VCB-Studio 教程 18　YUV 与 RGB 的互转(1)

本教程旨在讲述 avs 中 RGB 相关的知识。

1. avs 中视频的类型

我们在 avs 中，可以载入和编辑视频，但是视频格式各不相同，所以 avs 中需要对它们进行归类。avs 中原生支持 YUV 8bit 的视频和 RGB8bit 的视频，分别有以下类型：

YV12， 即 YUV420p8，UV 的分辨率是 Y 的 $\frac{1}{2}\times\frac{1}{2}$

YV16， 即 YUV422p8，UV 的分辨率是 Y 的 $\frac{1}{2}\times 1$

YV24， 即 YUV444p8，UV 的分辨率和 Y 完全相同。

YUY2， 即 YUV422p8 另一种组织格式。

Y8， 单独记录黑白视频，只有 Y 通道。这玩意很适合拆分 YUV/RGB 平面的时候使用。

RGB24，RGB 各 8 个 bit。

RGB32，在 RGB24 的基础上加上 8bit 的 alpha（透明度）通道。

这些是 avs 原生支持的视频格式。avs 中的原生滤镜，比如 Spline36Resize，就支持对这些格式进行处理。处理完的结果自然也是原生格式：

```
LWLibavVideoSource("00000.m2ts",threads=1) #YV12
Spline36Resize(1280,720) #YV12
```

类似 Fraps 这类，编码的 avi 是以 RGB 组织的，读入的时候就会是 RGB24:

```
AVISource("fraps.avi") #RGB24
Spline36Resize(1280,720) # avs 自带的 resizer 都支持 RGB 下做重采样
ConvertToYV12(matrix="Rec.709") #转为 YV12
```

当然，你也可以先转为 YV12 再做 resize:

```
AVISource("fraps.avi") #RGB24
ConvertToYV12(matrix="Rec.709") #转为 YV12
Spline36Resize(1280,720)
```

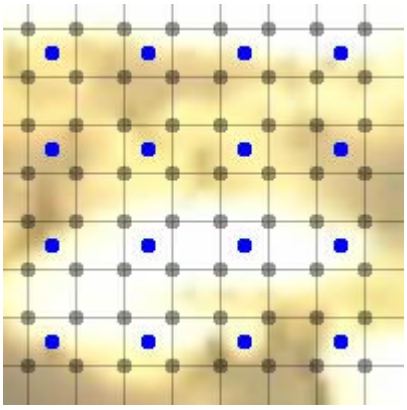
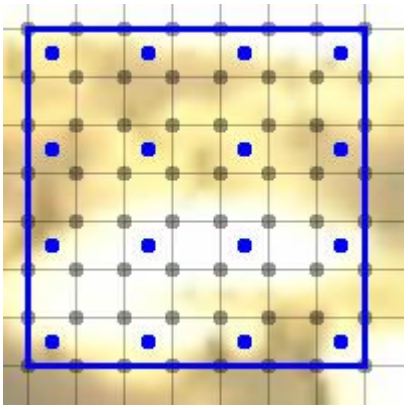
对于 RGB 源，除非成品为了兼容性因素必须做成 YUV420，一般都推荐保留 YUV444. 那么可以这么写：

```
AVISource("fraps.avi") #RGB24
Spline36Resize(1280,720) # avs 自带的 resizer 都支持 RGB 下做重采样
ConvertToYV24(matrix="Rec.709") #转为 YV24
```

x264 编码参数中加入 --input-csp "i444" --output-csp "i444"，表示使用 YUV 4:4:4 编码。x265 只需要指定--input-csp "i444"；因为 x265 不会自己做转换，给什么出什么。

2. YUV 模型中 Chroma 的那些事儿： Chroma Placement (cplace)

我们在科普中讲过，YUV 模型中，因为 chroma subsampling 的使用，绝大多数情况下 Chroma 是被缩水的。最常见的就是缩水成 $\frac{1}{2}\times\frac{1}{2}$：

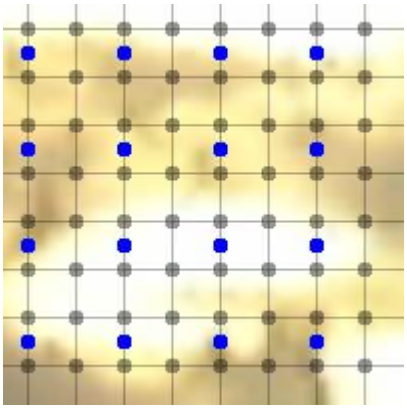
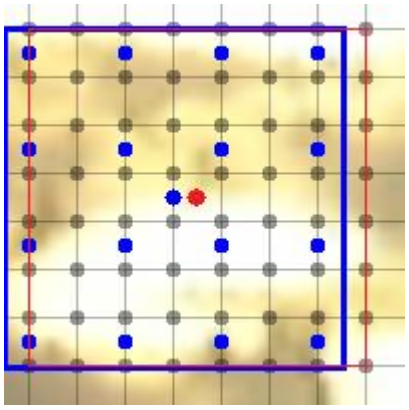



如图所示，图中每个黑点代表一个 Y，每个蓝点代表一对 UV。一共是 4 个 Y 对应 1 对 UV。

图上的这种表示方法，每对 UV 的采样位置，正好在 4 个 Y 的中央，横纵都是如此。这种对齐方式，叫做中央对齐 (center align)。这种对齐的特点是，将 chroma 放大到 2x2 倍之后，图像的中心和 Y 的中心是重合的（见右图，右图是将 UV 放大到 Y 的尺寸后，图像范围）

这种采样位置被称为 MPEG 1 Placement

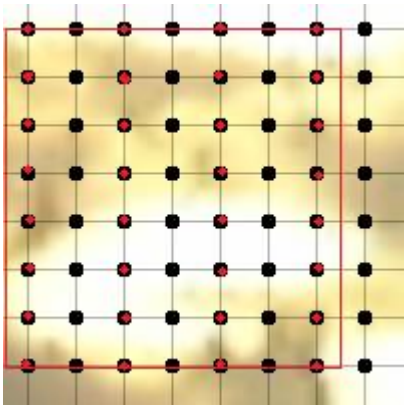
问题是，MPEG2 时代，规则就变了。UV 的位置，从上下看，依旧是在中间，但是从左右看，UV 的位置靠左（即上下保持 center align，左右改为 left align）：

如果直接把 UV 放大到 Y 的尺度（我们常用的 resizer 都是 center align 的，这意味着图像放大缩小后，图像的中心点是对齐的），会发现图像相对于 Y 是有错位的。放大后的错位量是向左溢出了 0.5 个像素，右边亏缺 0.5 像素。图像中细红框表示 Y 覆盖的范围，粗蓝框表示 UV 放大后覆盖的范围。

两个矩形，中心点的距离是 0.5 像素（蓝点和红点间的距离，以 luma 的尺度算。如果按照 chroma 的尺度是 0.25 像素）

这是 YUV 420 的表示方法。YUV 422 的表示方法如下(为了显眼，用红色点表示 UV 位置)：



可以发现，YUV422 是纵向上保留全高度的 UV，横向宽度上只有 1/2. 横向上依旧是左靠齐（如果是 MPEG1 的中靠齐，那么红点的位置就是在相邻两个黑点的中点上）
YUV422 依旧有着 left align 下，upscale 后中心不对其的问题，而且也是向左溢出 0.5 个像素。

MPEG2 的左对齐优势在于，处理插补的时候，可以保留一半插补另一半。以上图 YUV422 的示意图为例，如果要将 UV 分辨率拉升到 Y 的分辨率，所有红点保留，然后设法插补出剩下的一半（就是图中空白黑点所在的位置），就完成了拉升。

3. 利用 Resizer 做 YUV 不同 subsampling 之间的转换

给你一个 YV24 的视频，如何帮它转化为 YV12 呢？
如果是 8bit 和 8bit 互转，只需要用 avs 自带的 ConvertTo 命令：
ConvertToYV12()
同理，还有 ConvertToYV24, ConvertToYV16 这种。
几个参数设置：
ChromaInplacement：输入的 cplace，默认”MPEG2”，可选”MPEG1”
ChromaOutplacement：输出的 cplace，默认”MPEG2”，可选”MPEG1”
chromaresample：chroma 放大缩小算法。默认”bicubic”,可选类似”lanczos”,”spline36”等。

比如一个 YV24 的视频，我们想转换为 cplace 为 MPEG2 的 YV12，用 spline36 做 UV 的 downscale:
ConvertToYV12(chromaresample=”spline36”, ChromaOutplacement=”MPEG2”)
注意 ChromaOutplacement=”MPEG2”可以省略，因为是默认值

如果丢给你的是 16bit 的视频呢？
这就要求我们自己写转换了。

首先介绍 YUV 平面拉出来的几个工具：
UtoY8()/UtoY()，把 U 平面拉出来做成一个 Y8/YV12。如果输出 YV12，生成的 clip UV 都是空的。分辨率是实际 U 平面的分辨率。比如一个 1080p 的 YUV420 视频，UtoY8 得到的是一个 960x540 的 Y8。
VtoY8()/VtoY ()，同上。
ConvertToY8()，就是把 UV 平面都丢了，只保留 Y 平面。
YtoUV(A,B,C) 生成一个全新的 YUV 视频，Y 平面是 C 的 Y，U 平面是 A 的 Y，V 平面是 B 的 Y。常用于把处理完毕的 YUV 平面合并。AB 的 Y 平面必须分辨率相同（因为一个视频的 UV 平面肯定是等分辨率），合并后，根据 Y 平面和 UV 平面的分辨率决定是 YV12/16/24.

现在，假设我们手上有个 1080p stacked 16bit YV24 的视频,叫做 src16，我们想把它转换为 stacked 16bit YV12, MPEG1 格式:

U = src16.UtoY8 (). Dither_resize16(960,540) #U 平面单独拿出来放一个 Y8，然后 downscale
V = src16.VtoY8(). Dither_resize16(960,540) #V 平面一样。产生的 Y8 平面依旧是 stacked 16bit
YtoUV(U,V,src16) #Y 平面直接从 src16 的 Y 拿就行了。

vs 的写法是：

U = core.std.ShufflePlanes(src16,1,vs.GRAY)
U = core.fmtc.resample(U,960,540)
V = …
res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

MPEG1 的 cplace 是中心对齐的，而一般的 resizer 默认也是中心对齐的，所以可以直接 downscale，没有问题。
最后一步，因为将一对 960x540 的 UV 和 1920x1080 的 Y 对齐，系统自动判断为 YV12。

vs 还可以用自带的 Resize（用的 zimg library）：
res = core.resize.Spline36(src16, format=vs.YUV420P16, chromaloc=center)
注意 chromaloc=center 不能省略，因为默认是 left，即按照 MPEG2 的来

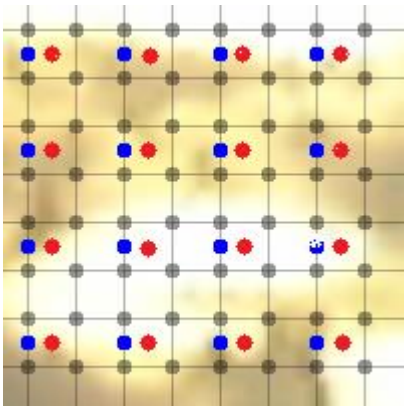
如果我们要将一个 cplace 为 MPEG1 的 YV12 转换为 YV24，方法也是一样：
U = src16.UtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4)
V = src16.VtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4)
YtoUV(U,V,src16) # UV 用 softcubic 60 转换到 1080p 分辨率，然后和 Y 合并。

用 vs 自带的你得这么写：
res = core.resize.Bicubic(src16, format=vs.YUV444P16, chromaloc_in=center, filter_param_a_uv=0.6, filter_param_b_uv=0.4)
注意输出的 chromaloc 是不需要指定的，YUV444 不存在 cplace 一说。

如果给你的是 MPEG2 的 YV12 呢？
我们提到过，MPEG2 的 cplace，直接将 UV 平面拉升，会有向左 0.5 像素的溢出。
所以拉升之后，还需要将视频左边切割掉 0.5 的像素，同时右边增补 0.5 的像素。
一种做法是，resizer 之后再接一个 resizer 做修正：
U = src16.UtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4)
U = U. Dither_resize16(1920,1080,src_left=0.5) #左边切割掉 0.5 像素，同时右边填补 0.5 像素以保持 1080p 的分辨率
V = src16.VtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4)
V = V. Dither_resize16(1920,1080,src_left=0.5)
YtoUV(U,V,src16)

但是这种做法并非最好的。理由是我们做了一次 resizer，又做了一次 shift。重复计算，对精度损失和效率没有好处。能不能拉升之前就做 shift 呢？答案是肯定的：

我们只要把采样点位于蓝点处的 UV，在红点处重采样就可以，或者说，原来 UV 的数值是蓝点处的 UV，我们要让 UV 数值变成红点处的 UV。



这个重采样的操作，相当于将 UV 平面左边，切掉一块宽度为红蓝点距离的图像。红蓝点距离，相对于 Y 的范围来说，是 1/2 像素，但是对于 UV 平面来讲，这个距离是相邻两个 UV 间距离的 1/4，即 0.25 个像素。换言之，upscale 之前，我们要把 UV 平面左边切掉 0.25 个像素，右边插补 0.25 像素：
U = src16.UtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4, src_left=0.25)
V = src16.VtoY8(). Dither_resize16(1920,1080,kernel="bicubic",a1=0.6,a2=0.4, src_left=0.25)
YtoUV(U,V,src16)

如果拉升后修复,src_left=0.5；如果拉升之前处理, src_left=0.25.
一般实际操作中，我们选择拉升之前就做处理。这样不但简单高效，而且有个好处：
如果你需要对 Y 也进行拉升的时候，比如 720p 的 YUV420 拉升成 1080p 的 YUV444，UV 拉升到 1080p 后需要做的 shift 就不止 0.5，但是如果拉升前做，shift 还是 0.25:

U = src16.UtoY8(). Dither_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src_left=0.25)
V = src16.VtoY8(). Dither_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src_left=0.25)
Y = src16.Dither_resize16nr(1920,1080,kernel=”lanczos”,taps=4)
YtoUV(U,V,Y)

vs 下的做法如下：
U = core.std.ShufflePlanes(src16,1,vs.GRAY)
U = core.fmtc.resample(1920,1080,kernel="bicubic",a1=0.6,a2=0.4,sx=0.25)
V = …
res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

或者 vs 自带的 Resize 也可以（用的 zimg library）：
res = core.resize.Bicubic(src16, format=vs.YUV444P16, filter_param_a_uv=0.6, filter_param_b_uv=0.4)
这时候 chromaloc 可以省略。默认就是按照输出是 MPEG2 的来。

同理，将一个 YUV444 的视频，转为 YUV420 MPEG2 的视频：

先故意把 UV 平面左边插补出 0.5 像素的溢出，右边删了 0.5 像素；
再降低成 1/4 分辨率：
U = src16.UtoY8(). Dither_resize16(960,540, src_left=-0.5)
V = src16.VtoY8().Dither_resize16(960,540, src_left=-0.5)
YtoUV(U,V,src16)

vs 的做法如下：
U = core.std.ShufflePlanes(src16,1,vs.GRAY).fmtc.resample(960,540,sx=-0.5)
V = …
res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

res = core.resize.Spline36(src16, format=vs.YUV420P16)

UV 平面的拉升，一般用 softcubic(烂源)、nnedi3(好源)、Bicubic（较为均衡），降低则用 Catmull-Rom/spline36 比较好。

4. YUV 转 RGB 时候的两大参数：matrix，range。

YUV444 的格式，就可以和 RGB 互转了。因为每个像素，YUV 三个数值，经过计算，可以得到同样 RGB 三个数值；反之，RGB 数值通过逆运算，也能得到 YUV444 的格式。这种计算方法背后的规定就是 matrix 和 range。

首先说 matrix，这玩意规定了大体上运算应该怎么进行。比如说最初的 BT601 matrix（了解一下就好，不用记住的）

$$
\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 - \frac{37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}
\end{aligned}
$$

有 RGB 数据，你就可以用这个公式转换为 YCbCr。
反过来：

$$
\begin{aligned}
R'_D &= \frac{298.082 \cdot Y'}{256} + \frac{408.583 \cdot C_R}{256} - 222.921 \\
G'_D &= \frac{298.082 \cdot Y'}{256} - \frac{100.291 \cdot C_B}{256} - \frac{208.120 \cdot C_R}{256} + 135.576 \\
B'_D &= \frac{298.082 \cdot Y'}{256} + \frac{516.412 \cdot C_B}{256} - 276.836
\end{aligned}
$$

这些运算基本上是线性的，所以相当于做了矩阵乘法。matrix 就是定义这种矩阵的关键字。常见的有 BT601（常用于标清），BT709（常用于高清，也是目前 720p/1080p 蓝光制作时候我们用的最多的），BT2020（4K 时代的标准），YCgCo，等等。

另一个概念叫做 range，range 是定义 YUV 数据的范围。以 8bit 为例：8bit 下数据范围是 0~255，但是 YUV 表示的时候并不一定使用了所有的范围。tvrange 下，Y 的有效范围是 16~235,16 表示最低亮度，235 表示最高亮度；UV 的范围是 240。跟 tvrange 相对的是 pcrange，永远是 0 为最低，255 为最高。

range 的引入使得同一个值可能有不同的含义。以黑白图像为例，比如 Y=16，在 tvrange 下就是最黑的值；但是如果把它看做 pcrange，那么它还是有一点灰的。同理，Y=235 在 tvrange 下就是白，而在 pcrange 下就会是接近白的亮灰色。

一张 tvrange 的图片被误认为是 pcrange，图像的对比度会被压缩，就是颜色没有那么鲜艳；反之，一张 pcrange 的图被当做是 tvrange，图像的对比度会被拉大，颜色似乎变得很鲜艳，然而极亮和极暗的地方会有细节损失（因为 0~15,236~255 的有效段位数据被丢弃了）

range 的概念只存在于 YUV 下，默认一般是 tvrange。RGB 下没有 range；永远是 0 代表最低值，255 代表最高值。

avs 中，convertto 这个工具就可以实现 YUV 和 RGB 的互转，同时照顾到 matrix 和 range：

AVISource(“fraps.avi”)
ConvertToYV12(matrix=”Rec709”)

就是将 RGB 的 avi 转换为 BT709　tvrange 的 YV12
matrix 可以有以下值：
Rec601：　BT601 tvrange
Rec709：　BT709 tvrange
PC.601：BT601 pcrange
PC.709：BT701 pcrange

编码的时候，x264 可以指定输入的 matrix 和 range，默认的 matrix 按照分辨率来，默认的 range 则是 tv。用 --matrix 和--range 指定。
--matrix 可选”BT470bg”/”smpte170m”, 或者”BT709”。前两个是 BT601 的别名；
--range 一般留空（自动选），否则可以指定 “tv”或者”pc”

所以如果你转为 pcrange BT601 YUV444 8bit:
AVISource(“fraps.avi”)
ConvertToYV24(matrix=”PC.601”)

编码的时候记得在 misc 中指定：--matrix BT470bg --range pc --input-csp “i444” --output-csp ”i444”

同理，从 YUV 转换到 RGB 的时候，也需要指定这些参数。比如说 JPEG 图像的 YUV 数据格式：
YUV420/422/444 都有可能
8bit
BT601 PCrange
MPEG1 的 cplace

如果你拿到一个 JPEG 的 YUV 数据（可以用 JPEGSource 输入），转换成 RGB:

JPEGSource(“xxx.jpg”)
ConvertToRGB24(matrix=”PC.601”, ChromaInPlacement=”MPEG1”)

在现在视频规范下，如果一个视频不指定 matrix，编码器应该默认按照分辨率来算：
如果 长>1024 或者 宽>576，则使用 BT709；否则使用 BT601。
不过其实并不是所有渲染器都这么照做的……不按照分辨率自动设置 matrix，甚至设置了 matrix 也不读取的比比皆是。

5. avs 中 16bit RGB 的伪装

我们在前面的教程中讲过，YUV 格式的 10bit 和 16bit，是靠 interleaved/stacked 来 hack 成高 bitdepth。YV12 是这样，YV16 和 YV24 也是如此：

```
AVISource("fraps.avi") #RGB24
Spline36Resize(1280,720) # avs 自带的 resizer 都支持 RGB 下做重采样
ConvertToYV24(matrix="Rec.709") #转为 YV24
U16() # 转为 YUV444 stacked 16bit
Dither_out() #转为 YUV444 interleaved 16bit
```

avs 中，也有对 RGB 的 16bit hack。16bit 的 RGB 称为 RGB48，就是每个像素占用 48 个 bit。主要的方法有两种： RGB48Y，和 RGB48YV12

RGB48Y 比较好理解，以一个分辨率为 1080p 的 RGB48Y：
它把每一帧 RGB 拆成 3 帧 Y8，按顺序分别表示 R,G,B；
每个 Y8 分辨率为 1920x2160，是一个 stacked 16bit 的 Y8。以 R 对应的平面为例：MSB 部分（就是上半部分）表示前 8bit 的 R 数值；LSB 部分（下半部分）表示后 8bit 的 R 数值。

RGB48Y 这种表示方法的优势在于，它相当于 stacked 16bit 的 RGB，并且伪装成了 YUV 格式。那么就可以用 Dither_resize16 这种只适用于 stacked 16bit YUV 的滤镜：

```
JPEGSource("xxx.jpg")
nnedi3_resize16(lsb_in=false, matrix="601", tv_range=false, cplace="MPEG1",output="RGB48Y")
# 用 nnedi3 将一张 jpeg 图像高精度转换为 RGB48Y
Dither_resize16(1280,720)
对图像在 RGB 下做 16bit 的 resize
```

RGB48YV12 则是另一种伪装：它把 RGB48 的信息伪装成 YV12。我们假设一个 2x2 的图像：

RGB48 模式下，它需要储存 4 个 R，4 个 G，4 个 B，每个各 16bit，总共是 3*4*16=192bit；
一个 4x4 的 YV12：
4*4 个 Y，每个 8bit，总共是 4*4*8=128bit
2*2 对 UV，每个 8bit，总共是 2*2*2*8=64bit
总共是 128+64=192bit

所以，横纵都是 2 倍分辨率的 YV12 视频，正好可以塞下 RGB48 的信息。只不过这些信息本身作为 YV12 来看杂乱无章了。

RGB48YV12 多用于输出给 ImageMagick 等绘图工具。实际在 avs 中一般使用 RGB48Y。
16bit RGB 的知识我们会在后续教程中继续详解。

6. 高精度的 RGB 和 YUV 互转

avs 自带的 ConvertTo 命令，虽然简单方便，功能也基本上齐全，但是毕竟精度低。YUV 和 RGB 的互转，需要大量的计算，我们还是希望能够有原生高精度的工具，支持 16bit 的输入输出。Dither Tools 里面提供了这个工具：

Dither_convert_yuv_to_rgb()
Dither_convert_rgb_to_yuv()

从名称看，不难理解它们分别是 YUV 转 RGB，和 RGB 转 YUV 的工具。

Dither_convert_yuv_to_rgb() 接收 YUV 格式的输入，输出 RGB。参数如下：
matrix，可选”601”, ”709”, “2020”, “YCgCo”。如果不输入，自动选择 601 或者 709。视频高度>600 则选择 709。
tv_range，true/false. 默认 true 表示是 tvrange。
cplace，可选”MPEG1”和”MPEG2”
chromak，chroma upscaling 的 kernel。和 dither_resize16 的 kernel 类似。默认 bicubic
fh, fv, taps, a1, a2, a3，也是从 Dither_resize16 中继承而来。
noring，true/false，是否使用 non-ringing 算法。推荐搭配 lanczos/spline 等使用。
lsb_in，true/false. 指定输入的 YUV 是否是 stacked 16bit。
mode, ampo, ampn, staticnoise 这些是 dither（抖动相关）。下一个章节单独讲述这些。
output，输出的格式。可选”RGB32”(带个空的 alpha 通道), “RGB24”, “RGB48YV12”,”RGB48Y”。

Dither_convert_rgb_to_yuv()接受 RGB 格式，转为 YUV。输入是一个 RGB24 或者 RGB32，也可以接受 RGB48Y。输入 RGB48Y 的时候，需要三个参数指定：

```
# RGB48Y
R = SelectEvery(3,0) #每 3 帧中取第 1 帧，为红色通道
G = SelectEvery(3,1) #每 3 帧中取第 2 帧，为绿色通道
B = SelectEvery(3,2) #每 3 帧中取第 3 帧，为蓝色通道
Dither_convert_rgb_to_yuv(R,G,B)
```

matrix，同上。
tv_range，同上。表示输出的 YUV 视频的 range
cplace，同上
chromak，chroma downscaling 的 kernel。和 dither_resize16 的 kernel 类似。默认 bicubic
fh, fv, taps, a1, a2, a3，也是从 Dither_resize16 中继承而来。
noring，true/false，是否使用 non-ringing 算法。一般 downscaling chroma 就不用了
lsb_in，true/false. 指定输出的 YUV 是否是 stacked 16bit。
mode, ampo, ampn, staticnoise 这些是 dither（抖动相关）。下一个章节单独讲述这些。
output，输出的格式。可选”YV12”,“YV16”, “YV24”,”Y8”。

比如 Fraps 的视频，我们把它转为 YUV444p16 的高精度视频，并且丢给 x264 编码：

```
AVISource(“fraps.avi”) #RGB24
Dither_convert_rgb_to_yuv(matrix=”709”, tv_range=true, output=”YV24”, lsb=true)
Dither_out() #把 yuv444p16 stacked 转为 yuv444p16 interleaved.
```

x264 中编码设置：
--input-csp “i444” --output-csp “i444” --input-depth 16 --matrix “BT709”
x265 中省略--output-csp "i444"，建议加上--cbqpoffs 5 --crqpoffs 5

vs 中类似的存在是 mvf.ToYUV 和 mvf.ToRGB。对照着 doc（源码）很容易看懂用法，非常类似：

res=mvf.ToRGB(JPEGYUV, full=True, cplace="MPEG1", matrix="601", depth=8)，就是把 JPEG 的 YUV，转为 8bit RGB。
src16 = mvf.ToYUV(RGBimg, css="420",depth=16)
则是把一个 RGB，按照分辨率自适应选取 matrix，转为 YUV420P16, tvrange 的 YUV。

7. Dither (抖动)

图像算法中，当做高精度->低精度转换的时候，四舍五入往往不是最佳的选择。比如下图是播放的时候输出 4bit 的 RGB，采用四舍五入：

很可怕吧？低 bit 下色带问题表现的非常明显
但是如果借助抖动算法，同样是 4bit RGB，效果就完全不一样：

抖动算法通过增加抖动噪点，来达到次精度级别的过渡效果。多数 dither 算法还有误差均摊，将取整后误差值均摊到四周的噪点，来降低图像取整的偏差。对于 YUV 8bit 这种容易出现精度问题的，调节 Dither 参数尤为重要。Dither Tools 里面，很多涉及转换的，例如 DitherPost(), Down10(), Dither_convert_xxx(),都有给你设置抖

动的参数：

mode： 抖动的算法。可选：
-1： 不做抖动，四舍五入
0： 8bit ordered dither+噪点。ordered dither 是非常常用的算法之一，它产生的噪点规律不易被有损压缩破坏，所以在视频和图像处理中广泛使用。这个我们后续再说。
1: 1bit 抖动
2：2bit 抖动，轻微
3：2bit 抖动，中等
4：2bit 抖动，较强
5：2bit 抖动，很强
6：fs dither 误差均摊，非常均衡的算法。也一般是默认值。
7：Stucki dither 误差均摊，看上去很锐利，对细微线条保护的较好。
8：Atkinson dither 误差均摊，噪点规律很独特，但是平面处很干净。

一般需要特别处理 YUV 8bit 的时候选 0，否则选 6（默认）

ampo：ordered dither 或者其他误差均摊算法的强度，这个值越高，ordered dither 噪点的规律就越明显，或者其他误差均摊的算法的噪点规律也越突出。范围 0~10.0

ampn: 噪点的强度。在执行抖动算法前，还可以加上一层随机噪点，来强化噪点的力度，并且让抖动噪点的分布更均匀。ampn 就是控制这个力度的。范围 0~4.0，默认 0，表示不加入随机噪点。

staticnoise: 如果加上随机噪点(ampn 控制)，是否加入静态噪点。加入静态噪点可以在压制的时候省一些码率（因为噪点的时间复杂度降低），但是容易对观看产生负面效果——画面在动，噪点不动，看上去很像屏幕上一层灰……默认 false，表示不用静态噪点。

转 RGB24/YUV 10bit, 绝大多数情况下，默认的参数就已经足够好了。

8. 其他一些值得注意的知识点

如果图像处理中，你拿到的是原生 RGB 的图像，或者你某一步必须经过 RGB 处理，再转回 YUV 编码的时候，优先选择 YUV444，保留全部的 chroma 信息。

比如说原图（左），注意右上角极红/极蓝的 logo，和右下角进度条：

上图右是把它转为 YUV420p16，然后再转回 RGB24，用 non-ringing lanczos 4（非常锐利的算法）做 chroma upscaling。不难看出，类似上方 logo，和下方文字、边框等地方，颜色损失非常明显。这是 chroma subsampling 必然的代价——UV 平面的高频信息会遭到毁灭性的打击。这在原生 RGB 源，尤其是游戏视频中非常常见。

保留 YUV422 则好一点（下图左），保留 YUV444 则最佳（下图右）

所以，除非兼容性问题，尽量保持较高的 chroma 分辨率。

x26x 输入可以选--input-csp（输入的 sampling） 和 --output-csp（编码输出的 sampling, x265 不可指定）。分别如下：
“i420”，默认，YUV420
“i422”, YUV422
“i444”, YUV444

如果输入输出不一样，x264 会自己做转换。但是效果不是很好，所以不建议丢给 x264 转换。
如果--output-csp 是 i444，x264 会将 chroma-qp-offset 加 5。这是为了让 chroma 编码的更烂一点，以节省码率。事实上我们说过，chroma 的重要性低于 luma，所以这么做是有利于压缩率的。
宁可让 x264 编码的烂一点，也不要暴力的直接缩减成 422 甚至 420，因为 x264 编码的损失要小于暴力的缩小再放大。同理，x265 也建议手动调整 offset

x26x 编码中可以用--chroma-loc 来设定 chroma placement，但是很多播放器根本不认。所以丢给 x26x 的东西永远保持 MPEG2 的 cplace 吧。就算是 MPEG1 的，你自己给转成 MPEG2 就好了。只需要把 UV 平面分离后，用 src_left=-0.25 在 UV 平面左侧插补出 0.25 像素的空间，或者说向右 shift 0.25 像素。关于 Chroma shift 的修正，我们下一章节再说。