Fundamentals of Media Signal Processing

Project Report

基于 JPEG 图像压缩的视频编码解码器

Name：谢楚琳

Student ID：3160102602

Class：数字媒体技术 1602

Date：2017-11-15

1 Project Introduction

1.1 选题

基于 JPEG 图像压缩的视频编码解码器。

1.2 工作简介

视频压缩编码器是对一个视频提取每一帧进行 JPEG 模式的图像压缩过程，把每一帧编码后的结果存到一个 txt 文档里。最后把视频的必要属性另存在一个 txt 文档。

视频解码器是根据视频属性文档新建视频，再按顺序读取每一个 txt 文档里的内容进行解码，还原每一帧图片，最后重构成一个经过压缩的视频。

1.3 开发环境及系统运行要求

使用软件：Matlab

使用材料：同一个路径里有一个待处理视频，’Videoencode.m’,’ Videodecode.m’

2 Technical Details

1.1 理论知识阐述

为了提高多媒体信号传输和存储的效率，降低对信道容量和存储容量的要求，一般不直接采用简单的脉冲编码调制(PCM)方式传输和存储数字信息，而是对数字化后的信源信号先进行数据压缩，然后再以压缩的形式传输或存储。数据压缩是通过寻求一种有效的信号编码方式来实现的，这种编码称为信源编码，又称高效编码。图像编码是对图像信源进行信源编码，是在保证达到所要求的图像质量前提下，设法降低所必须的数码率而采取的压缩编码技术。通过图像编码达到节省传输带宽或节省所需存储量的目的，同时也为多媒体计算机处理提供可能。

JPEG(Joint Photographic Experts Group)，是第一个国际图像压缩标准。JPEG 系统基本压缩编码主要包括以下几个部分：首先，需要进行色域空间的转换，即将 RGB 分量转换为 YUV 分量并进行颜色下采样；其次，把图像分成 8*8 的子块，对每个子块进行二维离散余弦变换(DCT for Discrete Cosine Transform)，根据量化表对二维 DCT 得到的结果进行量化；

然后，将量化后的数据用 ZigZag 扫描编码，使用差分脉冲编码调制(Differential Pulse Code Modulation, DPCM)对直流(direct current, DC)系数进行编码，对交流(alternating current, AC)系数进行行程长度编码(run-length encoding, RLE)；最后，将所得到的数据进行 Huffman 编码。

视频压缩技术是计算机处理视频的前提。视频信号数字化后数据带宽很高，通常在20MB/秒以上，因此计算机很难对之进行保存和处理。采用压缩技术通常数据带宽降到1-10MB/秒，这样就可以将视频信号保存在计算机中并作相应的处理。采用压缩技术以减少码率，数字化后的视频信号能进行压缩主要依据两个基本条件：

①数据冗余。例如如空间冗余、时间冗余、结构冗余、信息熵冗余等，即图像的各像素之间存在着很强的相关性。消除这些冗余并不会导致信息损失，属于无损压缩。

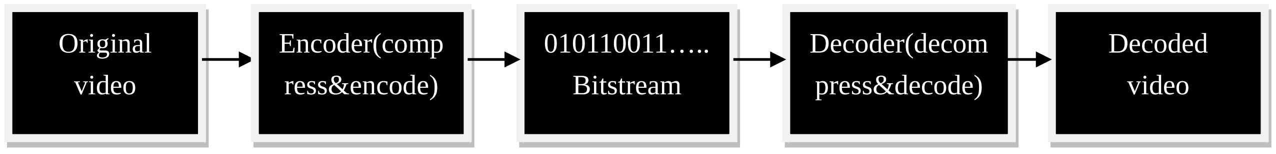
②视觉冗余。人眼的一些特性比如亮度辨别阈值，视觉阈值，对亮度和色度的敏感度不同，使得在编码的时候引入适量的误差，也不会被察觉出来。可以利用人眼的视觉特性，以一定的客观失真换取数据压缩。这种压缩属于有损压缩。

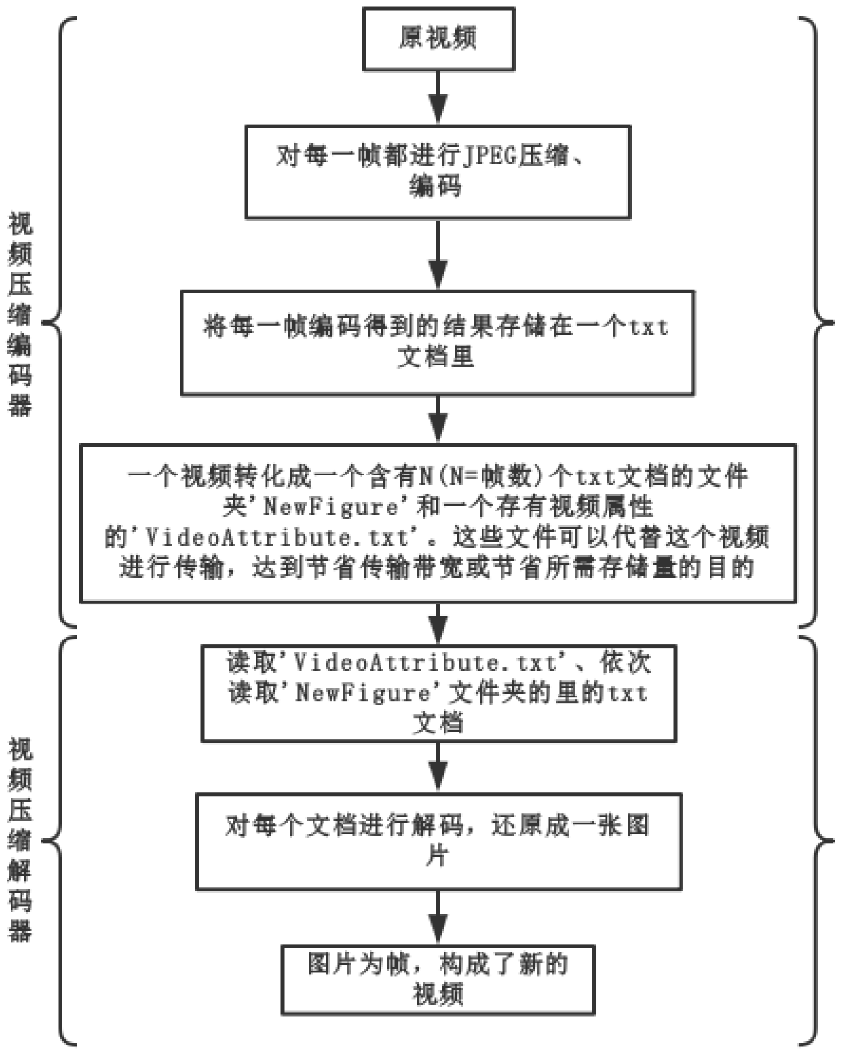
数字视频信号的压缩正是基于上述两种条件，使得视频数据量得以极大的压缩，有利于传输和存储。一般的数字视频压缩编码方法都是混合编码，即将变换编码，运动估计和运动补偿，以及熵编码三种方式相结合来进行压缩编码。通常使用变换编码来消去除图像的帧内冗余，用运动估计和运动补偿来去除图像的帧间冗余，用熵编码来进一步提高压缩的效率。

本实验写出视频压缩的编码器和解码器，根据 JPEG 图像压缩过程对视频的每一帧进行处理，即用颜色下采样、DCT 量化来压缩、用 DPCM、RLE、Huffman 编码和对应解码，还原出视频。是一个简单的视频处理过程。

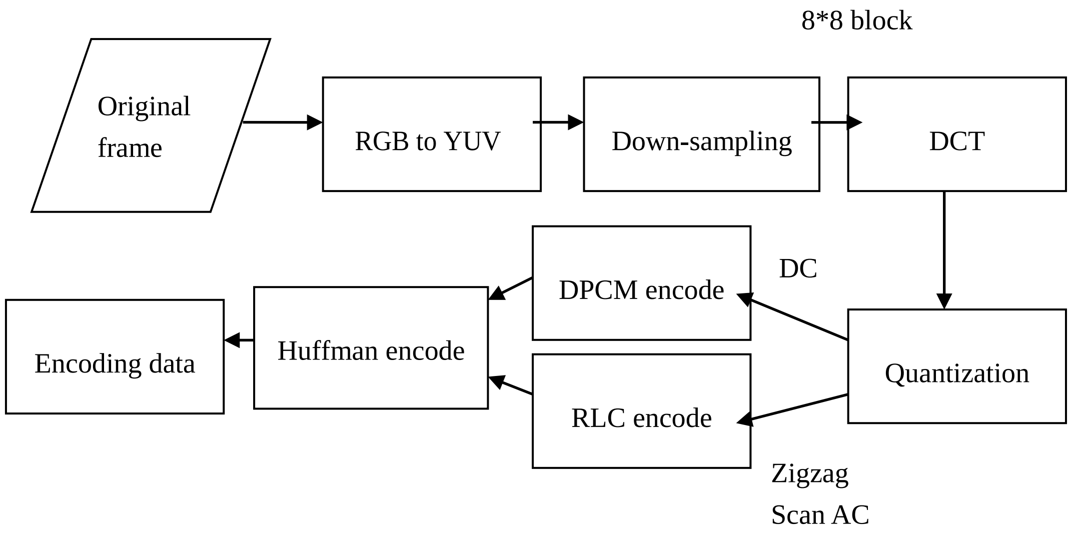
1.2 具体算法描述

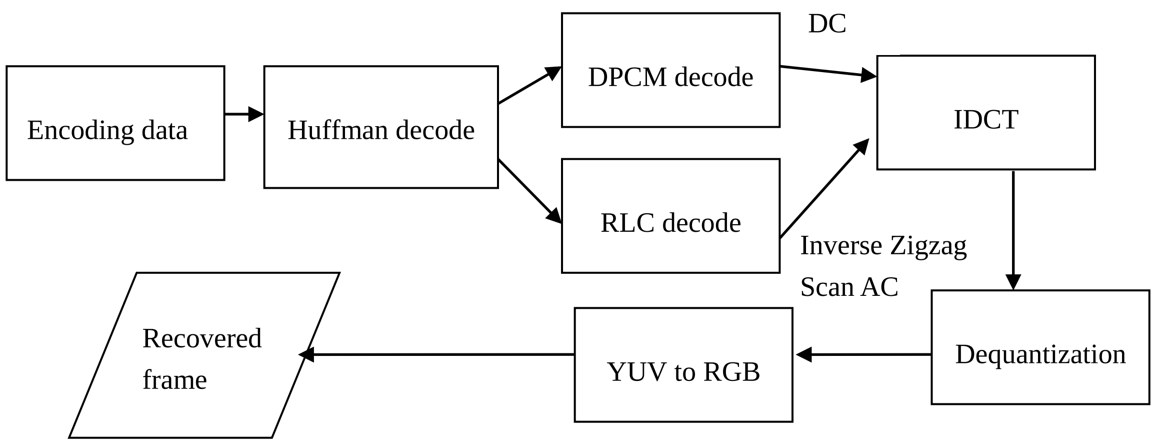
1.2.1 Main Idea：




原视频
对每一帧都进行JPEG压缩、编码
将每一帧编码得到的结果存储在一个txt文档里
一个视频转化成一个含有N(N=帧数)个txt文档的文件夹'NewFigure'和一个存有视频属性的'VideoAttribute.txt'。这些文件可以代替这个视频进行传输，达到节省传输带宽或节省所需存储量的目的
读取'VideoAttribute.txt'、依次读取'NewFigure'文件夹的里的txt文档
对每个文档进行解码，还原成一张图片
图片为帧，构成了新的视频
视频压缩编码器
视频压缩解码器

1.2.2 Frame Process Overview:





1.2.3 Video Encoder

Algorithm 0 – Video preprocess	
function Videoencode()	
(1) video_obj=Videoreader(filename)	% read the video and save it as video_obj
(2) frame_number=video_obj.NumberOfFrames	% get the frame number of the video
(3) frame_rate=video_obj.FrameRate;	% get the frame rate of the video
(4) create a new folder named 'Figure'	% 'Figure' is usd to save the original frames as txt documents to make comparison with the encoding data in txt documents.
(5) create a new folder named 'NewFigure'	% 'NewFigure' is used to save the encoding data of every frames as txt documents.
(6) for i=1:frame_number	
begin	
(6.1) oriframe= the i-th frame	% process the frame one by one
(6.2) save the oriframe as a txt document in the folder 'Figure'	
(6.3) encode the oriframe and save the encoding data as a txr document in the folder 'NewFigure'	
end	
(7) long,height =size (oriframe).	% get the size of the frame and save them as long and height.
(8) save the long, heigth, frame_number, frame_rate in a txt document named 'VideoAttribute.txt'.	% save video's attributes in order to recover the video in the decoding preocess

Algorithm 1 – RGB to YUV Conversion	
Y=0.299* R+0.587*G+0.114*B;	% Y is the luma component which represents the intensity

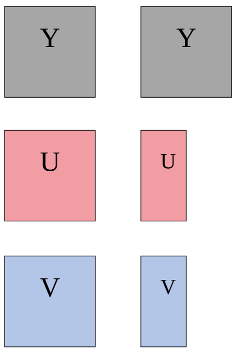
U=-0.169* R-0.3316*G+0.5*B;

V=0.5*R-0.4186*G-0.0813B;

of the image and look likes a gray scale version. U and V are the chrominance components which represent the color information in the image. Converting from RGB to YUV color is necessary for Chrominance Down-sampling afterwards because human eyes are less sensitive to the color information.

Algorithm 2 – Chrominance Down-Samlping

4:4:4 ➔ 4:2:2



% After changing to YUV color space, the imgae is down-sampled by the format 4:2:2. It can cause reduction of pixels in chrominance, that is to say, For the U and Y image layer, making the adjacent pixel share the same chromatic value while Y image layer don't change. Then Y, U, V layers will be processed respectively.

Algorithm 3 – DCT Image Transform

```
function DCT(Img: two dimension matrix)
(1) T ： = 8*8 two-dimension DCT conversion matrix;
(2) for block ： = every 8*8 two-dimension matrix in Img
begin
        (2.1) block=T*block*T'
        (2.2) block=round(block/Norm_Mat)
end
(3) return   Img
```

% for image compression, Discrete Cosine Transform(DCT)is used. To transform spatial to frequency domain for image compression, the system is needed to determine with the forward 2D-DCT transformation equation.

TABLE1: LUMINANCE QUANTIZATION TABLE

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92

% Quantization is the step where most of the image compression takes place. DCT itself doesn't compress the image because it is lossless. JPEG standard has two quantization tables for the luminance and chrominance coefficients. DCT Transformed matrix will divied by the quantization matrix,

49	**64**	**78**	**87**	**103**	**121**	**120**	**101**
72	**92**	**95**	**98**	**112**	**100**	**103**	**99**

then rounded to be the nearest integer calue.

TABLE2: CHROMINANCE QUANTIZATION TABLE

17	**18**	**24**	**47**	**99**	**99**	**99**	**99**
18	**21**	**26**	**66**	**99**	**99**	**99**	**99**
24	**26**	**56**	**99**	**99**	**99**	**99**	**99**
47	**66**	**99**	**99**	**99**	**99**	**99**	**99**
99	**99**	**99**	**99**	**99**	**99**	**99**	**99**
99	**99**	**99**	**99**	**99**	**99**	**99**	**99**
99	**99**	**99**	**99**	**99**	**99**	**99**	**99**
99	**99**	**99**	**99**	**99**	**99**	**99**	**99**

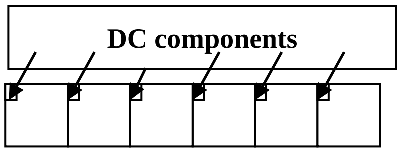
Algorithm 4 – DPCM on DC components

```
function DPCMencode(quanti_img:the DCTed two dimension matrix)

(1) transform the quanti_img matrix to a 8*8n matix, every 8*8 matix is a block.
(2) for block ： = every 8*8 two-dimension matrix
begin
        (2.1) TempDC=(1,j)
        (2.2) NextDC=(1,j+8)
        (2.3) $d_i$=NextDC-TempDC
end
(3) $d_0$=FirstDC
(4) DPCMseq=[$d_0$, $d_1$, $d_2$, ….$d_n$]
(5) return    DPCMseq
```

% After quantization, Diffrential Pulse Code Modulation(DPCM) is the coding method for DC components.

Block 1	**Block 2**	**Block 3**	**…….**	**Block n**



Algorithm 5 –Huffman encoding for DPCM results

```
function HUFFMANencodeDC(DPCM: the DPCM result sequence)
(1)  element，prob= tabulate(DPCM)
```

%Huffman table is created which represents commonly used values with a shorter code.

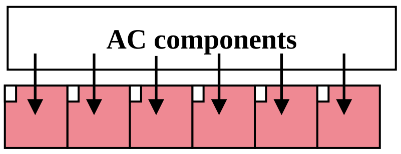
% use matlab's function 'tabulate()' to get the elements and their corresponding probability.

(2) [dict_DC, avglen1] = huffmandict(element,prob)	% use the element and probability to generate the huffman dictionary
(3) huff_DC = huffmanenco(DPCM, dict_DC)	% use the original sequence and the dictionary to encode the sequence

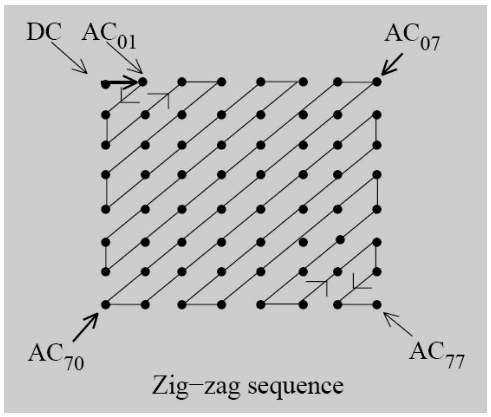
Algorithm 6–ZigZag Scan for AC coefficients

```
function ZIGZAGscan(quanti_img: the  DCTed two dimension matrix)
(1) z=[9 2 3 10 17 25 18 11 4 5 12 19 26 33 41 34 27 20 13 6 7 14 21 28 35 42 49 57 50 43 36 29 22 15 8 16 23 30 37 44 51 58 59 52 45 38 31 24 32 39 46 53 60 61 54 47 40 48 55 62 63 56 64];
(2) for block ： = every 8*8 two-dimension matrix
begin
        (2.1) zigseq(1,(i-1)*63+1:i*63)=block(z)
end
(1)  return zigseq
```

% after quantization, the zigzage scan orders 63 AC coefficients into the one-dimensional vector format. Zigzag order rearranges the quantized coefficients of each 8*8 block for further encoding



For every 8*8 block :



% define a sequence z which contains the Zigzag order to read the AC coefficients

%read the AC coefficients in order of the value of element in sequence z.

Algorithm 7 – Run-Length encoding for AC coefficients

```
function RunlengthEncode(zig_seq: the one-dimensional vector format)
(1)for temp ： = every element in zig_seq
begin
    (1.1) if (temp == 0)
```

% after quantization and zigzag scanning, the one-dimensional vectors with a lot of consecutive zeros are obtained. To be more efficient to save the non-zero values and to skip the number of zeros, the run length

```
        (1.1.1) Skip := the number of zeros ++      coding technique is applied.
      (1.2) else
        (1.2.1) Value:= the next non-zero
component =temp;
        (1.2.2) RLE(++i)=Skip;
        (1.2.3) RLE(++i)=Value;
        (1.2.4) Skip=0;
end
(2)RLE = (skip$_1$, value$_1$, skip$_2$, value$_2$, skip$_3$,
value$_3$,...)
```

Algorithm 8 – Huffman encoding for Run-length-encoding results

It is same as the Algorithm 5 –Huffman encoding for DPCM results

Algorithm 9 – Save data

For every layer, it will have four encoding results, that is , DC huffman code, DC huffman code dictionary, AC huffman code, AC huffman code dictionary. Because we have Y,U,V three layers, we have to save the 12 data into one txt document named 'pic_number.txt' where number is this frame's position in the video. This documents are in the folder 'NewFigure'.

1.2.4 Video decoder

Algorithm Overview (details omitted)

1. **Create a new video and initialize the frame number and frame rate according to the 'VideoAttribute.txt'. Then load the encoding data one by one to recover the frame.**
2. **Use huffman decoding function to decode the DPCM results for DC coefficients and RLE results for AC coefficients.**
3. **Use the run length decoding technique to decode the AC coefficients. Then transform the one-dimensional vector into a blocknumber*63 two-dimensional matrix.**
4. **For every rows, use reverse zigzag scaning for 63 AC coefficients and reform them into a 8*8 block.**
5. **Use DCPM decoding technique to decode the DC coefficients. Then assign the DC coefficients to every corresponding 8*8 blocks.**
6. **Multiply each element in the block by a corresponding quantization value.**
7. **Performs the IDCT on the de-quantized block.**

8. YUC to RGB conversion and recover the image.

9. Return the reformed frame to the video.

1.3 重要技术细节

1.3.1 Matlab funcion

v =VideoReader(filename)
read(v,i)
v = VideoWriter(filename)
writeVideo(v,frame)
	Read the video
	Read i-th frame of v
	Create a video
	Write a frame into v

save(FILENAME,VARIABLES)	Save workspace variables to file
A = importdata(filename)	Load data from file By using A.xx will regain the component whose variable name is xx when it is saved.
mkdir('folderName')	Create a new folder in the current catalog
cd(newFolder) cd...	Enter the new folder Come back to the last folder
D = dctmtx(n)	Discrete cosine transform matrix. Calculate each DCT using D*A*D'.
B = blkproc(A,[m n],fun, parameter1, parameter2, ...)	Block processing
tbl = tabulate(x)	Create a frequency table of data in vector x. Information in tbl is arranged as follows: 1st column — The unique values of x 2nd column — The number of instances of each value 3rd column — The percentage of each value
[dict_AC, avglen1] = huffmandict(element,prob)	Generate a Huffman code dictionary corresponding to a source with a known probability model
dsig = huffmandeco(comp,dict)	Decode the numeric Huffman code vector comp using the code dictionary dict

1.3.2 My technique

U(i,2*j)=U(i,2*j-1)
V(i,2*j)=V(i,2*j-1)
	Chrominance down-samlping, make the adjacent pixels share the same chromatic value.

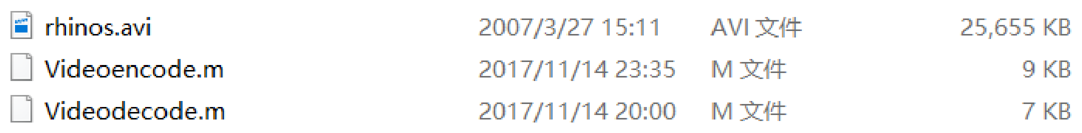
[huff_DCY,dict_DCY,huff_ACY,dict_ACY] =Compression(Y,Norm_Mat1)	It is a function in which it will call other concrete functions to implement the compress and encode.
quanti_img=DCT(img,Norm_Mat);	DCT
DPCM=DPCMencode(quanti_img);	DPCM encode

[huff_DC,dict_DC]= HUFFMANencodeDC(DPCM);	Huffman encode for DC coefficients
zig_seq=ZIGZAGscan(quanti_img);	Zigzag scan
ac_rle=RunlengthEncode(zig_seq);	Run length encode
[huff_AC,dict_AC]= HUFFMANencodeAC(ac_rle);	Huffman encode for AC coefficients
rec_Y=Decompression(huff_DCY,dict_DCY, huff_ACY,dict_ACY,long,height,Norm_Mat1);	It is a function in which it will call other concrete functions to implement the decompress and decode.
re_zig_seq=RunlengthDecode(re_RLE,blocknumber);	Run length decode
re_quanti_img=reverseZIGZAGscan(re_zig_seq,long,height);	Reverse zigzag scan
re_quanti_img=DPCMdecode(re_DPCM, re_quanti_img);	DPCM decode
rec_img=IDCT(re_quanti_img,Norm_Mat);	IDCT

3 Experiment Results

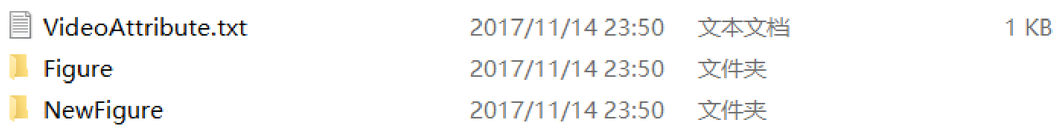
3.1 results

1.Starting from the following instrument.

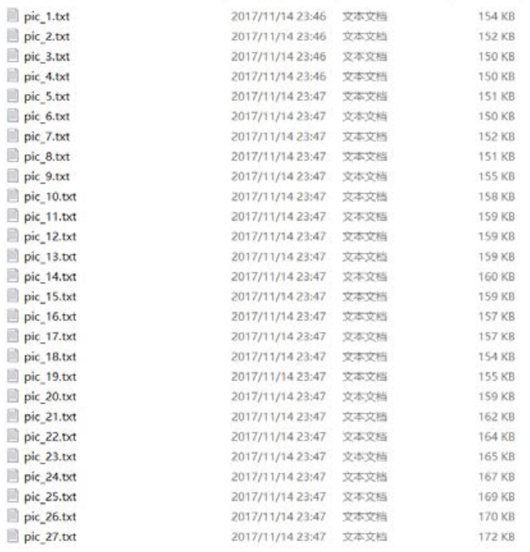


rhinos.avi	2007/3/27 15:11	AVI 文件	25,655 KB
Videoencode.m	2017/11/14 23:35	M 文件	9 KB
Videodecode.m	2017/11/14 20:00	M 文件	7 KB

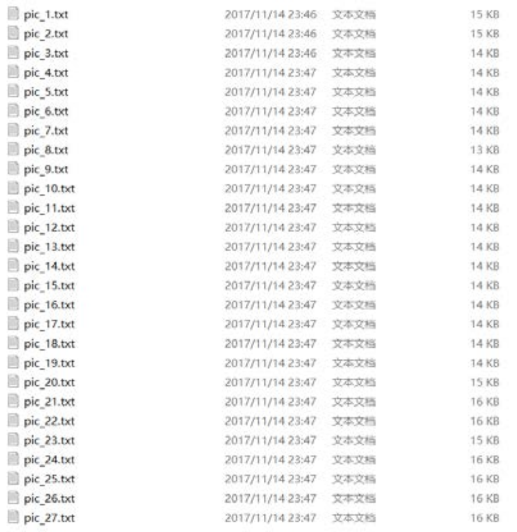
2.After running the ‘videoencode.m’ , you will get the following documents.



VideoAttribute.txt	2017/11/14 23:50	文本文档	1 KB
Figure	2017/11/14 23:50	文件夹	
NewFigure	2017/11/14 23:50	文件夹	



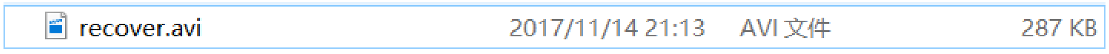
```
pic_1.txt     2017/11/14 23:46  文本文档  154 KB
pic_2.txt     2017/11/14 23:46  文本文档  152 KB
pic_3.txt     2017/11/14 23:46  文本文档  150 KB
pic_4.txt     2017/11/14 23:46  文本文档  150 KB
pic_5.txt     2017/11/14 23:47  文本文档  151 KB
pic_6.txt     2017/11/14 23:47  文本文档  150 KB
pic_7.txt     2017/11/14 23:47  文本文档  152 KB
pic_8.txt     2017/11/14 23:47  文本文档  151 KB
pic_9.txt     2017/11/14 23:47  文本文档  155 KB
pic_10.txt    2017/11/14 23:47  文本文档  158 KB
pic_11.txt    2017/11/14 23:47  文本文档  159 KB
pic_12.txt    2017/11/14 23:47  文本文档  159 KB
pic_13.txt    2017/11/14 23:47  文本文档  159 KB
pic_14.txt    2017/11/14 23:47  文本文档  160 KB
pic_15.txt    2017/11/14 23:47  文本文档  159 KB
pic_16.txt    2017/11/14 23:47  文本文档  157 KB
pic_17.txt    2017/11/14 23:47  文本文档  157 KB
pic_18.txt    2017/11/14 23:47  文本文档  154 KB
pic_19.txt    2017/11/14 23:47  文本文档  155 KB
pic_20.txt    2017/11/14 23:47  文本文档  159 KB
pic_21.txt    2017/11/14 23:47  文本文档  162 KB
pic_22.txt    2017/11/14 23:47  文本文档  164 KB
pic_23.txt    2017/11/14 23:47  文本文档  165 KB
pic_24.txt    2017/11/14 23:47  文本文档  167 KB
pic_25.txt    2017/11/14 23:47  文本文档  169 KB
pic_26.txt    2017/11/14 23:47  文本文档  170 KB
pic_27.txt    2017/11/14 23:47  文本文档  172 KB
```



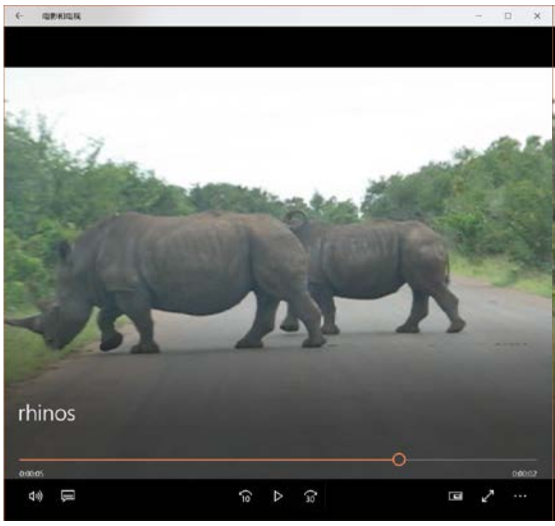
```
pic_1.txt     2017/11/14 23:46  文本文档  15 KB
pic_2.txt     2017/11/14 23:46  文本文档  15 KB
pic_3.txt     2017/11/14 23:46  文本文档  14 KB
pic_4.txt     2017/11/14 23:47  文本文档  14 KB
pic_5.txt     2017/11/14 23:47  文本文档  14 KB
pic_6.txt     2017/11/14 23:47  文本文档  14 KB
pic_7.txt     2017/11/14 23:47  文本文档  14 KB
pic_8.txt     2017/11/14 23:47  文本文档  13 KB
pic_9.txt     2017/11/14 23:47  文本文档  14 KB
pic_10.txt    2017/11/14 23:47  文本文档  14 KB
pic_11.txt    2017/11/14 23:47  文本文档  14 KB
pic_12.txt    2017/11/14 23:47  文本文档  14 KB
pic_13.txt    2017/11/14 23:47  文本文档  14 KB
pic_14.txt    2017/11/14 23:47  文本文档  14 KB
pic_15.txt    2017/11/14 23:47  文本文档  14 KB
pic_16.txt    2017/11/14 23:47  文本文档  14 KB
pic_17.txt    2017/11/14 23:47  文本文档  14 KB
pic_18.txt    2017/11/14 23:47  文本文档  14 KB
pic_19.txt    2017/11/14 23:47  文本文档  14 KB
pic_20.txt    2017/11/14 23:47  文本文档  15 KB
pic_21.txt    2017/11/14 23:47  文本文档  16 KB
pic_22.txt    2017/11/14 23:47  文本文档  16 KB
pic_23.txt    2017/11/14 23:47  文本文档  15 KB
pic_24.txt    2017/11/14 23:47  文本文档  16 KB
pic_25.txt    2017/11/14 23:47  文本文档  16 KB
pic_26.txt    2017/11/14 23:47  文本文档  16 KB
pic_27.txt    2017/11/14 23:47  文本文档  16 KB
```

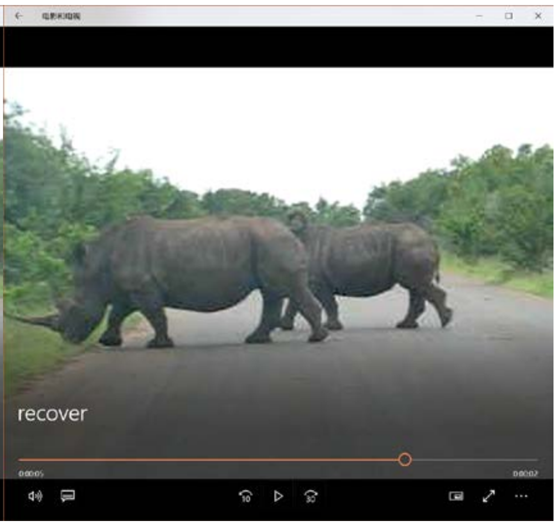
(in Figure folder)　　　　　　　　　　　　　(in NewFigure folder)

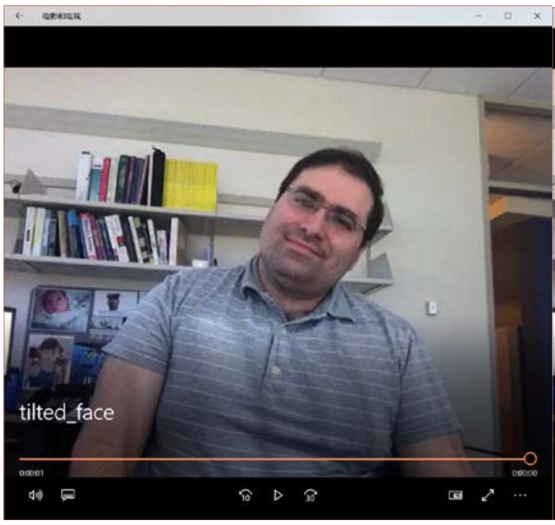
3.After running the ‘videodecode.m’ , you will get following document.

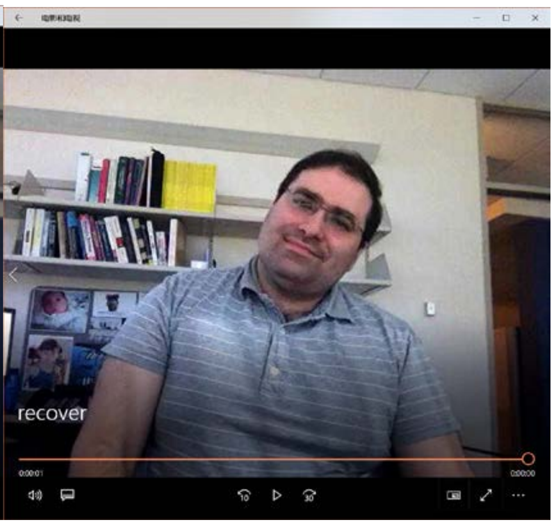


```
recover.avi    2017/11/14 21:13    AVI 文件    287 KB
```









(Comparison of the original video and recovered video, the latter is more indistinct but acceptable)

3.2 space analysis

测试 1：

rihnos.avi	25655KB
recover.avi	1053KB
Every pic_number.txt in ‘Figure’	About 150KB
Every pic_number.txt in ‘NewFigure’	About 15KB
Figure	17.7MB
NewFigure	1.55MB
VideoAttribute.txt	1KB
Encoding results (NewFigure+VideoAttribute.txt)	1.55MB
Coding rate (Figure : encoding results)	17.7:1.55 = 11:1
Compression ratio (original.avi : recover.avi)	25655:1053 = 24:1

测试 2：

tilted_face.avi	27980KB
recover.avi	1366KB
Every pic_number.txt in ‘Figure’	About 680KB
Every pic_number.txt in ‘NewFigure’	About 58KB
Figure	20.2MB
NewFigure	1.69MB
VideoAttribute.txt	1KB
Encoding results (NewFigure+VideoAttribute.txt)	1.69MB
Coding rate (Figure : encoding results)	20.2:1.69 =11:1
Compression ratio (original.avi : recover.avi)	27980:1366 = 20:1

对测试 1 分析：

因为原视频 rihnos.avi 是 avi 格式，大小是 25655KB，本身就被 avi 这个标准压缩编码过，matlab 读取这个视频的时候会把它解码还原成原本的大小，用 txt 格式保存每一帧在 Figure 文件夹，这个文件夹就是这个视频原本的大小，为 17.7MB。

根据视频压缩编码器处理可以得到 VideoAttribute.txt 和 NewFigure 文件夹，总大小为 1.55MB。这些文件可以作为 bitstreams 传输，与 Figure 文件夹相比，编码效率接近 11：1，因此达到节省传输带宽或节省所需存储量的目的。

而根据视频压缩解码器还原出的视频 recover.avi 大小为 1053KB，与 rihnos.avi 相比，压缩比接近 24：1。

对测试 2 分析：

同理可知，编码效率接近 11：1，压缩比接近 20：1。

总的来说，本实验的编码效率接近 10：1，压缩比接近于 20：1。编码和压缩受到原视频每一帧图像的本身质量的影响，一般来说，处理清晰度高的视频，压缩比会较大，而如果原视频已经很模糊，那么压缩结果不明显。

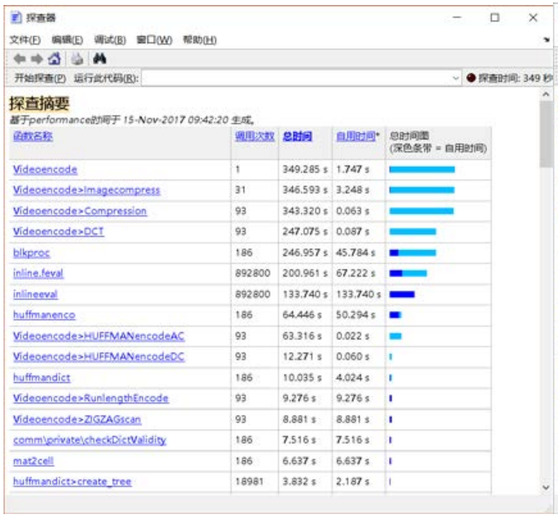
3.3 time analysis

测试 1:

Video name	**rhinos.avi**
Video duration	7s
Frame rate	15 frame/s
Frame number	105 frame
Videoencode running time	201s
Videodecode running time	798s
Total time	999s(16.65min)

测试 2:

Video name	**tilted_face.avi**
Video duration	1s
Frame rate	30 frame/s
Frame number	30 frame
Videoencode running time	349s
Videodecode running time	1546s
Total time	1895s(31.5min)



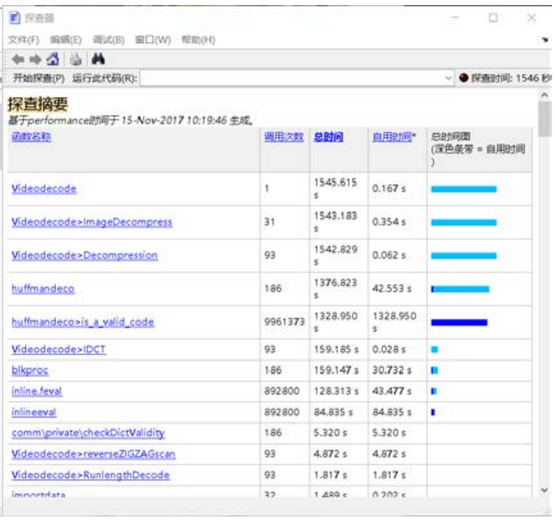
探查器

开始探查(P) 运行此代码(R): ● 探查时间: 349 秒

探查摘要

基于performance时间于 15-Nov-2017 09:42:20 生成。

函数名称	调用次数	总时间	自用时间*	总时间图 (深色条带 = 自用时间)
Videoencode	1	349.285 s	1.747 s	
Videoencode>Imagecompress	31	346.593 s	3.248 s	
Videoencode>Compression	93	343.320 s	0.063 s	
Videoencode>DCT	93	247.075 s	0.087 s	
blkproc	186	246.957 s	45.784 s	
inline.feval	892800	200.961 s	67.222 s	
inlineeval	892800	133.740 s	133.740 s	
huffmanenco	186	64.446 s	50.294 s	
Videoencode>HUFFMANencodeAC	93	63.316 s	0.022 s	
Videoencode>HUFFMANencodeDC	93	12.271 s	0.060 s	
huffmandict	186	10.035 s	4.024 s	
Videoencode>RunlengthEncode	93	9.276 s	9.276 s	
Videoencode>ZIGZAGscan	93	8.881 s	8.881 s	
comm\private\checkDictValidity	186	7.516 s	7.516 s	
mat2cell	186	6.637 s	6.637 s	
huffmandict>create_tree	18981	3.832 s	2.187 s	



探查器

开始探查(P) 运行此代码(R): ● 探查时间: 1546 秒

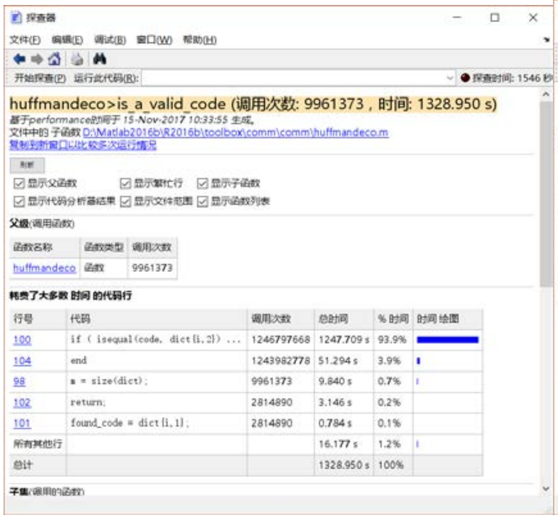
探查摘要

基于performance时间于 15-Nov-2017 10:19:46 生成。

函数名称	调用次数	总时间	自用时间*	总时间图 (深色条带 = 自用时间)
Videodecode	1	1545.615 s	0.167 s	
Videodecode>ImageDecompress	31	1543.183 s	0.354 s	
Videodecode>Decompression	93	1542.829 s	0.062 s	
huffmandeco	186	1376.823 s	42.553 s	
huffmandeco>is_a_valid_code	9961373	1328.950 s	1328.950 s	
Videodecode>IDCT	93	159.185 s	0.028 s	
blkproc	186	159.147 s	30.732 s	
inline.feval	892800	128.313 s	43.477 s	
inlineeval	892800	84.835 s	84.835 s	
comm\private\checkDictValidity	186	5.320 s	5.320 s	
Videodecode>reverseZIGZAGscan	93	4.872 s	4.872 s	
Videodecode>RunlengthDecode	93	1.817 s	1.817 s	
importdata	32	1.489 s	0.202 s	

(测试 2 编码、解码用时)

通过表格和探查器的结果可以看到视频解压用时大于压缩，比例接近 4：1~5:1。



探查器

开始探查(P) 运行此代码(R): ● 探查时间: 1546 秒

huffmandeco>is_a_valid_code (调用次数: 9961373 , 时间: 1328.950 s)

基于performance时间于 15-Nov-2017 10:33:55 生成。

文件中的 子函数 D:\Matlab2016b\R2016b\toolbox\comm\comm\huffmandeco.m

复制到新窗口以比较多次运行情况

☑ 显示父函数 ☑ 显示繁忙行 ☑ 显示子函数

☑ 显示代码分析器结果 ☑ 显示文件范围 ☑ 显示函数列表

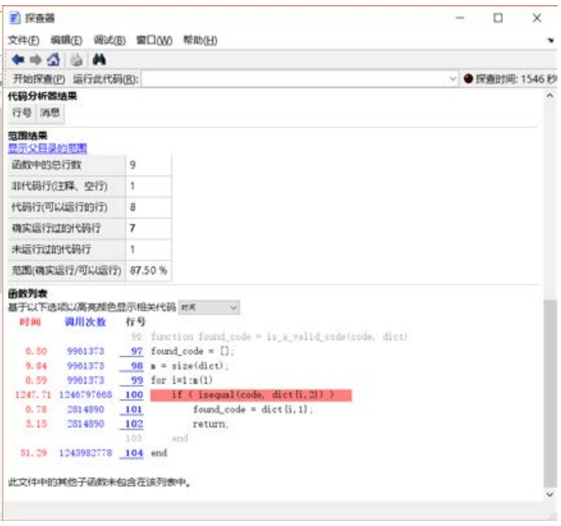
父级(调用函数)

函数名称	函数类型	调用次数
huffmandeco	函数	9961373

耗费了大多数 时间 的代码行

行号	代码	调用次数	总时间	% 时间	时间 绘图
100	`if ( isequal(code, dict{i,2}) ...`	1246797668	1247.709 s	93.9%	
104	`end`	1243982778	51.294 s	3.9%	
98	`m = size(dict);`	9961373	9.840 s	0.7%	
102	`return;`	2814890	3.146 s	0.2%	
101	`found_code = dict{i,1};`	2814890	0.784 s	0.1%	
所有其他行			16.177 s	1.2%	
总计			1328.950 s	100%	

子集(调用的函数)



探查器

开始探查(P) 运行此代码(R): ● 探查时间: 1546 秒

代码分析器结果

行号	消息

范围结果

显示父目录的范围

函数中的总行数	9
非代码行(注释、空行)	1
代码行(可以运行的行)	8
确实运行过的代码行	7
未运行过的代码行	1
范围(确实运行/可以运行)	87.50 %

函数列表

基于以下选项以高亮颜色显示相关代码 时间

```
 时间       调用次数     行号
                          96  function found_code = is_a_valid_code(code, dict)
   0.50      9961373      97  found_code = [];
   9.84      9961373      98  m = size(dict);
   0.59      9961373      99  for i=1:m(1)
1247.71   1246797668     100      if ( isequal(code, dict{i,2}) )
   0.78      2814890     101          found_code = dict{i,1};
   3.15      2814890     102          return;
                         103      end
  51.29   1243982778     104  end
```

此文件中的其他子函数未包含在该列表中。

（测试 2 视频解码器里的 huffmandeco 函数用时和函数耗时的关键代码）

通过探查器分析两次测试的解压用时，都能发现主要花在 Huffmadeco 解码函数中，用时的关键代码是 if (isequal(code, dict{i,2}))。但是因为这是 Matlab 自带的 Huffmandeco 函数，目前无法做出改进。

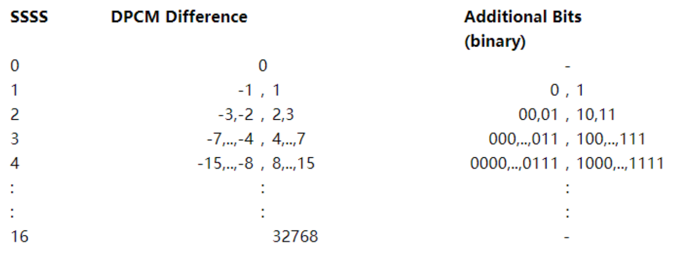
3.4 further analysis of shortages

1．还原后的视频损失了声音，原因是编码器只编码了图像的信息，没有保留音频信息，因此解码器重构的视频也只有图像而没有声音。

2．解码的时候提取数据必须使用编码时候的变量名。即必须知道 ReadResult 里面存储的信息的名称，huff_DCY,dict_DCY, huff_ACY,dict_ACY……

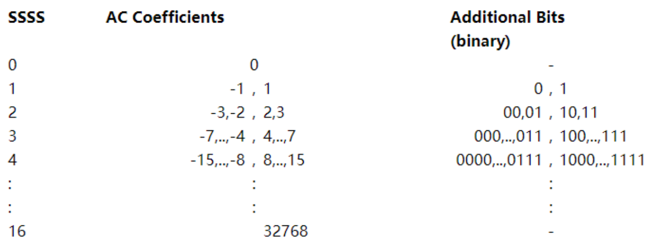
```
%load data from 'ReadResult'
    huff_DCY=ReadResult.huff_DCY;
    dict_DCY=ReadResult.dict_DCY;
    huff_ACY=ReadResult.huff_ACY;
    dict_ACY=ReadResult.dict_ACY;
    huff_DCU=ReadResult.huff_DCU;
    dict_DCU=ReadResult.dict_DCU;
    huff_ACU=ReadResult.huff_ACU;
    dict_ACU=ReadResult.dict_ACU;
    huff_DCV=ReadResult.huff_DCV;
    dict_DCV=ReadResult.dict_DCV;
    huff_ACV=ReadResult.huff_ACV;
    dict_ACV=ReadResult.dict_ACV;
```

3. 本实验的 huffman 解码耗时很长，而标准 JPEG 的 Huffman 编码有自己的 DC 和 AC 的 coding table，详见如下图。但是本实验的 Huffman 编码是根据 matlab 自带的霍夫曼编码函数实现的，与实现标准的 JPEG 过程仍有差距。



SSSS	DPCM Difference	Additional Bits (binary)
0	0	-
1	-1 , 1	0 , 1
2	-3,-2 , 2,3	00,01 , 10,11
3	-7,...,-4 , 4,...,7	000,...,011 , 100,...,111
4	-15,...,-8 , 8,...,15	0000,...,0111 , 1000,...,1111
:	:	:
:	:	:
16	32768	-

Additional bits for Huffman DPCM coding [1]



SSSS	AC Coefficients	Additional Bits (binary)
0	0	-
1	-1 , 1	0 , 1
2	-3,-2 , 2,3	00,01 , 10,11
3	-7,...,-4 , 4,...,7	000,...,011 , 100,...,111
4	-15,...,-8 , 8,...,15	0000,...,0111 , 1000,...,1111
:	:	:
:	:	:
16	32768	-

Additional bits for Huffman AC Coefficients coding [1]

4. 本实验处理的视频的效率很低，处理大于五秒的视频可能需要半小时以上，因此实用性低。而且如果处理清晰度高的视频，压缩比会较大，而如果原视频已经很模糊，那么压缩结果不明显。

3.5 Conclusion

图像处理和视频处理在现代的计算机领域非常重要，JPEG 压缩模式是一种有效的压缩图像而且损失较小的方式。本实验基于 JPEG 图像压缩写成的视频编码和解码器也实现了 10：1 的压缩比以及视频的重构。尽管这是一个简单的视频处理实验，但是在这个过程中也

体会到了 DCT 压缩，DPCM，RLE，HUFFMAN 编码的有效性，进一步熟悉了 MATLAB 的使用，增加了去国内外网站寻找资料的经验，通过对实验结果的空间和时间的分析让我对媒体信号处理有了更深的了解。

然而本实验还有很多不足之处，可有以下方面可以改进：

1. 存储音频信息，使得还原出有声音的视频。
2. 深入研究 Huffmandeco 函数为何用时如此久（虽然目前很难找到相关资料），对视频解码器进行改进。
3. 利用 JPEG 标准的 huffman coding 进行编码和解码，可能可以解决 huffman 解码用时过长的问题，并与国际通用的标准接轨。
4. 对视频压缩加入运动补偿的算法，进一步实现视频时间冗余的去除。

References:

[1] 视频压缩编码和音频压缩编码的基本原理
http://blog.csdn.net/leixiaohua1020/article/details/28114081
[2] 张春田,苏育挺,张静. 数字图像压缩编码[M] .北京:清华大学出版社,2006
[3] JPEG Overview
http://home.elka.pw.edu.pl/~mmanowie/psap/neue/1%20JPEG%20Overview.htm
[4] EI EI PHYO, NANG AYE AYE HTWE. JPEG Image Compression and Decompression using Discrete Cosine Transform (DCT). International Journal of Scientific Engineering and Technology Research, 2016,3(9):1780~1785(in English)