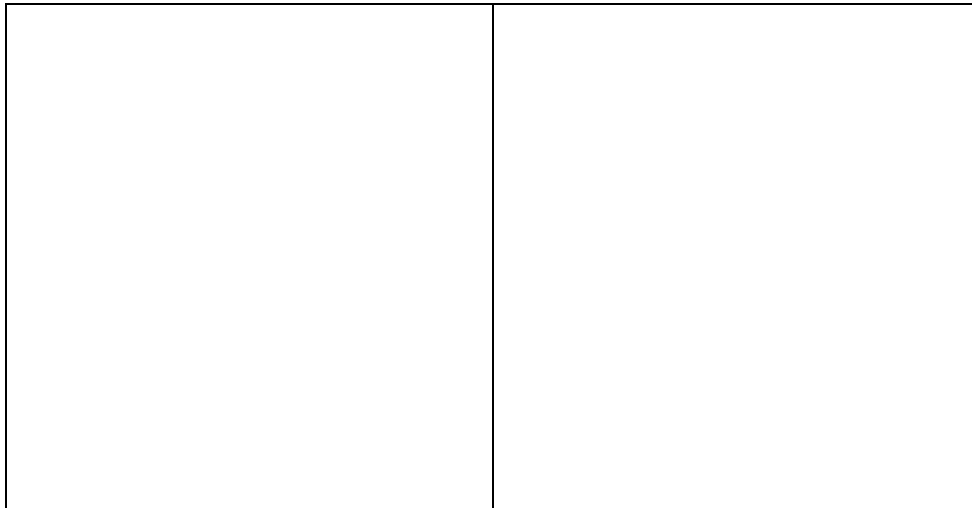


bright=20

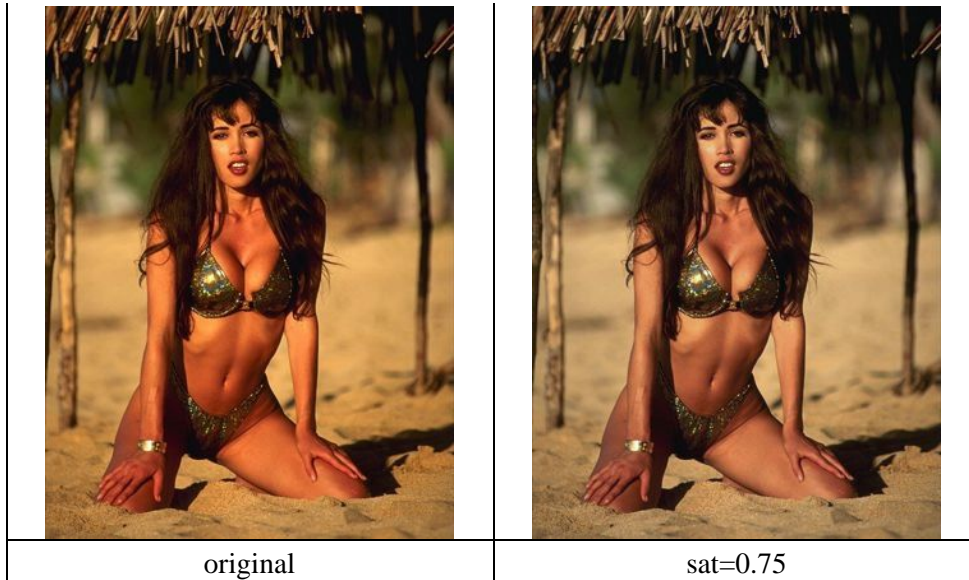


bright=20, cont=1.2

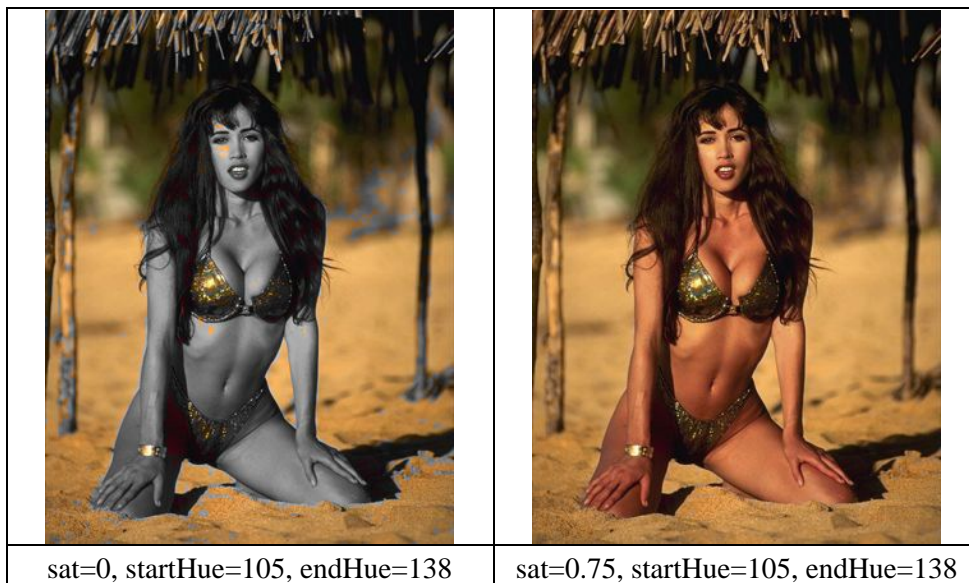
### 14.68.2 Usage and examples: adjusting saturation



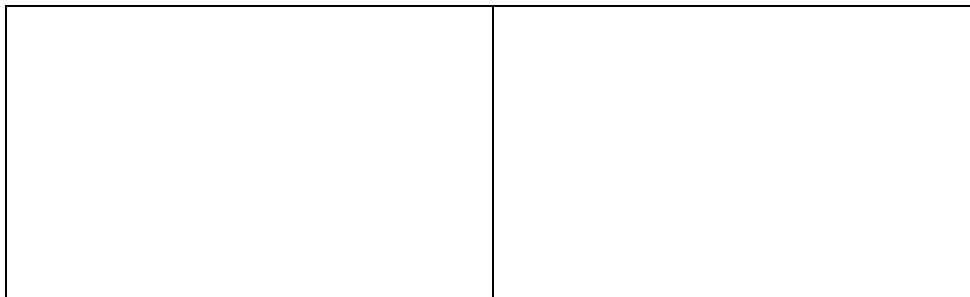
## Avisynth 2.5 Selected External Plugin Reference





Suppose we want to lower the saturation of the skin of the girl, and the background should be left intact. The proper way to do this is to set `sat=0`, and lower the hue range till you found the correct hue range which should be processed. This can be done by using a Vectorscope, but also manually. (If it is needed the saturation range can also be specified if the dark and white parts of that hue should not be processed.) The result is below.



Instead, we can also try to "select" the skin of the girl by narrowing the saturation range only. The result is below. In this case the result is pretty much identical.



	
sat=0, maxSat=75, minSat=55	sat=0.75, maxSat=75, minSat=55

**Changelog:**

v2.56	added sse=true/false to enable sse code
v2.58	added startHue, endHue, maxSat, minSat and interp

\$Date: 2009/09/12 15:10:22 \$

## 14.69 Version

Version()

Version() # [AviSynth 2.51 beta]  
 generates a 468x80 pixel, 24.000fps, 10-second, RGB24-format video clip with a short version and copyright statement in manila 16-point Arial text on 25% grey background.

Version() # [AviSynth 2.08]  
 generates a 420x80 pixel, 24.000fps, 10-second, RGB24-format video clip with a short version and copyright statement in manila 16-point Arial text on 25% grey background.

Version() # [Legacy versions]  
 generates a 512x32 pixel, 15fps, 10-second, RGB-format video clip with a short version and copyright statement in orange 24-point Arial text.

Anyway, a script containing only Version() is the best way to check if AviSynth in principal works.

\$Date: 2004/03/09 21:28:07 \$

## 14.70 Weave

Weave (*clip*)

## Avisynth 2.5 Selected External Plugin Reference

Weave is the opposite of [SeparateFields](#): it takes pairs of fields from the input video clip and combines them together to produce interlaced frames. The new clip has half the frame rate and frame count. Weave uses the frame-parity information in the source clip to decide which field to put on top. If it gets it wrong, use [ComplementParity](#) beforehand or [SwapFields](#) afterwards.

All AviSynth filters keep track of field parity, so Weave will always join the fields together in the proper order. If you want the other order, you'll have to use [ComplementParity](#), [AssumeTFF](#) or [AssumeBFF](#) beforehand or [SwapFields](#) afterwards.

From versions 2.5.6 this filter raises an exception if the clip is already frame-based. You may want to use [AssumeFieldBased](#) to force weave a second time. Prior versions did a no-op for materials that was already frame-based.

\$Date: 2009/09/12 15:10:22 \$

### 14.71 WriteFile / WriteFileIf / WriteFileStart / WriteFileEnd

`WriteFile` (*clip*, *string filename*, *string expression1*, ... , *string expression16*, *bool "append"*, *bool "flush"*)

`WriteFileIf` (*clip*, *string filename*, *string expression1*, ... , *string expression16*, *bool "append"*, *bool "flush"*)

`WriteFileStart` (*clip*, *string filename*, *string expression1*, ... , *string expression16*, *bool "append"*)

`WriteFileEnd` (*clip*, *string filename*, *string expression1*, ... , *string expression16*, *bool "append"*)

`WriteFile` evaluates each *expressionN*, converts the result to a string and puts the concatenated results into a file, followed by a newline.

The "run-time" variable *current\_frame* is set so that you can use it in an "expression" (this works similar as with `ScriptClip`, look there in the docu for more infos).

*current\_frame* is set to -1 when the script is loaded and to -2 when the script is closed.

`WriteFile` evaluates the "expression"s and generates output for each frame rendered by the filter.

`WriteFileIf` is similar, but generates output only if the first expression is true. In both cases, there is no output at script opening or closure. Note that since output is produced only for "rendered" frames, there will be no output at all if the result of the filter is not used in deriving the final result of the script.

`WriteFileStart` and `WriteFileEnd` generate output only on script opening and closure respectively, there is no action on each frame. In both cases, the "expression"s are evaluated exactly once, at the location of the filter in the script.

When *append* = true, the result(s) will be appended to any existing file. The default for *append* is always true, except for `WriteFileStart` (here it is false).

When *flush* = true, the file is closed and reopened after each operation so you can see the result immediately (this may be slower). The default for *flush* (`WriteFile` and `WriteFileIf`) is true. For `WriteFileStart` and `WriteFileEnd`, the file is always closed immediately after writing.

### 14.71.0.1 Usage is best explained with some simple examples:

```
filename = "c:\myprojects\output.txt"
# create a test video to get frames
Version()

# the expression here is only a variable, which is evaluated and put in the file
# you will get a file with the framenummer in each line
WriteFile(filename, "current_frame")

# this line is written when the script is opened
WriteFileStart(filename, "" "This is the header" "")

# and this when the script is closed
WriteFileEnd(filename, "" "Now the script was closed" "")
```

Look how you can use triple-quotes to type a string in a string!

If the expression cannot be evaluated, the error message is written instead.

In case this happens with the If-expression in WriteFileIf the result is assumed to be true.

```
# will result in "I don't know what "this" means"
WriteFile(filename, "this is nonsense")
```

### 14.71.0.2 There are easier ways to write numbers in a file, BUT:

... with this example you can see how to use the "runtime function" AverageLuma:

```
# create a test video to get different frames
Version.FadeIn(50).ConvertToYV12

# this will print the frame number, a ":" and the average luma for that frame
colon = ":"
WriteFile("F:\text.log", "current_frame", "colon", "AverageLuma")
```

Or maybe you want the actual time printed too:

```
# create a test video to get different frames
Version.FadeIn(50).ConvertToYV12

# this will print the frame number, the current time and the average luma for that frame
# the triple quotes are necessary to put quotes inside a string
WriteFile(last, filename, "current_frame", "" time(" %H:%M:%S") "", "AverageLuma")
```

### 14.71.0.3 More examples:

In WriteFileIf the FIRST expression is expected to be boolean (true or false). Only if it is TRUE the other expressions are evaluated and the line is printed. (Remember: && is AND, || is OR, == is EQUAL, != is NOT EQUAL) That way you can omit lines completely from your file.

```
# create a test video to get different frames
Version.FadeIn(50).ConvertToYV12

# this will print the frame number, but only of frames where AverageLuma is between 30 and 60
WriteFileIf(last, filename, "(AverageLuma>30) && (AverageLuma<60)", "current_frame", "" ":" "
```



\$Date: 2009/10/11 11:43:32 \$

## 14.72 AviSynth License

AviSynth v2.5 Copyright © 2002–2006 Ben Rudiak-Gould et al.  
<http://www.avisynth.org>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA, or visit <http://www.gnu.org/copyleft/gpl.html>

Linking Avisynth statically or dynamically with other modules is making a combined work based on Avisynth. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holders of Avisynth give you permission to link Avisynth with independent modules that communicate with Avisynth solely through the interfaces defined in `avisynth.h`, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that every copy of the combined work is accompanied by a complete copy of the source code of Avisynth (the version of Avisynth used to produce the combined work), being distributed under the terms of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on Avisynth, such as 3rd-party filters, import and export plugins, or graphical user interfaces.

See [full text](#) of GNU General Public license (GPL) version 2.

See copyright notices of Avisynth developers in supplied source codes for details.

See AviSynth external Filter SDK license [text](#) (link valid if installed).

### 14.72.1 AviSynth documentation license

AviSynth documentation is Copyright © 2002–2007 AviSynth developers and contributors.

Starting from 5 August 2007 the following documentation is released under the [Creative Commons Attribution–ShareAlike 3.0 License](#) (abbreviated by "CC BY–SA 3.0", see also [full license terms](#)):

- All content of the AviSynth Wiki at [avisynth.org](http://avisynth.org), except the AviSynth Filter SDK.
- The off–line documentation which is distributed with AviSynth itself, except any external plugin documentation which is copyright corresponding authors (see description of these plugins for their license terms).

## Avisynth 2.5 Selected External Plugin Reference

When editing this Wiki or using material from this Wiki or off-line documentation you are bound by the terms of this license, and these terms are briefly as follows:

You are allowed to copy, modify or improve the content of the AviSynth Wiki and the off-line documentation provided that:

- **Share Alike (SA):** The resulting changes are licensed under the same or a compatible license as the AviSynth Wiki and the off-line documentation (which is "CC BY-SA 3.0").
- **Attribution (BY):** Permit users to copy, distribute, display and perform the work and make derivative works based upon it only if they give the author or licensor the credits in the manner specified by these. This implies that if you distribute parts of the documentation (such as articles or scripts) your should refer to "AviSynth documentation Wiki" (or "AviSynth documentation") with link to these pages or to the main site: avisynth.org. You must also keep attributions for the parts attributed to other resources or authors.

Besides above two requirements, you also agree that:

- attribution of your contributions will (or may) be given to AviSynth project in a form "AviSynth documentation" or "AviSynth documentation Wiki" (avisynth.org).
- your contribution (and main Wiki content) may be relicensed by the AviSynth developers team under a compatible license.
- you are also promising us that you wrote the contributions yourself, or copied it from public domain, or from work with a compatible license. **DO NOT SUBMIT COPYRIGHTED WORK WITHOUT PERMISSION!**

Note that this license has a fair-use clause (article 2):

*"2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws."*

which, in our opinion, explicitly permits you to post and discuss scripts (from the documentation) where ever you want without the need to mention its license or its authors, but it is civil to credit the developers and script authors. Of course, you may not declare yourself as its author or distribute the documentation under an incompatible license.

AviSynth documentation translations to other languages (both at the wiki and off-line) should be under same or compatible license terms.

### 14.72.2 AviSynth logo

'Film and Gears' logo was contributed to AviSynth project by doom9 forum member 'Shayne'.

'Tray and AviSynth' logo (now at old wiki pages) was contributed to AviSynth project by anonymous sh0dan's friend.

Permission to use and/or modify these logos is granted provided that your materials are related to AviSynth and you give an attribution to avisynth.org.

### 14.72.3 Avisynth uses codes from following projects:

Avisynth's frameclient and OpenDML code, part of the subtitling code, CPUExtensions code, VirtualDub plugins interface were taken from:  
VirtualDub - Video processing and capture application  
Copyright © 1999-2001 Avery Lee  
<http://www.virtualdub.org>  
VirtualDub is distributed under terms of GNU GPL.  
Special permission by Avery Lee to distribute existing (at November 2006) VirtualDub-derived elements in Avisynth under the current Avisynth license, which is GPL plus a special exemption.

The DirectShowSource filter was adapted from parts of  
bbMPEG by Brent Beyeler, 1999-2000  
<http://members.home.net/beyeler/bbmpeg.html> (old link)  
<http://members.cox.net/beyeler/bbmpeg.html> (new link)  
bbMPEG is released as freeware. Freely distributable.

Avisynth C interface  
Copyright © 2003-2004 Kevin Atkinson  
[http://kevin.atkinson.dhs.org/avisynth\\_c/](http://kevin.atkinson.dhs.org/avisynth_c/)  
Distributed under terms of GNU GPL with a special exception:  
As a special exception, I give you permission to link to the Avisynth C interface with independent modules that communicate with the Avisynth C interface solely through the interfaces defined in `avisynth_c.h`, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that every copy of the combined work is accompanied by a complete copy of the source code of the Avisynth C interface and Avisynth itself (with the version used to produce the combined work), being distributed under the terms of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on Avisynth C Interface, such as 3rd-party filters, import and export plugins, or graphical user interfaces.

ImageLib (DevIL) image processing library  
(v1.6.6; used in ImageRead, ImageWrite)  
Copyright © 2000-2002 by Denton Woods  
<http://openil.sourceforge.net/>  
Distributed under terms of the GNU Lesser Public License (LGPL).

See [full text](#) of GNU LGPL for license terms of such libraries.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA, or visit <http://www.gnu.org/copyleft/lgpl.html>

Corresponding source codes are supplied.

SoftWire class library (v4.4.1; used in Limiter and image resiser)  
Copyright © 2002-2003 Nicolas Capens  
<http://softwire.sourceforge.net/> (closed in October 2005)  
It was distributed under terms of the GNU LGPL with following notice:  
If you only derive from a class to write your own specific implementation, you don't have to release the source code of your whole project, just give credit where due.

## Avisynth 2.5 Selected External Plugin Reference

SoundTouch audio processing library (v1.3.1 or other; used in TimeStretch)  
Copyright © 2002-2006 Olli Parviainen  
<http://www.surina.net/soundtouch>  
Distributed under terms of the GNU LGPL.

Audio super equalizer and sampling rate converter are based on:  
Shibatch Super Equalizer (SuperEQ) and Sampling Rate Converter (SSRC)  
Copyright © 2001-2003 Naoki Shibata  
<http://shibatch.sourceforge.net/>  
Both are distributed under terms of GNU LGPL (except FFT part).

Some changes are:  
Copyright © 2001-2003, Peter Pawlowski  
<http://www.foobar2000.org>  
(with addition of PFC library)

Other changes are:  
Copyright © 2003, Klaus Post

PFC class library (portion of Foobar2000 0.7 SDK)  
Copyright © 2001-2003, Peter Pawlowski  
All rights reserved.

Used Foobar2000 SDK was distributed under following conditions:  
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions are  
met: Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.  
Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.  
Neither the name of the author nor the names of its contributors may be  
used to endorse or promote products derived from this software without  
specific prior written permission.  
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS  
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED  
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A  
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR  
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

FFT part of SuperRQ and SSRC is based on General Purpose FFT package  
<http://momonga.t.u-tokyo.ac.jp/~ooura/fft.html>

It is originally distributed under following license terms:  
Copyright © 1996-2001 Takuya OOURA  
(Email: [ooura@kurims.kyoto-u.ac.jp](mailto:ooura@kurims.kyoto-u.ac.jp) or [ooura@mmm.t.u-tokyo.ac.jp](mailto:ooura@mmm.t.u-tokyo.ac.jp))  
You may use, copy, modify and distribute this code for any purpose  
(include commercial use) and without fee.  
Please refer to this package when you modify this code.

### **[14.72.4 Thanks also to everyone else contributing to the AviSynth project!](#)**

§Date: 2007/08/16 05:09:08 §