Program 3

A081605 清華研 張睿

Outline

- Test images
- Comparison of different block sizes
- Information packing ability
- Comparison of different quantization schemes with qualitative comparison
    - Scheme 1: different k
    - Scheme 2: different k
    - Scheme 1 vs scheme 2 (with the same k)
    - Scheme 3: different number of bits
- Code listing

Test images

- Image 1: more spatial details

- Image 2: less spatial details

Comparison of different block sizes

- With 1st quantization scheme: (remaining the left-upper triangle of the coefficients map, i.e. remaining the first half of the coefficients)

- Image 1: DCT

Block size	4	8	16	32
eRMS	146.2	171.0	183.3	189.1
SNR	121.6	103.8	96.7	93.8

- Image 1: WHT

Block size	4	8	16	32
eRMS	152.7	185.1	202.4	212.0
SNR	116.4	95.8	87.6	83.5

- Image 1: DFT

Block size	4	8	16	32
eRMS	138.1	171.5	215.6	284.7
SNR	128.1	102.4	80.8	60.6

- For all the three transforms, smaller sub-image sizes perform better on reconstruction.
- To find the reason, I did a further test and found that for larger block size, there are less “first-half largest coefficients” falls exactly in the upper-left triangle of the coefficient map. For block sizes 4, 8, 16, 32, the ratios are 38.1%, 33.8%, 31.7%, 30.5%, respectively. (test with DCT)

- Image 2: DCT

Block size	4	8	16	32
eRMS	6.4	7.6	7.7	8.3
SNR	1233.3	1049.5	1034.2	958.0

- Image 2: WHT

Block size	4	8	16	32
eRMS	6.7	8.0	8.8	9.1
SNR	1194.1	998.6	905.2	871.2

- Image 2: DFT

Block size	4	8	16	32
eRMS	5.3	6.5	9.3	13.5
SNR	1491.1	1225.2	850.4	587.0

- Same as image 1: smaller sub-image sizes perform better on reconstruction.
- The ratio mentioned above for image 2 are 35.5%, 31.2%, 28.9%, 27.8%, respectively.
- The reconstruction error is much lower than image 1 possibly because the spatial details are much less in the image 2.

- With 2nd quantization scheme: (keep the first-half largest coefficients for each sub-images)
    - Image 1: DCT

Block size	4	8	16	32
eRMS	52.0	38.2	35.4	35.9
SNR	343.7	467.3	505.5	497.6

    - Image 1: WHT

Block size	4	8	16	32
eRMS	59.4	45.4	41.2	42.6
SNR	300.6	393.3	433.9	420.0

    - Image 1: DFT

Block size	4	8	16	32
eRMS	98.1	83.4	76.0	75.2
SNR	181.6	213.7	234.7	237.2

    - Opposite from the result of scheme 1, the trend is larger blocks lead to smaller reconstruction errors.

    - Image 2: DCT

Block size	4	8	16	32
eRMS	3.0	2.0	1.6	1.5
SNR	2630.1	3986.0	4889.4	5323.5

    - Image 2: WHT

Block size	4	8	16	32
eRMS	4.5	3.9	2.7	2.1
SNR	1763.9	2037.2	2879.2	3683.7

    - Image 2: DFT

Block size	4	8	16	32
eRMS	6.3	5.4	4.5	3.7
SNR	1257.4	1470.2	1743.0	2160.1

    - The trend is same as image 1, but the reconstruction errors are much smaller.

Information packing ability

- I follow the mean square error with assumption in the textbook to compare the information packing ability of the different transforms.

$$
\begin{aligned}
e_{ms} &= E\left\{\left\|\mathbf{G}-\hat{\mathbf{G}}\right\|^2\right\} \\
&= E\left\{\left\|\sum_{u=0}^{n-1}\sum_{v=0}^{n-1}T(u,v)\mathbf{S}_{uv}-\sum_{u=0}^{n-1}\sum_{v=0}^{n-1}\chi(u,v)T(u,v)\mathbf{S}_{uv}\right\|^2\right\} \\
&= E\left\{\left\|\sum_{u=0}^{n-1}\sum_{v=0}^{n-1}T(u,v)\mathbf{S}_{uv}\left[1-\chi(u,v)\right]\right\|^2\right\} \\
&= \sum_{u=0}^{n-1}\sum_{v=0}^{n-1}\sigma^2_{T(u,v)}\left[1-\chi(u,v)\right]
\end{aligned}
$$

where

$$
\chi(u,v)=\begin{cases}0 & \text{if } T(u,v) \text{ satisfies a specified truncation criterion}\\ 1 & \text{otherwise}\end{cases}
$$

is the coefficient masking function and $Sigma_{T(u,v)}$ ^2 is the variance of the coefficient at transform location (u,v). The final simplification is based on the orthonormal nature of the basis images and the assumption that the pixels of G are generated by a random process with zero mean and known covariance.

- My setting: block size=8, keep only the first 32 coefficients for quantization.
- The e_{ms} results show that DCT has the best information packing ability.
  - Image 1, DCT: 10941.5
  - Image 1, WHT: 11842.8
  - Image 1, DFT: 16691.3
  - Image 2, DCT: 488.0
  - Image 2, WHT: 512.9
  - Image 2, DFT: 672.8

Comparison of different quantization schemes

- Scheme 1: keep only the first k coefficients (block size=8; k follows the zig-zag fashion)
  - Image 1: DCT

k	6	15	32	54
eRMS	600.4	402.7	171.0	50.8
SNR	28.8	43.5	103.8	351.9

  - Image 1: WHT

k	6	15	32	54
eRMS	663.1	454.0	185.1	56.4
SNR	26.0	38.5	95.8	316.8

  - Image 1: DFT

k	6	15	32	54
eRMS	653.2	489.8	171.5	45.5
SNR	23.0	34.9	102.4	390.7

- As expected, larger k lead to smaller reconstruction error.
- Image 1: Qualitative comparison of DCT: k = 32 is better enough.

Original k = 6 k = 15 k = 32 k = 54

- Image 2: Qualitative comparison of DCT: the differences in the sky part are subtle in all settings.

Original k = 6 k = 15 k = 32 k = 54

- Scheme 2: keep only the coefficients of the k largest coefficients (block size=8)
  - Image 1: DCT

k	6	15	32	54
eRMS	379.9	167.3	38.2	1.2
SNR	46.2	106.1	467.3	14445.6

  - Image 1: WHT

k	6	15	32	54
eRMS	416.5	189.9	45.4	1.7
SNR	42.0	93.4	393.3	10051.3

  - Image 1: DFT

k	6	15	32	54
eRMS	470.7	251.3	83.4	7.2
SNR	37.0	70.3	213.7	2471.6

  - As expected, larger k lead to smaller reconstruction error.
  - Image 1: Qualitative comparison of DCT: similar with scheme 1, k = 32 is good enough.

Original k = 6 k = 15 k = 32 k = 54

- Scheme 1 vs scheme 2
  - As expected, scheme 2 has a much better performance than scheme 1 due to remaining the basis blocks with more importance. However, the computation is heavier due to like sorting.
  - Image 1: Qualitative comparison of DCT: with the same k, scheme 2 is better


k = 15, scheme 1 k = 15, scheme 2

- Scheme 3: distribute different number of bits to different coefficients (comparing different number of total bits used) (block size=8)
  - Image 1: DCT

total	16	64	128	512	1024
eRMS	967.3	759.4	600.2	394.0	299.2
SNR	17.4	22.3	25.1	44.0	57.7

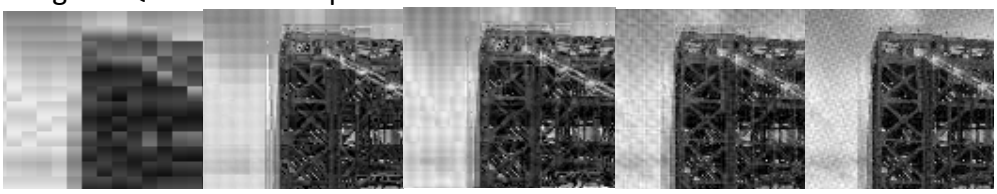
  - Image 1: WHT

total	16	64	128	512	1024
eRMS	1063.9	797.4	690.7	381.0	289.7
SNR	15.7	21.3	24.7	45.5	59.6

  - Image 1: DFT

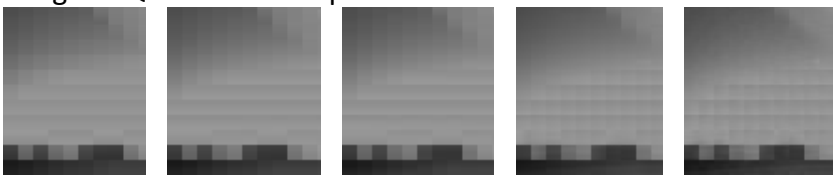
total	16	64	128	512	1024
eRMS	509.2	586.5	510.1	425.2	352.8
SNR	33.9	29.2	33.7	40.5	48.9

  - As expected, more bits lead to better performance.
  - Image 1: Qualitative comparison of DCT



k = 16 k = 64 k = 128. k = 512 k = 1024

  - Image 2: Qualitative comparison of DCT



k = 16. k = 64. K = 128 k = 512 k = 1024

  - Comparison of the bits distribution result of image 1 and image 2 (DCT, total bit = 1024): The distribution of image 2 is more concentrated to upper left, and the rest part is more sparse because the image 2 is much more smoother
    - Image 1:

```
[[809.  26.  10.   8.   7.   5.   3.   3.]
 [ 37.   7.   4.   3.   2.   2.   2.   2.]
 [ 11.   5.   3.   2.   2.   2.   1.   1.]
 [  5.   3.   2.   2.   2.   1.   1.   1.]
 [  3.   3.   2.   2.   2.   1.   1.   1.]
 [  2.   2.   2.   1.   1.   1.   1.   1.]
 [  2.   2.   2.   1.   1.   1.   1.   1.]
 [  2.   1.   2.   2.   1.   1.   1.   1.]]
```

- Image 2:

```
[[952.   3.   1.   0.   0.   0.   0.   0.]
 [ 43.   1.   0.   0.   0.   0.   0.   0.]
 [  5.   1.   0.   0.   0.   0.   0.   0.]
 [  3.   0.   0.   0.   0.   0.   0.   0.]
 [  1.   0.   0.   0.   0.   0.   0.   0.]
 [  1.   0.   0.   0.   0.   0.   0.   0.]
 [  1.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.]]
```

Code listing

compress.py

blockTransform.py

compress.py

```
import numpy as np
import math
import cv2
from blockTransform import reconstruct, transform
import argparse
import os
import pickle


if __name__ == "__main__":
    # reference:
https://docs.python.org/2/howto/argparse.html#introducing-
optional-arguments
    parser = argparse.ArgumentParser()
    # positional argument
    parser.add_argument('input')
    # optional arguments
    parser.add_argument('-t', '--transform', default='dct')
    parser.add_argument('-b', '--blockSize', type=int, default=8)
    parser.add_argument('-q', '--quantizeId', type=int,
default=1)
    parser.add_argument('-p', '--quantizePara', type=int,
default=8)
    args = parser.parse_args()

    inputNameNoExt = os.path.splitext(args.input)[0]
    inputExt = os.path.splitext(args.input)[1]

    resizeFolder = 'resize'
    resizeName = inputNameNoExt + '_resize' + inputExt
    resizePath = os.path.join(resizeFolder, resizeName)
    outputFolder = 'output'
    outputName = inputNameNoExt + '_' + args.transform + '_' +
```

```
str(args.blockSize) + \
                '_' + str(args.quantizeId) + '_' +
str(args.quantizePara) + '.bmp'
    outputPath = os.path.join(outputFolder, outputName)

    blockSize = args.blockSize

    img = cv2.imread(args.input, cv2.IMREAD_GRAYSCALE)
    # print(img.shape) # (2160, 3840)
    img = cv2.resize(img, (640, 320))
    # print(img.shape) # (320, 640)
    cv2.imwrite(resizePath, img)

    # get size of the image
    (h, w) = img.shape

    height = h
    width = w
    h = np.float32(h)
    w = np.float32(w)

    nbh = math.ceil(h / blockSize)
    nbh = np.int32(nbh)

    nbw = math.ceil(w / blockSize)
    nbw = np.int32(nbw)

    ### Pad the image
    # height of padded image
    H = blockSize * nbh
    # width of padded image
    W = blockSize * nbw
    padded_img = np.zeros((H, W))
    padded_img[0:height, 0:width] = img[0:height, 0:width]

    # the last two dimension of imgCof are corresponding to u, v,
respectively
    if args.transform == 'dft':
        imgCof = np.zeros((nbh, nbw, blockSize, blockSize),
dtype=complex)
    else:
        imgCof = np.zeros((nbh, nbw, blockSize, blockSize))
    print("nbh:", nbh, " nbw:", nbw)

    for i in range(nbh):
        # Compute start and end row index of the block
        row_ind_1 = i * blockSize
        row_ind_2 = row_ind_1 + blockSize
        for j in range(nbw):
            # Compute start & end column index of the block
            col_ind_1 = j * blockSize
            col_ind_2 = col_ind_1 + blockSize
```

```
            block = padded_img[row_ind_1: row_ind_2, col_ind_1:
col_ind_2]

            imgCof[i, j] = transform(block, blockSize,
args.transform)

    ### information packing ability
    # print(imgCof[0, 0])
    # print(imgCof[1, 1])
    # print(imgCof[2, 2])
    # print("mean:\n", np.mean(imgCof, axis=(0, 1)))
    cofVar = np.var(imgCof, axis=(0, 1))
    # print("coefficient variances across sub-images:\n", cofVar)
    # pickle.dump(cofVar, open("dft.pkl", "wb"))

    # textbook p581
    mask = np.add(*np.indices((blockSize, blockSize)))
    mask = mask >= blockSize
    ems = (cofVar * mask).sum()
    print("ems:", ems)

    ### quantization
    if args.quantizeId == 0:
        pass
    elif args.quantizeId == 1:
        # Keep only the first k coefficients
        keepSize = args.quantizePara
        mask = np.add(*np.indices((blockSize, blockSize)))
        mask = mask < keepSize
        imgCof = imgCof * mask

    elif args.quantizeId == 2:

        # Keep only the coefficients with the k largest
coefficients
        k = args.quantizePara
        inn = list()
        for i in range(nbh):
            for j in range(nbw):
                thres = np.sort(np.absolute(imgCof[i,
j]).flatten())[-k]
                # print(imgCof[i, j, 5, 5], np.absolute(imgCof[i,
j, 5, 5]))
                mask = np.absolute(imgCof[i, j]) >= thres
                imgCof[i, j] = imgCof[i, j] * mask

                mask2 = np.add(*np.indices((blockSize,
blockSize)))
                mask2 = mask2 < blockSize
                inn.append((mask * mask2).sum())
        print((sum(inn) / len(inn)) / blockSize**2)
```

```
    elif args.quantizeId == 3:

        # Distribute a fixed number of bits to all the
coefficients (me: at the same position)
        # according to the logarithm of coefficient variances
        # also mentioned the mail from teacher

        totalBits = args.quantizePara
        qi = np.var(imgCof, axis=(0, 1))
        ni = np.round(totalBits * qi / qi.sum()) # ni.shape:
(blockSize, blockSize)
        print(ni)
        # print(ni.sum())
        # yet: the case if ni.sum() > totalBits

        # print("before:\n", imgCof[:, :, -1, -1].mean())
        # print(imgCof[:, :, -1, -1].max())
        # print(imgCof[:, :, -1, -1].min())

        # count = 0

        if args.transform == 'dft':
            for i in range(blockSize):
                for j in range(blockSize):

                    cofMin = np.percentile(imgCof.real[:, :, i,
j], 5)
                    cofMax = np.percentile(imgCof.real[:, :, i,
j], 95)
                    cofRange = cofMax - cofMin
                    if ni[i, j] == 0:
                        imgCof.real[:, :, i, j] = 0
                        continue
                    intvlWidth = cofRange / (2 ** ni[i, j] / 2)
                    tmpCof = imgCof.real[:, :, i, j].copy()
                    tmpCof[tmpCof > cofMax] = cofMax
                    tmpCof[tmpCof < cofMin] = cofMin
                    imgCof.real[:, :, i, j] = cofMin + intvlWidth
* ((tmpCof - cofMin) // intvlWidth + 0.5)

                    cofMin = np.percentile(imgCof.imag[:, :, i,
j], 5)
                    cofMax = np.percentile(imgCof.imag[:, :, i,
j], 95)
                    cofRange = cofMax - cofMin
                    if ni[i, j] == 0 or (i == 0 and j == 0):
                        imgCof.imag[:, :, i, j] = 0
                        continue
                    intvlWidth = cofRange / (2 ** ni[i, j] / 2)
                    if i == 0 and j == 0:
                        print('intvlWidth:', intvlWidth)
```

```
                    tmpCof = imgCof.imag[:, :, i, j].copy()
                    tmpCof[tmpCof > cofMax] = cofMax
                    tmpCof[tmpCof < cofMin] = cofMin
                    imgCof.imag[:, :, i, j] = cofMin + intvlWidth
* ((tmpCof - cofMin) // intvlWidth + 0.5)

        else:
            for i in range(blockSize):
                for j in range(blockSize):

                    cofMin = np.percentile(imgCof[:, :, i, j], 5)
                    cofMax = np.percentile(imgCof[:, :, i, j],
95)
                    cofRange = cofMax - cofMin
                    if ni[i, j] == 0:
                        imgCof[:, :, i, j] = 0
                        continue
                    intvlWidth = cofRange / 2 ** ni[i, j]
                    tmpCof = imgCof[:, :, i, j].copy()
                    tmpCof[tmpCof > cofMax] = cofMax
                    tmpCof[tmpCof < cofMin] = cofMin
                    imgCof[:, :, i, j] = cofMin + intvlWidth *
((tmpCof - cofMin) // intvlWidth + 0.5)

        # print("after:\n", imgCof[:, :, -1, -1].mean())
        # print(imgCof[:, :, -1, -1].max())
        # print(imgCof[:, :, -1, -1].min())

    ### reconstruction
    reconsImg = np.zeros((H, W))

    for i in range(nbh):

        # Compute start and end row index of the block
        row_ind_1 = i * blockSize
        row_ind_2 = row_ind_1 + blockSize

        for j in range(nbw):
            # Compute start & end column index of the block
            col_ind_1 = j * blockSize
            col_ind_2 = col_ind_1 + blockSize

            if args.transform == 'dft':
                # print(imgCof[i, j])
                reconsSubImg = reconstruct(imgCof[i, j],
blockSize, 'dft')
                # print(np.imag(reconsSubImg))

                reconsImg[row_ind_1: row_ind_2, col_ind_1:
col_ind_2] += np.real(reconsSubImg)
                if args.quantizeId == 0 and
np.sum(np.imag(reconsSubImg) > 1) > 0:
```

```
                    print("unexpected sub-image reconstruction")
                    print('np.imag(reconsSubImg).max():',
np.imag(reconsSubImg).max())
                    exit()
            else:
                reconsImg[row_ind_1: row_ind_2, col_ind_1:
col_ind_2] += \
                    reconstruct(imgCof[i, j], blockSize,
args.transform)

    cv2.imwrite(outputPath, reconsImg)

    #### fidelity

    # test
    # reconsImg = reconsImg * 0

    errMap = reconsImg[0:height, 0:width] - img
    eRMS = (1 / (height * width)) * np.sum(errMap**2)
    print("eRMS:", eRMS)
    snr = np.sum(reconsImg[0:height, 0:width]**2) /
np.sum(errMap**2)
    print("SNR:", snr)
```

blockTransform.py

```
import numpy as np
import math


# textbook p487 Eq.7-83
def dctInverseTransformationKernel(x, u, N):
    if u == 0:
        return (1 / N)**0.5 * math.cos((2 * x + 1) * u * math.pi
/ (2 * N))
    elif 1 <= u <= N - 1:
        return (2 / N) ** 0.5 * math.cos((2 * x + 1) * u *
math.pi / (2 * N))
    else:
        print("Error from dctInverseTransformationKernel()")
        exit()


# textbook p498 Eq.7-102
def whtInverseTransformationKernel(x, u, N):

    ### calculate the power
    n = int(round(math.log2(N)))
    power = 0

    # _x means that the input of this function mentioned in the
```

```
textbook is fix to x
    b_x = [int(i) for i in bin(x)[2:]]
    b_x.reverse()
    while len(b_x) < n:
        b_x.append(0)

    b_u = [int(i) for i in bin(u)[2:]]
    b_u.reverse()
    while len(b_u) < n:
        b_u.append(0)

    p_u = [0] * n
    p_u[0] = b_u[n - 1]
    for i in range(1, n):
        p_u[i] = b_u[n - i] + b_u[n - i - 1]

    for i in range(n):
        power += b_x[i] * p_u[i]
    power = power % 2
    return (1 / N)**0.5 * (-1)**power


# textbook p476 Eq.7-56
def dftInverseTransformationKernel(x, u, N):
    p = 2 * math.pi * u * x / N
    return 1 / N**0.5 * (math.cos(p) + 1j * math.sin(p))


# textbook p468 Eq.7-22
def basisVector(u, N, tf):
    if tf == 'dct':
        su = np.zeros((N, 1))
        for i in range(N):
            su[i, 0] = dctInverseTransformationKernel(i, u, N)
    elif tf == 'wht':
        su = np.zeros((N, 1))
        for i in range(N):
            su[i, 0] = whtInverseTransformationKernel(i, u, N)
    elif tf == 'dft':
        su = np.zeros((N, 1), dtype=complex)
        for i in range(N):
            su[i, 0] = dftInverseTransformationKernel(i, u, N)
    return su


# textbook p468 Eq.7-24
def transformationMatrix(N, tf='dct'):
    if tf in ['dct', 'wht']:
        A = np.zeros((N, N))
    elif tf == 'dft':
        A = np.zeros((N, N), dtype=complex)
```

```
    for i in range(N):
        A[i, :] = basisVector(i, N, tf).T
    if tf == 'dft':
        A = np.conj(A)
        # should meet textbook p485 Fig.7.7(a) when N, the
blockSize, is 8
        # print("8**0.5 * dft matrix\n", 8**0.5 * A)
    return A


# textbook p470 Eq.7-35, not used
def dct(F, N):
    A = transformationMatrix(N, 'dct')
    return (A.dot(F)).dot(A.T)


# textbook p470 Eq.7-35, not used
def wht(F, N):
    A = transformationMatrix(N, 'wht')
    return (A.dot(F)).dot(A.T)


# textbook p473 Eq.7-41, not used
def dft(F, N):
    A = transformationMatrix(N, 'dft')
    # not sure if the textbook has no mistake
    # Astar = np.conj(A)
    # return (Astar.dot(F)).dot(Astar.T)
    return (A.dot(F)).dot(A.T)


# the combination version of the three functions above
def transform(F, N, tf="dct"):
    A = transformationMatrix(N, tf)
    return (A.dot(F)).dot(A.T)


# Set10 slide p24, not used
def generateDctBasis(N):
    basis = np.zeros((N, N, N, N)) # (x, y, u, v)
    tmp1D = np.zeros((N, N))
    for x in range(N):
        for u in range(N):
            tmp1D[x, u] = dctInverseTransformationKernel(x, u, N)
    for x in range(N):
        for y in range(N):
            for u in range(N):
                for v in range(N):
                    basis[x, y, u, v] = tmp1D[x, u] * tmp1D[y, v]
    return basis
```

```python
# textbook p499 Eq.7-107 "separable", not used
def generateWhtBasis(N):
    basis = np.zeros((N, N, N, N)) # (x, y, u, v)
    tmp1D = np.zeros((N, N))
    for x in range(N):
        for u in range(N):
            tmp1D[x, u] = whtInverseTransformationKernel(x, u, N)
    for x in range(N):
        for y in range(N):
            for u in range(N):
                for v in range(N):
                    basis[x, y, u, v] = tmp1D[x, u] * tmp1D[y, v]
    return basis

# not used
def generateDftBasis(N):
    basis = np.zeros((N, N, N, N), dtype=complex) # (x, y, u, v)
    tmp1D = np.zeros((N, N), dtype=complex)
    for x in range(N):
        for u in range(N):
            tmp1D[x, u] = dftInverseTransformationKernel(x, u, N)
            # print(tmp1D[x, u])
    for x in range(N):
        for y in range(N):
            for u in range(N):
                for v in range(N):
                    basis[x, y, u, v] = tmp1D[x, u] * tmp1D[y, v]
    return basis


# textbook p470 Eq.7-36, not used
def reconstructDct(subImgCof, N):
    A = transformationMatrix(N, 'dct')
    return A.T.dot(subImgCof).dot(A)


# textbook p470 Eq.7-36, not used
def reconstructWht(subImgCof, N):
    A = transformationMatrix(N, 'wht')
    return A.T.dot(subImgCof).dot(A)


# textbook p473 Eq.7-42, not used
def reconstructDft(subImgCof, N):
    A = transformationMatrix(N, 'dft')
    A = np.conj(A)
    return A.T.dot(subImgCof).dot(A)


# the combination version of the three functions above
def reconstruct(subImgCof, N, tf="dct"):
    A = transformationMatrix(N, tf)
```

```
    if tf == "dft":
        A = np.conj(A)
    return A.T.dot(subImgCof).dot(A)
```