Numerical Analysis

Carmen Arévalo

Lund University

carmen@maths.lth.se

Discrete cosine transform

$$C = \sqrt{\frac{2}{n}} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \cos\frac{(2n-1)\pi}{2n} \\ \cos\frac{2\pi}{2n} & \cos\frac{6\pi}{2n} & \cdots & \cos\frac{2(2n-1)\pi}{2n} \\ \vdots & & \vdots & \vdots \\ \cos\frac{(n-1)\pi}{2n} & \cos\frac{(n-1)3\pi}{2n} & \cdots & \cos\frac{(n-1)(2n-1)\pi}{2n} \end{pmatrix}$$

The DCT of x is the vector

$$y = Cx$$

C is a real orthogonal matrix. The DCT involves only real computations and consists only of cosines.

Trigonometric interpolation

$$P(t) = \frac{1}{\sqrt{n}} y_0 + \sqrt{\frac{2}{n}} \sum_{k=1}^{n-1} y_k \cos \frac{k(2t+1)\pi}{2n}$$

with

$$y = Cx$$

satisfies

$$P(j) = x_j, \quad j = 0, \ldots, n-1.$$

Like the DFT, the DCT transforms n data points into n interpolation coefficients.

Least squares with DCT

Like for the DFT, the choice of $m < n$ coefficients $y_0, \ldots, y_{m-1}$ in

$$\frac{1}{\sqrt{n}} y_0 + \sqrt{\frac{2}{n}} \sum_{k=1}^{m-1} y_k \cos \frac{k(2t+1)\pi}{2n}$$

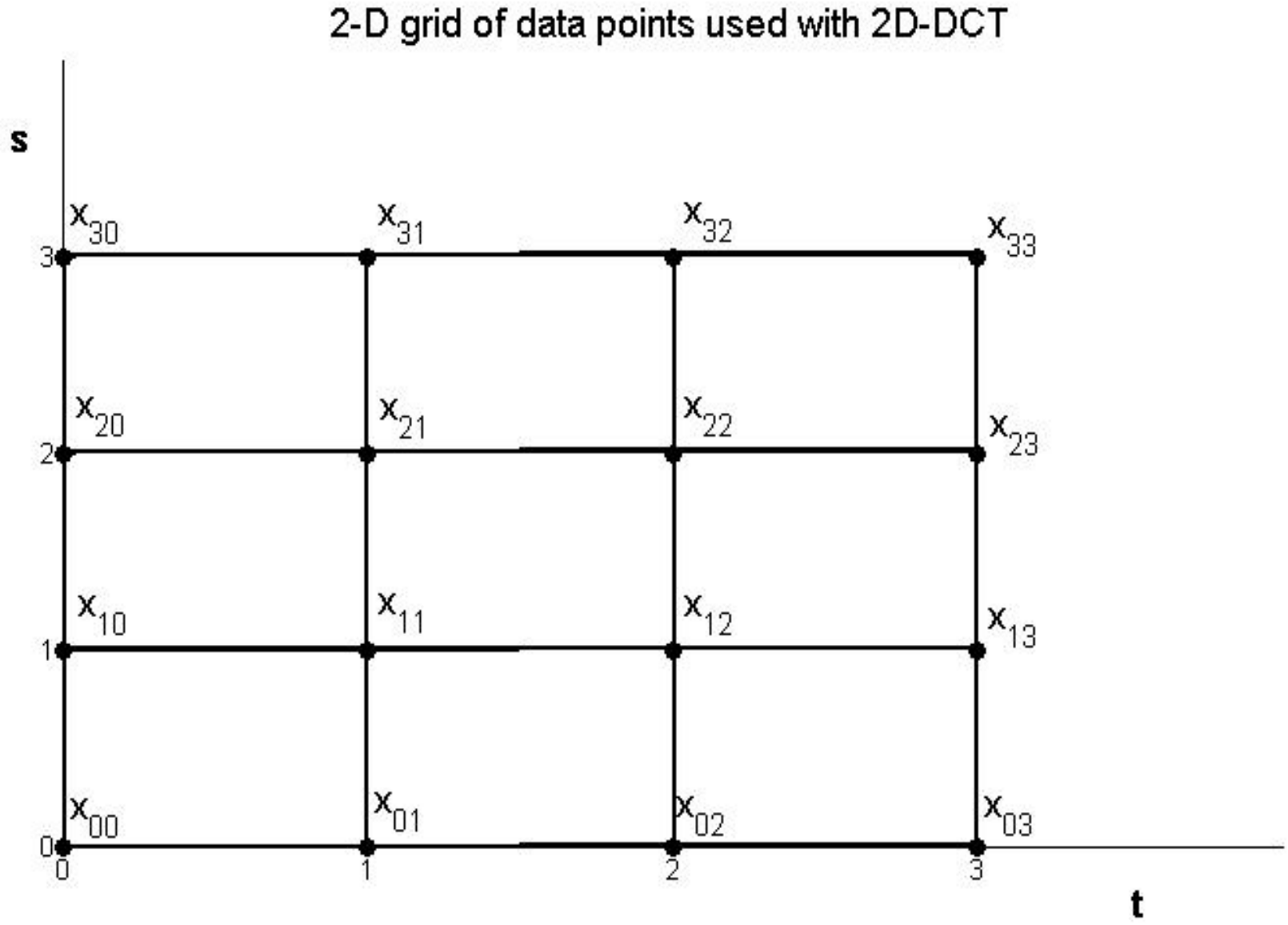
gives the best approximation in the 2-norm.

2D discrete cosine transform

Applying the DCT to a matrix of data (e.g. a pixel file), first in a vertical direction and then in a horizontal direction, we can use the DCT in image processing.

The human eye does not distinguish the higher visual frequencies, so here we will also be interested in compressing by doing a least squares that eliminates the higher frequencies.

Convention: Each point is represented by a pair (s, t), where s is the vertical coordinate and t the horizontal component. We start the numeration at 0.



Each point (s_i, t_j) has a corresponding value x_{ij}.

Matrix form of the 2D DCT and its inverse

The 2D DCT constructs an interpolating function $F(s,t)$ that interpolates the n^2 points (s_i, t_j, x_{ij}).

Let the matrix X contain the values x_{ij}.

1. Apply the DCT in the horizontal t direction. Each column of CX^T corresponds to a fixed s_i.

2. Apply the DCT in the vertical s direction, $Y = C(CX^T)^T = CXC^T$.

$$\boxed{Y = CXC^T}$$

$$\boxed{X = C^T Y C}$$

2D-DCT interpolation and least squares

The inverse of the transform performs an interpolation, just like for the DFT.

The 2-dimensional trigonometric polynomial that satisfies $P(i,j) = x_{ij}$ for $i, j = 0, \ldots, n-1$ is

$$P(s,t) = \frac{2}{n} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} y_{kl} a_k a_l \cos \frac{k(2s+1)\pi}{2n} \cos \frac{l(2t+1)\pi}{2n}$$

where $a_0 = 1/\sqrt{2}$ and $a_k = 1$ for $k > 0$.

Least squares is done in the same way as for the 1D-DCT.

The DCT in Matlab

```
>> y = dct(x)
```

When x is a matrix it does the DCT of each column of x.

Matlab does not use the DCT matrix to compute the DCT transform, it maximizes efficiency by using the FFT.

We can get the DCT matrix from

```
>> C = dct(eye(n))
```

To do the 2D-DCT transform,

```
>> Y = dct2(X)
```

Image compression

Images consist of pixels, represented by numbers (or vectors for color images). DCT least squares reduces the information (bits) required while maintaining the quality of the picture to the human eye.

A grayscale picture represents the grayness of each pixel by an integer between 0 and 255.

A picture containing $n \times m$ pixels will have $n \times m$ integers of information.

To import a picture in Matlab and get its matrix

```
>> ximage = imread('picture_name.jpg');
>> x = double(ximage);
```

Example in Matlab

Grayscale picture with pixels in a 306×224 grid.

To get the image matrix:

```
>> xin = imread('hippo1.jpg');
```

To render the grayscale image in Matlab:

```
>> imagesc(xin)
>> colormap(gray)
>> axis image
```

When working with the image matrix it is a good practice to subtract 128 (256/2) in order to center the values around 0. This contributes to better numerical stability.

To convert back to JPG:

```
>> x = double(xin);
>> x=uint8(x);
```

```
>> imwrite(x,'hippo2.jpg','jpg')
```

Crude compression: replace each 4×3 pixel block by its average value

```
xc=x1-128;
for i=1:56
    for j=1:102
        A=xc(4*(i-1)+1:4*i,3*(j-1)+1:3*j);
        mv=mean(mean(A));
        xc(4*(i-1)+1:4*i,3*(j-1)+1:3*j)=mv;
    end
end
xc=xc+128;
```

Goes from 68,544 bytes of information to 5,712.

Grayscale picture with pixels in a 102×56 grid. Compressed by averaging.

DCT compression

```
x1 = imread('hippo1.jpg'); x1=double(x1); xc=x1-128;
for i=1:56
    for j=1:102
        A=xc(4*(i-1)+1:4*i,3*(j-1)+1:3*j);
        Y=dct2(A);
        Y(1,3)=0;
        Y(2,2:3)=0;
        Y(3:4,:)=0;
        Y=round(Y);
        xc(4*(i-1)+1:4*i,3*(j-1)+1:3*j)=idct2(Y);
    end
end
xc=xc+128; xc=uint8(xc); imwrite(xc,'hippo3.jpg','jpg')
```

Grayscale picture with pixels in a 102×56 grid. Compressed by 2D-DCT.

Quantization

Instead of zeroing the lower right hand side of the transform matrix we assign fewer bits to store the information there. We want to compress a range of values to a single value. To do this we do the following:

- $z = \text{round}(y/q)$ (rounding to the nearest integer)
- $y_q = qz$

For instance, if we quantize the numbers between 30 and 33 modulo 4,

$$4 * \text{round}([30\ 31\ 32\ 33]/4) = 4 * \text{round}[7.5\ 7.75\ 8\ 8.25] = 32$$

The size of the maximum error is $q/2 = 2$. In the example above, the maximum error occurs when $y = 30$ and it is equal to -2.

Quantization matrix

The matrix Q formed by the numbers q_{kl} that regulate how many bits are assigned to each entry of the transform matrix Y is the quantization matrix. Matrix Y is replaced by the compressed matrix

$$Y_Q = \text{round}\left(\frac{y_{kl}}{q_{kl}}\right)$$

The rounding is where the loss occurs and makes this method a lossy compression. To decompress, the elements of Y_Q are multiplied by the corresponding elements of Q.

The larger q_{kl}, the more the loss and the greater the compression.

Linear quantization

We will now work with quantization matrices of dimension 8×8. Linear quantization is defined by

$$\boxed{q_{kl} = 8p(k+l+1) \quad \text{for} \quad 0 \leq k, l \leq 7}$$

for the loss parameter p.

A small loss parameter gives a better reconstruction (less compression).

The Hilbert matrix has entries $\boxed{h_{kl} = 1/(k+l+1)}$ and can be used directly for linear quantization in Matlab:

```
>> Yq = round(Y.*hilb(8)/(8*p));
```

To reconstruct, we need to divide pointwise by the Hilbert matrix:

```
>> Xq = 8*p*Yp./hilb(8);
```

Hilbert matrix

The Hilbert matrix acts as a low pass filter, reducing the value of the coefficients of the higher frequencies. These are located in the lower right corner of the transform matrix.

$$\begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 \\ 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 \\ 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 \\ 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 \end{pmatrix}$$

For instance, if one of the 8×8 submatrices is

```
A =
 40    40    40    37    39    36    41    38
193   195   159    34    25    29    43    35
229   224   194    34    33    29    29    27
225   224   211    24    44    36    19    32
 19    29    42   238   226   136    29    13
 16    32    62   223   228   143    29    24
 18    26    60   234   238   150    24    20
 14    16    50   227   228   143    16    15
```

after undergoing linear quantization with p=2 becomes

3	1	1	1	0	0	0	0
6	4	2	0	0	0	0	0
5	4	2	0	0	0	0	0
4	3	2	0	0	0	0	0
0	0	0	2	2	1	0	0
0	0	0	2	1	1	0	0
0	0	0	1	1	1	0	0
0	0	0	1	1	1	0	0

Best quantization matrix

The matrix that experimentally gives the best results for the human eye is

$$Q_Y = p \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

It gives almost perfect reconstruction for $p = 1$.

Compression due to quantization

p	bits/pixel
0	8
1	3.89
2	2.98
4	2.30

Color images

Each pixel is assigned 3 numbers, for red, green and blue intensity. The matrix of a color image is a tensor of dimension $n \times m \times 3$.
One way to treat color compression is to work independently with the matrix for each color (R, G and B) and then reassemble the tensor.
There are more sophisticated ways of dealing with this problem, in particular by defining three matrices

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ U &= B - Y \\ V &= R - Y, \end{aligned}$$

applying a certain quantization matrix to each and then reassembling.

Lossless compression

Lossy compression occurs because of quantization of the transform matrix.

There is a tradeoff between the amount of compression and the amount of accuracy. The smaller the resulting file, the bigger the loss of accuracy.

After quantization, we can further compress without losing any more accuracy.

This is done by efficient coding of the transformed image.