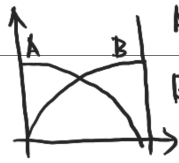
6. Image Processing

Image Load/Save

```
class CKhuGleSignal {      // KhuGleSignal.h
public:
  …
  int m_nW, m_nH;
  unsigned char **m_Red, **m_Green, **m_Blue;
  …
  void ReadBmp(char *FileName);
  bool SaveBmp(char *FileName);
};
```

A: 낮은 주파수 살려줌

B: 높은 〃

Time Frequency

$f \cdot g \rightarrow F * g$

$f * g \rightarrow F \cdot g$

Convolution 한 것과 동일한 처리

Mean Filter

→ Convolution 하면 변화가 심한 부분이 뭉개짐

Convolutional Mask

t만큼 shift 후 y축 대칭

$$(f*g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

반전 큰 의미 없음

Convolution

τ축에 대해서 원래 Signal을 놓고

1/9

1	1	1
1	1	1
1	1	1

1/25

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

경계가 점점 무뎌짐

A 필터

Edge

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

변화가 심한 부분이 잘 나타남

B 필터

코드에서 회색 낮은 값
흰색 높은 값
검은색 음수

실수값만 가짐

2D DCT

cosine

원래 Signal → 복소수 형태

$$F(u,v) = C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

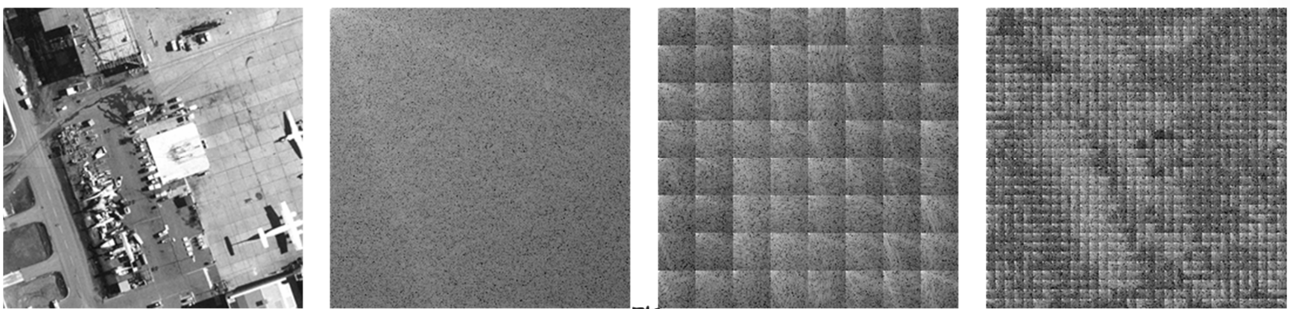
x축 y축

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

$$C(\alpha) = \begin{cases} \sqrt{\frac{1}{N}} & \alpha = 0 \\ \sqrt{\frac{2}{N}} & \alpha = 1, 2, \cdots, N-1 \end{cases}$$

우함수에 대한 연산만 있음

높은 값



낮은 값

KhuGleBase.cpp (1)

```
void DCT2D(double **Input, double **Output, int nW, int nH, int nBlockSize) {
  int x, y;
  int u, v;
  int BlockX, BlockY;

  for(v = 0 ; v < nH ; v++)
    for(u = 0 ; u < nW ; u++)
      Output[v][u] = 0;

  for(BlockY = 0 ; BlockY < nH-nBlockSize+1 ; BlockY += nBlockSize)
    for(BlockX = 0 ; BlockX < nW-nBlockSize+1 ; BlockX += nBlockSize) {
      for(v = 0 ; v < nBlockSize ; v++)
        for(u = 0 ; u < nBlockSize ; u++) {
          Output[BlockY+v][BlockX+u] = 0;
          for(y = 0 ; y < nBlockSize ; y++)
            for(x = 0 ; x < nBlockSize ; x++) {
              Output[BlockY+v][BlockX+u] +=
                Input[BlockY+y][BlockX+x]
                * cos((2*x+1)*u*Pi/(2.*nBlockSize))
                * cos((2*y+1)*v*Pi/(2.*nBlockSize));
            }
```

Block의 시작 위치

KhuGleBase.cpp (2)

```
          if(u == 0)
            Output[BlockY+v][BlockX+u] *= sqrt(1./nBlockSize);
          else
            Output[BlockY+v][BlockX+u] *= sqrt(2./nBlockSize);

          if(v == 0)
            Output[BlockY+v][BlockX+u] *= sqrt(1./nBlockSize);
          else
            Output[BlockY+v][BlockX+u] *= sqrt(2./nBlockSize);
        }
    }
}
```

KhuGleBase.cpp (3)

```
void IDCT2D(double **Input, double **Output, int nW, int nH, int nBlockSize) {
  int x, y;
  int u, v;
  int BlockX, BlockY;

  for(y = 0 ; y < nH ; y++)
    for(x = 0 ; x < nW ; x++)
      Output[y][x] = 0;

  for(BlockY = 0 ; BlockY < nH-nBlockSize+1 ; BlockY += nBlockSize)
    for(BlockX = 0 ; BlockX < nW-nBlockSize+1 ; BlockX += nBlockSize) {
      for(y = 0 ; y < nBlockSize ; y++)
        for(x = 0 ; x < nBlockSize ; x++) {
          Output[BlockY+y][BlockX+x] = 0;
```


KhuGleBase.cpp (4)

```
          for(v = 0 ; v < nBlockSize ; v++)
            for(u = 0 ; u < nBlockSize ; u++) {
              double Cu, Cv;
              if(u == 0) Cu = sqrt(1./nBlockSize);
              else Cu = sqrt(2./nBlockSize);

              if(v == 0) Cv = sqrt(1./nBlockSize);
              else Cv = sqrt(2./nBlockSize);

              Output[BlockY+y][BlockX+x] +=
                Cu*Cv*Input[BlockY+v][BlockX+u]
                * cos((2*x+1)*u*Pi/(2.*nBlockSize))
                * cos((2*y+1)*v*Pi/(2.*nBlockSize));
            }
        }
    }
}
```

MSE/PSNR

- MSE: mean squared error

같은 신호에 대한 quality

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

noise: MSE

- PSNR: peak signal-to-noise ratio
  - Color: MSE of average mean

$$10\log_{10}\left(\frac{\text{Max}^2}{\text{MSE}}\right)$$

→ MSE가 제곱이라 얘도 함

KhuGleBase.cpp

```
double GetMse(unsigned char **I, unsigned char **O, int nW, int nH) {
  double Mse = 0;
  for(int y = 0 ; y < nH ; ++y)
    for(int x = 0 ; x < nW ; ++x)
      Mse += (I[y][x] - O[y][x])*(I[y][x] - O[y][x]);

  Mse /= nW*nH;
  return Mse;
}
double GetPsnr(unsigned char **IR, unsigned char **IG, unsigned char **IB,
  unsigned char **OR, unsigned char **OG, unsigned char **OB, int nW, int nH) {
  double MseR, MseG, MseB, Mse;

  MseR = GetMse(IR, OR, nW, nH);
  MseG = GetMse(IG, OG, nW, nH);
  MseB = GetMse(IB, OB, nW, nH);
  Mse = (MseR + MseG + MseB)/3.;

  if(Mse == 0) return 100.;
  return 10*log10(255*255 / Mse);
}
```

Main (1)

```
…
class CKhuGleImageLayer : public CKhuGleLayer {
public:
  CKhuGleSignal m_Image, m_ImageOut;
  CKhuGleImageLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0))
    : CKhuGleLayer(nW, nH, bgColor, ptPos) {}
  void DrawBackgroundImage();
};

void CKhuGleImageLayer::DrawBackgroundImage() {
  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(m_bgColor);
      m_ImageBgG[y][x] = KgGetGreen(m_bgColor);
      m_ImageBgB[y][x] = KgGetBlue(m_bgColor);
    }
```


Main (2)

```
  if(m_Image.m_Red && m_Image.m_Green && m_Image.m_Blue) {
    for(int y = 0 ; y < m_Image.m_nH && y < m_nH ; ++y)
      for(int x = 0 ; x < m_Image.m_nW && x < m_nW ; ++x) {
        m_ImageBgR[y][x] = m_Image.m_Red[y][x];
        m_ImageBgG[y][x] = m_Image.m_Green[y][x];
        m_ImageBgB[y][x] = m_Image.m_Blue[y][x];
      }
  }
  if(m_ImageOut.m_Red && m_ImageOut.m_Green && m_ImageOut.m_Blue) {
    int OffsetX = 300, OffsetY = 0;
    for(int y = 0 ; y < m_ImageOut.m_nH && y + OffsetY < m_nH ; ++y)
      for(int x = 0 ; x < m_ImageOut.m_nW && x + OffsetX < m_nW ; ++x) {
        m_ImageBgR[y + OffsetY][x + OffsetX] = m_ImageOut.m_Red[y][x];
        m_ImageBgG[y + OffsetY][x + OffsetX] = m_ImageOut.m_Green[y][x];
        m_ImageBgB[y + OffsetY][x + OffsetX] = m_ImageOut.m_Blue[y][x];
      }
  }
}
```

Main (3)

```
class CImageProcessing : public CKhuGleWin {
public:
  CKhuGleImageLayer *m_pImageLayer;
  CImageProcessing(int nW, int nH, char *ImagePath);
  void Update();
};
CImageProcessing::CImageProcessing(int nW, int nH, char *ImagePath)
  : CKhuGleWin(nW, nH) {
  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pImageLayer = new CKhuGleImageLayer(600, 420,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 30));
  m_pImageLayer->m_Image.ReadBmp(ImagePath);
  m_pImageLayer->m_ImageOut.ReadBmp(ImagePath);
  m_pImageLayer->DrawBackgroundImage();
  m_pScene->AddChild(m_pImageLayer);
}
void CImageProcessing::Update() {
  if(m_bKeyPressed['D'] || m_bKeyPressed['I'] || m_bKeyPressed['C']
    || m_bKeyPressed['E'] || m_bKeyPressed['M']) {
    bool bInverse = m_bKeyPressed['I'];
    bool bCompression = m_bKeyPressed['C'];
    bool bEdge = m_bKeyPressed['E'];
    bool bMean = m_bKeyPressed['M'];
```


Main (4)

```
    double **InputR = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    double **InputG = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    double **InputB = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);

    double **OutR = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    double **OutG = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    double **OutB = dmatrix(m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);

    for(int y = 0 ; y < m_pImageLayer->m_Image.m_nH ; ++y)
      for(int x = 0 ; x < m_pImageLayer->m_Image.m_nW ; ++x) {
        InputR[y][x] = m_pImageLayer->m_Image.m_Red[y][x];
        InputG[y][x] = m_pImageLayer->m_Image.m_Green[y][x];
        InputB[y][x] = m_pImageLayer->m_Image.m_Blue[y][x];
                      }
```

Main (5)

```
if(bEdge)  {
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      OutR[y][x] = OutG[y][x] = OutB[y][x] = 0.;
      if(x > 0 && x < m_pImageLayer->m_ImageOut.m_nW-1 &&
        y > 0 && y < m_pImageLayer->m_ImageOut.m_nH-1) {
        double Rx = InputR[y-1][x-1] + 2*InputR[y][x-1] + InputR[y+1][x-1]
          - InputR[y-1][x+1] - 2*InputR[y][x+1] - InputR[y+1][x+1];
        double Ry = InputR[y-1][x-1] + 2*InputR[y-1][x] + InputR[y-1][x+1]
          - InputR[y+1][x-1] - 2*InputR[y+1][x] - InputR[y+1][x+1];
        double Gx = InputG[y-1][x-1] + 2*InputG[y][x-1] + InputG[y+1][x-1]
          - InputG[y-1][x+1] - 2*InputG[y][x+1] - InputG[y+1][x+1];
        double Gy = InputG[y-1][x-1] + 2*InputG[y-1][x] + InputG[y-1][x+1]
          - InputG[y+1][x-1] - 2*InputG[y+1][x] - InputG[y+1][x+1];
        double Bx = InputB[y-1][x-1] + 2*InputB[y][x-1] + InputB[y+1][x-1]
          - InputB[y-1][x+1] - 2*InputB[y][x+1] - InputB[y+1][x+1];
        double By = InputB[y-1][x-1] + 2*InputB[y-1][x] + InputB[y-1][x+1]
          - InputB[y+1][x-1] - 2*InputB[y+1][x] - InputB[y+1][x+1];
        OutR[y][x] = sqrt(Rx*Rx + Ry*Ry);  OutG[y][x] = sqrt(Gx*Gx + Gy*Gy);
        OutB[y][x] = sqrt(Bx*Bx + By*By);
      }
    }
    std::cout << "Edge" << std::endl;
}
```


Main (6)

```
else if(bMean) {
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      OutR[y][x] = OutG[y][x] = OutB[y][x] = 0.;
      if(x > 0 && x < m_pImageLayer->m_ImageOut.m_nW-1 &&
        y > 0 && y < m_pImageLayer->m_ImageOut.m_nH-1) {
        for(int dy = -1 ; dy < 2 ; ++dy)
          for(int dx = -1 ; dx < 2 ; ++dx) {
            OutR[y][x] += InputR[y+dy][x+dx];   OutG[y][x] += InputG[y+dy][x+dx];
            OutB[y][x] += InputB[y+dy][x+dx];
          }
       }
    }
  std::cout << "Mean filter" << std::endl;
}
else {
  DCT2D(InputR,OutR,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  DCT2D(InputG,OutG,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  DCT2D(InputB,OutB,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  std::cout << "DCT" << std::endl;
}
```

Main (7)

```
if(!bInverse && ! bCompression) {
  double MaxR, MaxG, MaxB, MinR, MinG, MinB;
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      if(x == 0 && y == 0) {
        MaxR = MinR = OutR[y][x];
        MaxG = MinG = OutG[y][x];
        MaxB = MinB = OutB[y][x];
      }
      else {
        if(OutR[y][x] > MaxR) MaxR = OutR[y][x];
        if(OutG[y][x] > MaxG) MaxG = OutG[y][x];
        if(OutB[y][x] > MaxB) MaxB = OutB[y][x];

        if(OutR[y][x] < MinR) MinR = OutR[y][x];
        if(OutG[y][x] < MinG) MinG = OutG[y][x];
        if(OutB[y][x] < MinB) MinB = OutB[y][x];
      }
    }
```


CImageProcessing
Edge/Mean Output image

Main (8)

```
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      if(MaxR == MinR) m_pImageLayer->m_ImageOut.m_Red[y][x] = 0;
      else m_pImageLayer->m_ImageOut.m_Red[y][x]
             = (int)((OutR[y][x]-MinR)*255/(MaxR-MinR));
      if(MaxG == MinG) m_pImageLayer->m_ImageOut.m_Green[y][x] = 0;
      else m_pImageLayer->m_ImageOut.m_Green[y][x]
             = (int)((OutG[y][x]-MinG)*255/(MaxG-MinG));
      if(MaxB == MinB) m_pImageLayer->m_ImageOut.m_Blue[y][x] = 0;
      else m_pImageLayer->m_ImageOut.m_Blue[y][x]
             = (int)((OutB[y][x]-MinB)*255/(MaxB-MinB));
    }
}
```

Main (9)

```
else {
  if(bCompression) {
    for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
      for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
        if(x%8 > 3 || y %8 > 3) {
          OutR[y][x] = 0; OutG[y][x] = 0; OutB[y][x] = 0;
        }
      }
    std::cout << "Compression" << std::endl;
  }
  else
    std::cout << "Non compression" << std::endl;

  IDCT2D(OutR, InputR, m_pImageLayer->m_Image.m_nW,
    m_pImageLayer->m_Image.m_nH, 8);
  IDCT2D(OutG, InputG, m_pImageLayer->m_Image.m_nW,
    m_pImageLayer->m_Image.m_nH, 8);
  IDCT2D(OutB, InputB, m_pImageLayer->m_Image.m_nW,
    m_pImageLayer->m_Image.m_nH, 8);
```


Main (10)

```
  double MaxR, MaxG, MaxB, MinR, MinG, MinB;
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      if(x == 0 && y == 0) {
        MaxR = MinR = InputR[y][x];
        MaxG = MinG = InputG[y][x];
        MaxB = MinB = InputB[y][x];
      }
      else {
        if(InputR[y][x] > MaxR) MaxR = InputR[y][x];
        if(InputG[y][x] > MaxG) MaxG = InputG[y][x];
        if(InputB[y][x] > MaxB) MaxB = InputB[y][x];
        if(InputR[y][x] < MinR) MinR = InputR[y][x];
        if(InputG[y][x] < MinG) MinG = InputG[y][x];
        if(InputB[y][x] < MinB) MinB = InputB[y][x];
      }
   }
```

Main (11)

```
    for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
      for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
        if(MaxR == MinR) m_pImageLayer->m_ImageOut.m_Red[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Red[y][x]
          = (int)((InputR[y][x]-MinR)*255/(MaxR-MinR));→ 범위(0,255)로 만들기 위해
        if(MaxG == MinG) m_pImageLayer->m_ImageOut.m_Green[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Green[y][x]
          = (int)((InputG[y][x]-MinG)*255/(MaxG-MinG));
        if(MaxB == MinB) m_pImageLayer->m_ImageOut.m_Blue[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Blue[y][x]
          = (int)((InputB[y][x]-MinB)*255/(MaxB-MinB));
      }
  }
```

범위 제한을 안 하면

0~255를 벗어나면

음밑 → 255 오 255위 해서 그림에 이상한 부분이 생김

(int)(x+0.5) → 반올림

Main (12)

```
  if(bMean || bCompression || bInverse) {
    double Psnr = GetPsnr(m_pImageLayer->m_Image.m_Red,
      m_pImageLayer->m_Image.m_Green, m_pImageLayer->m_Image.m_Blue,
      m_pImageLayer->m_ImageOut.m_Red, m_pImageLayer->m_ImageOut.m_Green,
      m_pImageLayer->m_ImageOut.m_Blue,
      m_pImageLayer->m_Image.m_nW, m_pImageLayer->m_Image.m_nH);
    std::cout << Psnr << std::endl;
  }

  free_dmatrix(InputR, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
  free_dmatrix(InputG, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
  free_dmatrix(InputB, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
  free_dmatrix(OutR, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
  free_dmatrix(OutG, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
  free_dmatrix(OutB, m_pImageLayer->m_Image.m_nH,
    m_pImageLayer->m_Image.m_nW);
```

Main (13)

```
    m_pImageLayer->DrawBackgroundImage();
    m_bKeyPressed['D'] = m_bKeyPressed['I'] = m_bKeyPressed['C']
      = m_bKeyPressed['E'] = m_bKeyPressed['M'] = false;
  }

  m_pScene->Render();

  DrawSceneTextPos("Image Processing", CKgPoint(0, 0));

  CKhuGleWin::Update();
}
```


Main (14)

```
int main() {
  char ExePath[MAX_PATH], ImagePath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);
  int i;
  int LastBackSlash = -1;
  int nLen = strlen(ExePath);
  for(i = nLen-1 ; i >= 0 ; i--) {
    if(ExePath[i] == '\\') {
      LastBackSlash = i;
      break;
    }
  }
  if(LastBackSlash >= 0) ExePath[LastBackSlash] = '\0';
  sprintf(ImagePath, "%s\\%s", ExePath, "couple.bmp");

  CImageProcessing *pImageProcessing
    = new CImageProcessing(640, 480, ImagePath);
  KhuGleWinInit(pImageProcessing);
  return 0;
}
```

Practice IV (1)

- YCbCr
  - RGB→YCbCr
    - Y = 0.29900R+0.58700G+0.11400B
    - Cb = -0.16874R-0.33126G+0.50000B
    - Cr = 0.50000R-0.41869G-0.08131B
  - YCbCr→RGB
    - R=1.00000Y+1.40200Cr
    - G=1.00000Y-0.34414Cb-0.71414Cr
    - B=1.00000Y+1.77200Cb

Practice IV (2)

- Quantization

$$\hat{c}(x,y) = ROUND\left(\frac{c(x,y)}{q(x,y)}\right)$$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	66
14	13	16	24	40	57	69	57
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	36	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

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

Practice IV (3)

- Variable length code(VLC)

Huffman Coding

	Probability	1	2	3	4
A2	0.4	0.4	0.4	0.4	0.6
A6	0.3	0.3	0.3	0.3	0.4
A1	0.1	0.1	0.2	0.3	
A4	0.1	0.1	0.1		
A3	0.06	0.1			
A5	0.04				

	Code	1	2	3	4
A2	0.4 1	0.4 1	0.4 1	~~0.4~~ 1	0.6 ~~1~~ 0
A6	0.3 00	0.3 00	0.3 00	~~0.3~~ 00	~~0.4~~ ~~0~~ 1
A1	0.1 011	0.1 011	0.2 010	0.3 01	
A4	0.1 0100	~~0.1~~ 0100	0.1 011		
A3	0.06 01010	0.1 0101			
A5	0.04 01011				

Advanced Courses

- Png
- Animation

- Interpolation
- Anti-aliasing

- RMSE: root mean squared error
- MAE: mean absolute error
- MAPE: mean absolute percentage error
- SSIM: structural similarity index measure