Lab 5
More background and Implementation Hints

Blitzschnell DCT Review

Matrix description of transforming a column vector

- **The transform of a vector is a matrix multiply**
  - Make the xform basis vectors the rows of a matrix

The transform coefficients

$$\underline{\mathbf{c}} = A\underline{\mathbf{x}}$$

$$= \begin{bmatrix} \underline{\mathbf{u}}_1^T \\ \underline{\mathbf{u}}_2^T \\ \vdots \\ \underline{\mathbf{u}}_N^T \end{bmatrix} \underline{\mathbf{x}}$$

Vector being "transformed" (i.e. for which we are computing projections onto a new orthonormal basis)

  - Every set of orthonormal basis vectors forms a "transform" via a matrix multiply

Computing a 2-D Transform

- **Start with a 2-D array *X***
- **First 1-D transform all the columns to get an intermediate matrix**
- **Then transform all the rows of the intermediate matrix to get the final matrix**
- **This can be written concisely as the product of 3 matrices**

$$C = \underbrace{AX}_{\text{xforms the columns of X}} A^T$$

Multiplying *AX* by A^T xforms the rows of *AX*

The 1-D Discrete Cosine Transform (DCT)

- **The 1-D DCT is a matrix multiply as you would expect**
  - The basis vectors for the DCT are the following sinusoids

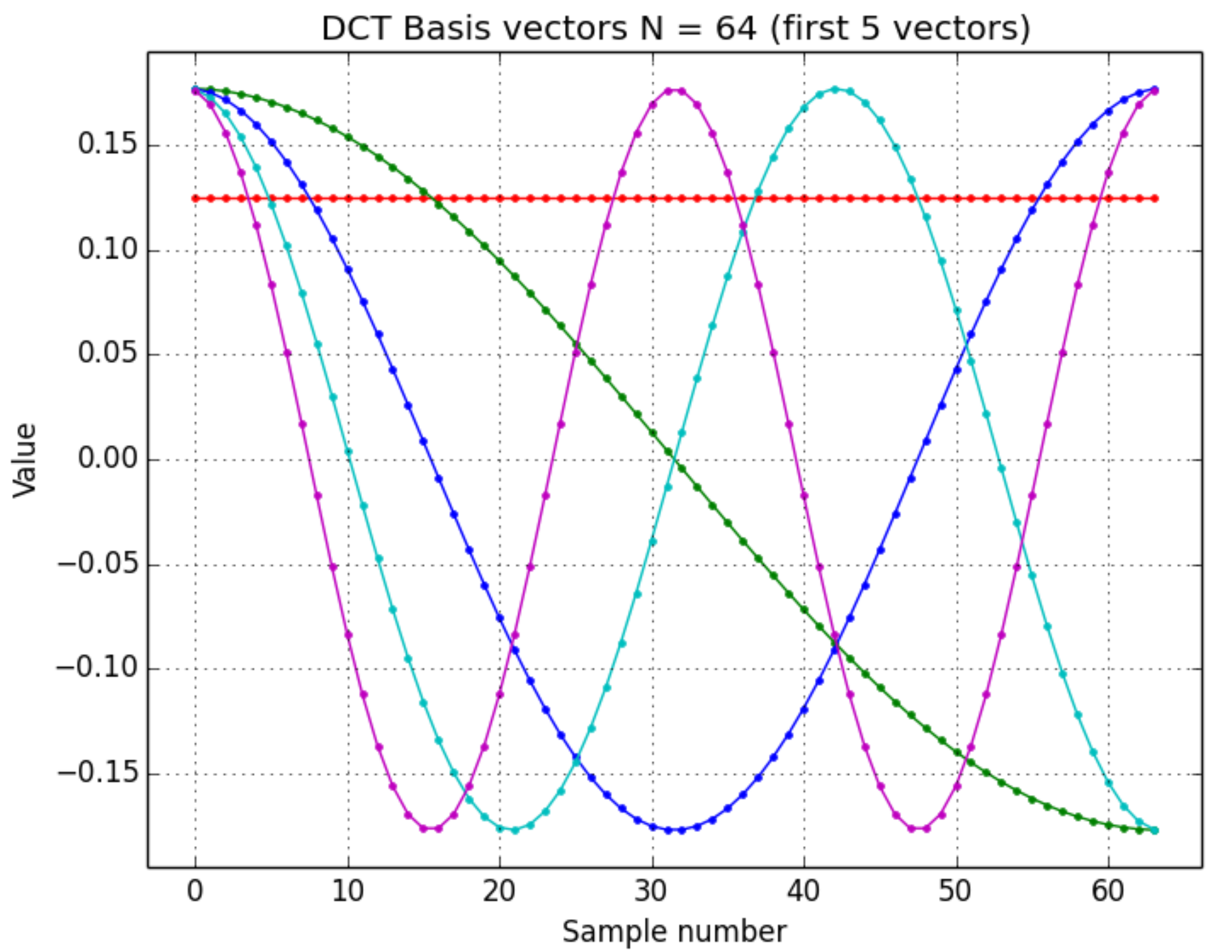
$$\underline{\mathbf{u}}_{k+1} = \alpha(k)\cos\left(2\pi\left(\frac{k}{2N}\right)n + \frac{\pi}{2N}k\right)$$

$$\alpha(k) = \begin{cases} \sqrt{\frac{1}{N}} \text{ for } k = 0 \\ \sqrt{\frac{2}{N}} \text{ for } k = 1, 2, \ldots, N-1 \end{cases}$$

The DCT Basis Functions

CU Boulder

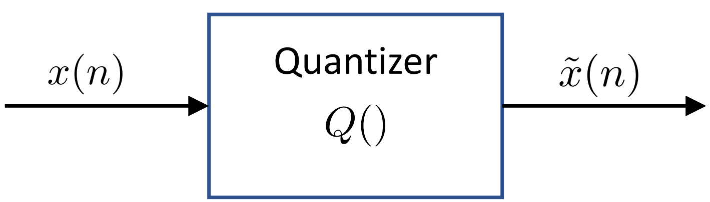
- **The basis functions are sinusoids**



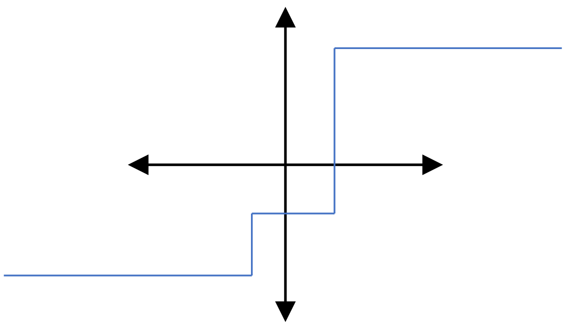
Quantization

Quantization

- **Often a value must be "quantized" to a smaller number of levels**
- **Quantizing is just rounding off**

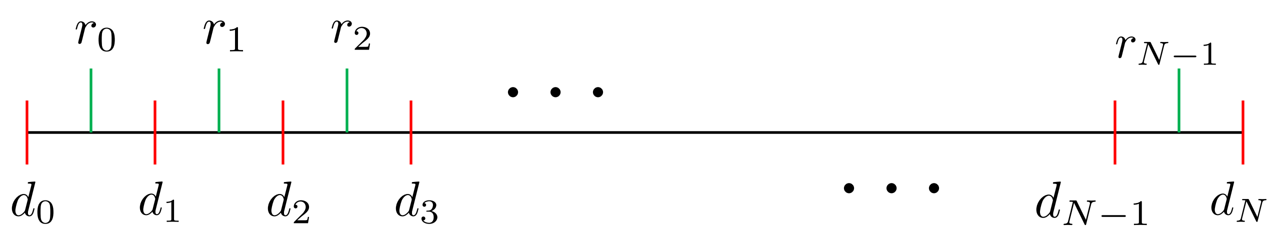


3 level non-uniform quantizer



Uniform Quantizer

- **Values between two decision levels are rounded off to a reconstruction level between them**
- **Decision levels and reconstruction levels are uniformly spaced**



A Deeper Look at JPEG

Reminder: Image is partitioned into 8-by-8 blocks

Reminder: Block Processing

- **Blocks are processed left-to-right and top-to-bottom**
- **Each block is transformed using a 2-D DCT**
  - There are 64 basis images and each block can be reprsented as a "mixture" (linear combination) of these basis images
  - This is just like the DFT!

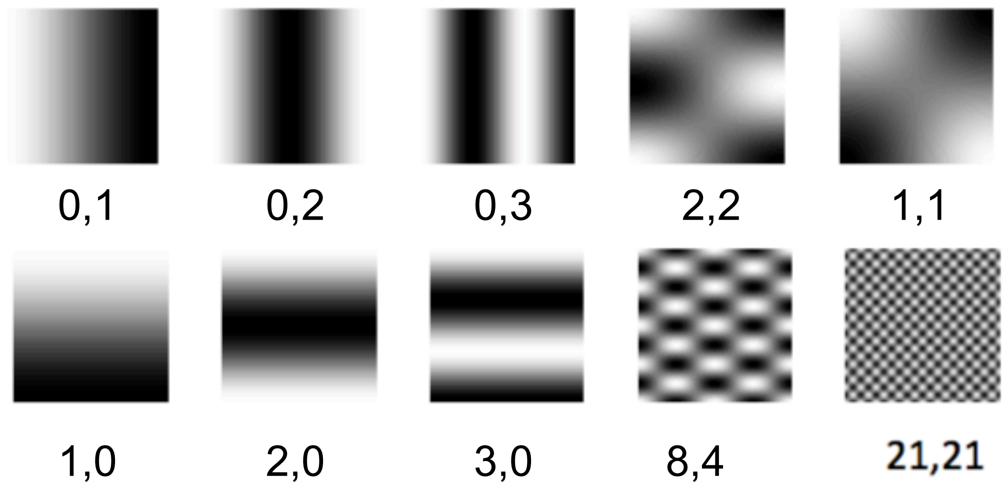


Figure 1: Some basis images where "i, j" means $\mathbf{u}_i\mathbf{u}_j^T$ for $N = 64$

JPEG Features

- **8 and 12 bit modes**
  - Medical imaging requires higher fidelity
- **Sequential encoding**
  - Single pass from top to bottom of an image to complete encoding
- **Has a lossless mode**
  - Not commonly used
- **Entropy coding via Huffman or Arithmetic coding is used**
  - Arithmetic does 5% to 10% better and adapts during coding, but is less common since more complex to implement

Image Components (e.g Y, Cr, Cb)

- An image may have up to 255 components (sometimes called “channels” or ”spectral bands”)
- No pre-defined color spaces
- Typical images have three channels: luminance and two chrominance
- Channels may have different dimensions but they must be related by an integer factor of 1,2,3 or 4 to the highest dimension channel (normally Y)

Image Components (e.g Y, Cr, Cb)

CU Boulder

- Chrominance can therefore be sub-sampled relative to luminance
- Image components are commonly sent sequentially, one after the other, but their blocks can be interleaved if desired

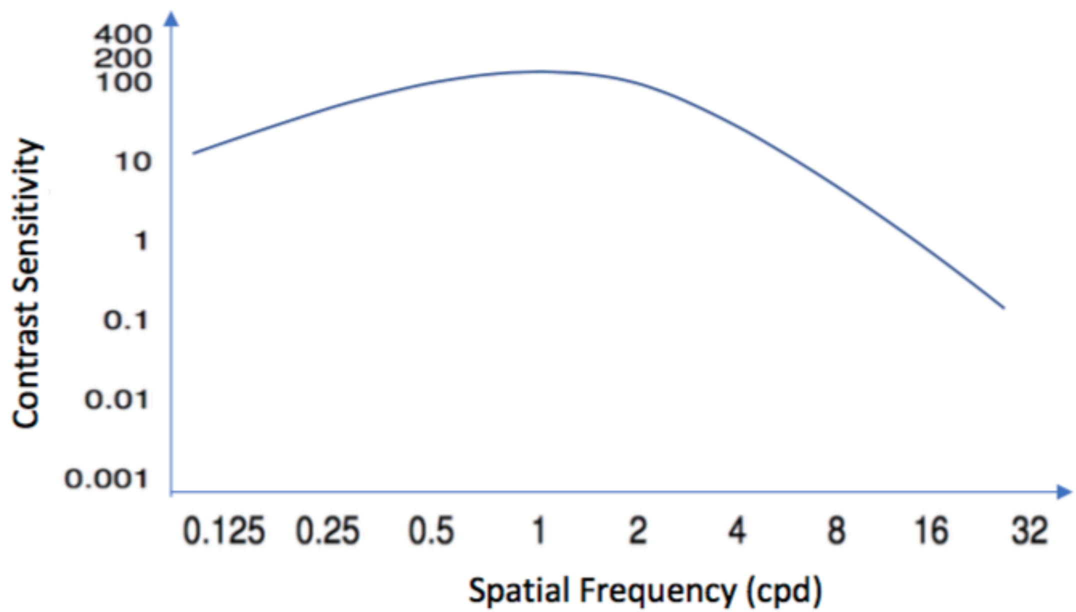
Transform and Quantization

- Each block is transformed using the 2D DCT to get C(u,v)
- Each coefficient is quantized using a uniform quantizer
- The quantizer stepsize depends on the coefficient. The stepsize is given by Q(u,v)

$$C_q(u,v) = Round\left(\frac{C(u,v)}{Q(u,v)}\right)$$

Quantization Considerations

- **Some coefficients are more important than others, so use bits where they matter the most**
  - Some carry more energy
  - The eye is more sensitive to some frequencies than others



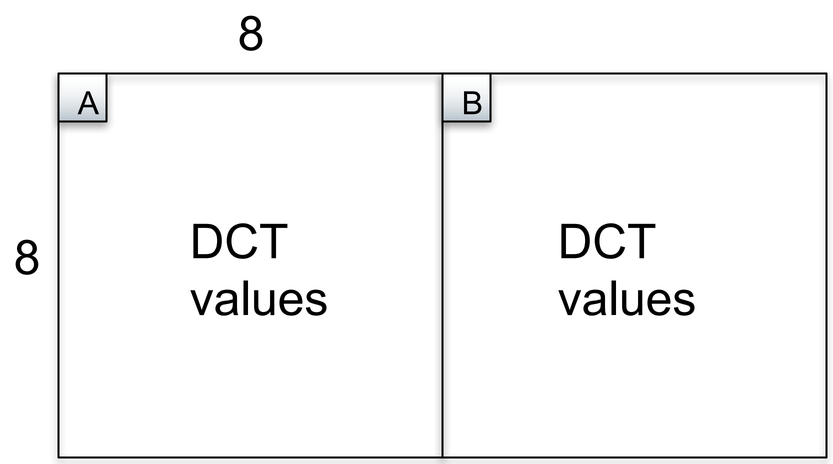
Transform and Quantization

- **The table Q(u,v) is sent with the image. The table below is often scaled and then used for Y (see annex K of standard)**

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

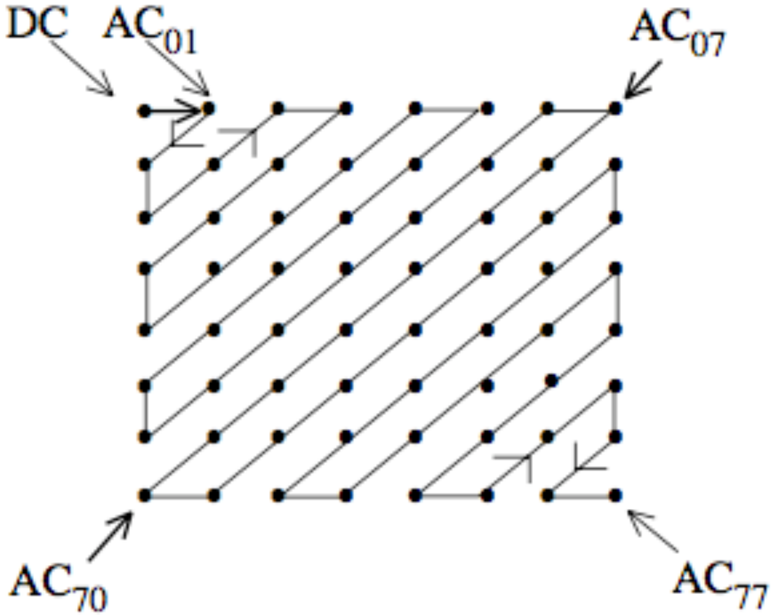
DC coefficient coding

- **The DC coefficient of each transformed 8-by-8 block, located in the upper left corner, is predicted by the preceding block's DC coefficient**
- **There is usually strong correlation between adjacent block DC values**
- **The difference, B – A, is encoded using a Huffman code**



2-D Zig-Zag Scan of coefficients

- **Converts 2-D array into a 1-D array**



Zig-Zag scan properties

- Energy is concentrated at the lowest frequencies
- Zig-zag scan therefore creates 1-D array with the largest “highest energy” coefficients first
- Typically there are many zeros at the end of each block
- The result of this scanning is then entropy coded

Implementation Suggestions

Overall Code Organization

CU Boulder

- **Create two files**
  - Library module
    - ✓ Contains all the support functions
  - Main module
    - ✓ Calls some of the routines from the library module to implement the main processing

Library Module

- **Library module routines**
  - dctmgr(), idctmgr()
    - ✓ mydct2d(), myidct2d()
    - ✓ zigzagscan(), izigzagscan()
  - quant_coeffs(), iquant_coeffs()
  - enc_rbv(), dec_rbv()

Main Program

- **Processing flow**
  - Input image
  - Encode:
    - ✓dctmgr()
    - ✓quant_coeffs()
    - ✓enc_rbv()
  - Decode:
    - ✓decode_rbv()
    - ✓iquant_coeffs()
    - ✓idct_mgr()

Implementing mydct2d( ) and myidct2d( )

- **Use scipy.fftpack’s dct() return**
  - Pass in a 2D 8x8 array to your routines
  - First xform/ixform the rows (axis 0)
  - Then xform/ixform the cols (axis 1)
  - use norm=’ortho’ in both cases
    - ✓ This is **critical** or you won’t get the results you need

Implementing dctmgr( ) and idctmgr( )

- **Use a datastructure to hold the DCT coefficients that has 64 rows, and as many columns as there are 8x8 blocks in the image**
  - Each column then holds one block's worth of coefficients
  - Scan image blocks from left to right, top to bottom as you populate the DCT coefficient array
  - The number of image blocks is rows*cols/64
  - zig-zag scan the block coefficients prior to storing them in your DCT coefficient array
- **While (or after) populating the array, implement the DC coefficient prediction**
  - Remember to reset prediction to zero at start of a row of blocks

Implementing zigzagscan( ) and izigzagscan( )

- **Use a table lookup. I have put a file "useful_arrays.py" in canvas which may help you depending on your choice of implementation strategies**

Implementing quant_coeffs( ) and iquant_coeffs( )

- **Pass in your DCT coefficient array to quant_coeffs()**
- **<u>Don't</u> quantize the DCT prediction residuals!**
- **Make the output of quant_coeffs( ) an array of ints the same size as the DCT coefficient array**
- **The file "useful_arrays.py" may help you with your implementation**

Implementing enc_rbv( ) and dec_rbv( )

- **Declare an initial symb array as follows**

```
symb = np.zeros( (0,3), np.int )
```

- **Add rows to this array as follows**

```
for i in range(coeffs_q.shape[1]):
  # Handle the DC coefficient
  symb = np.vstack( [ symb, [0, 12, coeffs_q[0,i]] ] )
```

- **You can get an array of indices where another array has non-zero values like this**

```
# Handle the AC coefficients. nzi means "non-zero indices"
tmp = coeffs_q[:,i].flatten()
nzi = np.where( tmp[1:] != 0 )[0]
```

Implementing enc_rbv( ) and dec_rbv( ) cont.

- **IMPORTANT!**
  - A DC prediction residual of zero does NOT count as one of the zeros when computing a run of zeroes before a non-zero AC coefficient
  - In other words, start counting runs of zeros with the first AC coefficent when computing triplets (triplets = the rows of symb[ ])
- **You can test two numpy arrays for equality like this**

```
EOB = np.asarray([0, 0, 0], np.int)

while not np.array_equal(symb[symb_row], EOB) :
```

Checkpoints

- **For Mandril and loss_factors of 1 and 8 my symb[ ] arrays start out like this**

```
bombay% lab05_a3.py mandril.pgm z.jpg 1
rows = 512; cols = 512; loss_factor = 1.0
symb[0:25] =
[[  0  12 562]
 [  0  11   7]
 [  0  11  -1]
 [  1  11   3]
 [  0  11  -7]
 [  0  11   2]
 [  0  11   6]
 [  0  11  -2]
 [  0  11   2]
 [  0  11   2]
 [  0  11   1]
 [  0  11  -3]
 [  0  11   5]
 [  0  11  -2]
 [  0  11  -1]
 [  0  11   2]
 [  0  11   1]
 [  4  11  -1]
 [  2  11   2]
 [  1  11   1]
 [  1  11  -1]
 [  0  11   1]
 [  2  11  -1]
 [  0  11  -1]
 [  9  11   1]]
bombay%
```

```
bombay% lab05_a3.py mandril.pgm z.jpg 8
rows = 512; cols = 512; loss_factor = 8.0
symb[0:25] =
[[  0  12 562]
 [  0  11   1]
 [  3  11  -1]
 [  1  11   1]
 [  5  11   1]
 [  0   0   0]
 [  0  12   0]
 [  0  11   1]
 [  1  11   1]
 [  4  11  -1]
 [  7  11   1]
 [  0   0   0]
 [  0  12 -36]
 [  0  11   1]
 [  0  11   1]
 [  0  11   1]
 [  0  11   1]
 [  2  11   1]
 [  0   0   0]
 [  0  12 307]
 [  0  11   1]
 [  3  11  -2]
 [  0  11  -1]
 [  0  11   2]
 [  0  11  -1]]
bombay%
```

Checkpoints

- **For Clown and loss factors of 1 and 8 I got the following PSNR values**
  - loss factor 1.......... 36.13 dB
  - loss factor 8..........28.52 dB

Supplemental JPEG Details

Entropy coding of DC Difference

- DC differences values are classified into a category. The category gives number of bits needed to code the exact value within the category
- The category in which a value falls is coded with a Huffman code
- The number of bits required to code the DC residual is therefore H + C where H is the number of bits in the Huffman code for the category, and C is the “category” number of bits following the Huffman code

Huffman table for DC Category

CU Boulder

- Below is the table for the DC difference category

Range in which absolute value of DC prediction residual falls	Category	Category Huffman code
0	0	00
1	1	010
[2, 3]	2	011
[4, 7]	3	100
[8, 15]	4	101
[16, 31]	5	110
[32, 63]	6	1110
[64, 127]	7	11110
[128, 255]	8	111110
[256, 511]	9	1111110
[512, 1023]	10	11111110
[1024, 2047]	11	111111110
Note: range of DC prediction residuals is [-2040,2040]		

Example DC coding

- Assume the prediction residual is -508
- The category is therefore "9" and is Huffman coded as "1111110"
- The difference lies in the range: [-511, -256] or [256, 511]
- The value of -508 is 3 to the right of -511, so the next 9 bits are: "000000011"
- A total of 7 + 9 = 16 bits are expended coding this value. Good thing it doesn't happen very often!

AC coding

- **Zig-zag scanned array is "run-level" encoded.**
- **First, the number of zeros preceding an AC coefficient is determined. This can be up to 15. Four bits are used to specify the number: "RRRR"**
- **Then, like for DC, the "category" is determined for the AC coefficient magnitude. Four bits specify it: "CCCC"**
- **The eight bits "RRRRCCCC" are Huffman coded**
  - The following bits (category dependent) specify the magnitude within the category
- **The "End of Block" (EOB) code means all remaining coefficients are zero**
- **A special codeword exists to indicate a run of 16 zeros**

Some sample suggested codewords

CU Boulder

Run/ Category	Base Code	Length	Run/ Category	Base Code	Length
0/0	**1010 (=EOB)**	**4**			
0/1	00	3	8/1	11111010	9
0/2	01	4	8/2	111111111000000	17
0/3	100	6	8/3	1111111110110111	19
0/4	1011	8	8/4	1111111110111000	20
0/5	11010	10	8/5	1111111110111001	21
0/6	111000	12	8/6	1111111110111010	22
0/7	1111000	14	8/7	1111111110111011	23
0/8	1111110110	18	8/8	1111111110111100	24
0/9	1111111110000010	25	8/9	1111111110111101	25
0/A	1111111110000011	26	8/A	1111111110111110	26
1/1	1100	5	9/1	111111000	10
1/2	111001	8	9/2	1111111110111111	18
1/3	1111001	10	9/3	1111111111000000	19
1/4	111110110	13	9/4	1111111111000001	20
1/5	11111110110	16	9/5	1111111111000010	21
1/6	1111111110000100	22	9/6	1111111111000011	22
1/7	1111111110000101	23	9/7	1111111111000100	23
1/8	1111111110000110	24	9/8	1111111111000101	24
1/9	1111111110000111	25	9/9	1111111111000110	25
1/A	1111111110001000	26	9/A	1111111111000111	26
2/1	11011	6	A/1	111111001	10
2/2	11111000	10	A/2	1111111111001000	18
2/3	1111110111	13	A/3	1111111111001001	19
2/4	1111111110001001	20	A/4	1111111111001010	20
2/5	1111111110001010	21	A/5	1111111111001011	21
2/6	1111111110001011	22	A/6	1111111111001100	22
2/7	1111111110001100	23	A/7	1111111111001101	23

Example AC Coding

- **Assume after quantization and zig-zag scan we have a 1-D array that looks like this: (DC_value, 0, 0, 10, 0, 64, 0, 0, 0, … ,0)**
- **The DC_value is coded using DPCM**
- **The array is parsed into: (0, 0, 10) ; (0, 64) ; EOB**
- **(0, 0, 10) become RRRR-CCCC = 0010-0100**
  - The 20 bit codeword for “2/4” on the previous slide is then output
  - The 4 bits “1010” are sent for “10”
- **(0, 64) is coded similarly**
  - “1/7” uses 23 bits
  - “64” uses 7 bits: “1000000”
- **EOB is sent last using the codeword “1010”**

Chroma quantization table

- The table below is commonly scaled for the chrominance components (Annex K of standard)
- This table assumes chroma sub-sampling by a factor of 2 in both directions was used

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