DIP

PB16061024

May 16, 2019

Contents

DIP PB16061024

`inline int** paddingImg2(int** pixelmat, int mheight, int mwidth);` `// padding 2 pixels:` pad pad

`std::complex<double> * *dftmat(const int length, const bool inverse = false):` length×length dft

`float** haarmat(const int length):` length×length haar

`template<typename T = std::complex<double>> T * *matprod(T * *mat1, int x1, int y1, T * *mat2, int x2, int y2, const bool transpose = false):` mat1 mat2 (x1,y1) (x2,y2)

`template<typename T> T clip3(const T & min, const T & max, const T & t):` t [min,max] min max t

`template<typename T> std::complex<double>** dft2(T** pixelmat, int mheight, int mwidth):` dft

`std::complex<double>** idft2(std::complex<double> * *pixelComp, int mheight, int mwidth):` dft

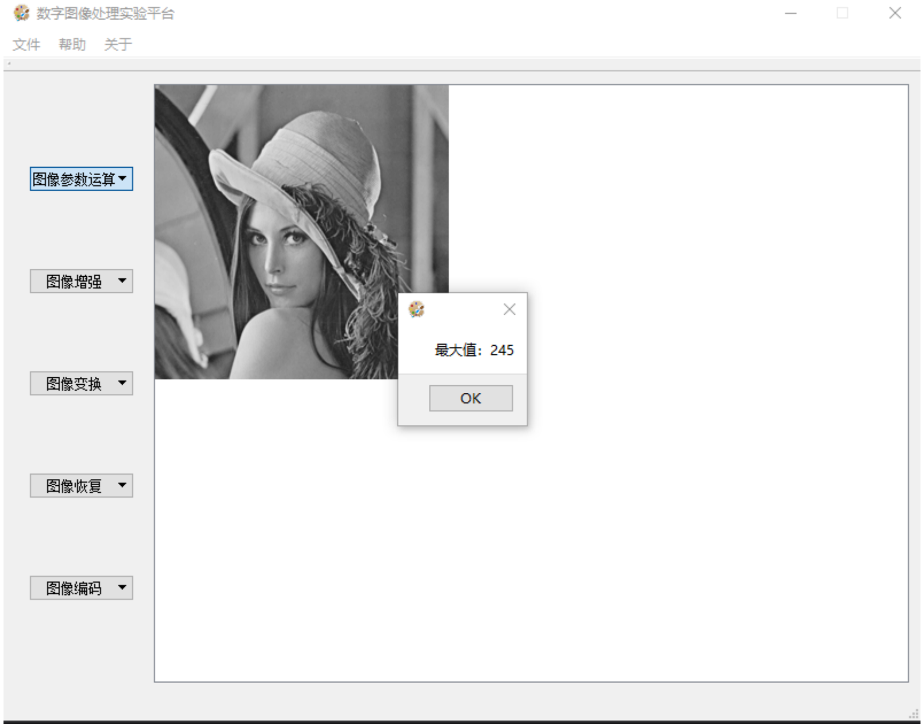
`float** haar2(int** pixelmat, int mheight, int mwidth):` haar

```
inline int** paddingImg2(int** pixelmat, int mheight, int mwidth) // padding 2 pixels
{
  int** imgWithPadding = new int* [mheight + 4];
  for (int i = 0; i < mheight + 4; i++) // copy content
  {
    imgWithPadding[i] = new int[mwidth + 4];
    if (i < 2)
    {
      std::copy(pixelmat[1 - i], pixelmat[1 - i] + mwidth, imgWithPadding[i] + 2);
    }
    else if (i < mheight + 2)
    {
      std::copy(pixelmat[i - 2], pixelmat[i - 2] + mwidth, imgWithPadding[i] + 2);
      imgWithPadding[i][0] = pixelmat[i - 2][1];
      imgWithPadding[i][1] = pixelmat[i - 2][0];
      imgWithPadding[i][mwidth + 2] = pixelmat[i - 2][mwidth - 1];
      imgWithPadding[i][mwidth + 3] = pixelmat[i - 2][mwidth - 2];
    }
    else
    {
      std::copy(pixelmat[2 * mheight + 1 - i], pixelmat[2 * mheight + 1 - i] + mwidth, imgWi
    }
  }
  // deal with corner (0,0)
  imgWithPadding[0][1] = imgWithPadding[0][2];
  imgWithPadding[1][0] = imgWithPadding[2][0];
  imgWithPadding[0][0] = (imgWithPadding[0][1] + imgWithPadding[1][0]) >> 1;
  imgWithPadding[1][1] = (imgWithPadding[1][2] + imgWithPadding[2][1]) >> 1;

  // deal with corner (0,1)
  imgWithPadding[0][mwidth + 2] = imgWithPadding[0][mwidth + 1];
  imgWithPadding[1][mwidth + 3] = imgWithPadding[2][mwidth + 3];
  imgWithPadding[0][mwidth + 3] = (imgWithPadding[0][mwidth + 2] + imgWithPadding[1][mwidth
  imgWithPadding[1][mwidth + 2] = (imgWithPadding[1][mwidth + 1] + imgWithPadding[2][mwidth

  // deal with corner (1,0)
  imgWithPadding[mheight + 2][0] = imgWithPadding[mheight + 1][0];
  imgWithPadding[mheight + 3][1] = imgWithPadding[mheight + 3][2];
  imgWithPadding[mheight + 3][0] = (imgWithPadding[mheight + 2][0] + imgWithPadding[mheight
  imgWithPadding[mheight + 2][1] = (imgWithPadding[mheight + 1][1] + imgWithPadding[mheight

  // deal with corner (1,1)
  imgWithPadding[mheight + 2][mwidth + 3] = imgWithPadding[mheight + 1][mwidth + 3];
  imgWithPadding[mheight + 3][mwidth + 2] = imgWithPadding[mheight + 3][mwidth + 1];
  imgWithPadding[mheight + 2][mwidth + 2] = (imgWithPadding[mheight + 1][mwidth + 2] + imgWi
  imgWithPadding[mheight + 3][mwidth + 3] = (imgWithPadding[mheight + 2][mwidth + 3] + imgWi

  return imgWithPadding;
}

std::complex<double> * *dftmat(const int length, const bool inverse = false)
{
  std::complex<double>** mat = new std::complex<double> * [length];
  for (int i = 0; i < length; i++)
  {
    mat[i] = new std::complex<double>[length];
    for (int j = 0; j < length; j++)
    {
      mat[i][j] = std::exp((inverse ? 1. : -1.) * std::complex<double>(0, 2 * _Pi * i * j /
    }
  }
  return mat;
}

float** haarmat(const int length)
{
  float** mat = new float* [length];
  for (int i = 0; i < length; i++)
  {
    mat[i] = new float[length];
    int p = 0, q = 0;
    while (i >> (p + 1))
    {
      p++;
    }
    q = 1 + (i - (1 << p));
    for (int j = 0; j < length; j++)
    {
      if (i == 0)
      {
        mat[i][j] = 1 / sqrt(float(length));
      }
      else
      {
        if ((j << p) >= length * (q - 1) && (j << p) < length * q)
        {
          if ((j << p) < length * (q - 1 / 2.))
          {
            mat[i][j] = pow(2, p / 2.) / sqrt(float(length));
          }
          else
          {
            mat[i][j] = -pow(2, p / 2.) / sqrt(float(length));
          }
        }
        else
        {
          mat[i][j] = 0;
        }
      }
    }
  }
  return mat;
}

template<typename T = std::complex<double>>
T * *matprod(T * *mat1, int x1, int y1, T * *mat2, int x2, int y2, const bool transpose = fa
{

  if (y1 != y2)
  {
    exit(1);
  }
  else
  {
    T** ret = new T * [x1];
    for (int i = 0; i < x1; i++)
    {
      ret[i] = new T[y2];
      for (int j = 0; j < y1; j++)
      {
        ret[i][j] = 0;
        for (int k = 0; k < x2; k++)
        {
          if (transpose)
          {
            ret[i][j] += mat1[i][k] * mat2[j][k];
          }
          else
          {
            ret[i][j] += mat1[i][k] * mat2[k][j];
          }
        }
      }
    }
    return ret;
  }
}

template<typename T>
T clip3(const T & min, const T & max, const T & t)
{
  return std::min(std::max(t, min), max);
}

template<typename T>
std::complex<double>** dft2(T** pixelmat, int mheight, int mwidth)
```

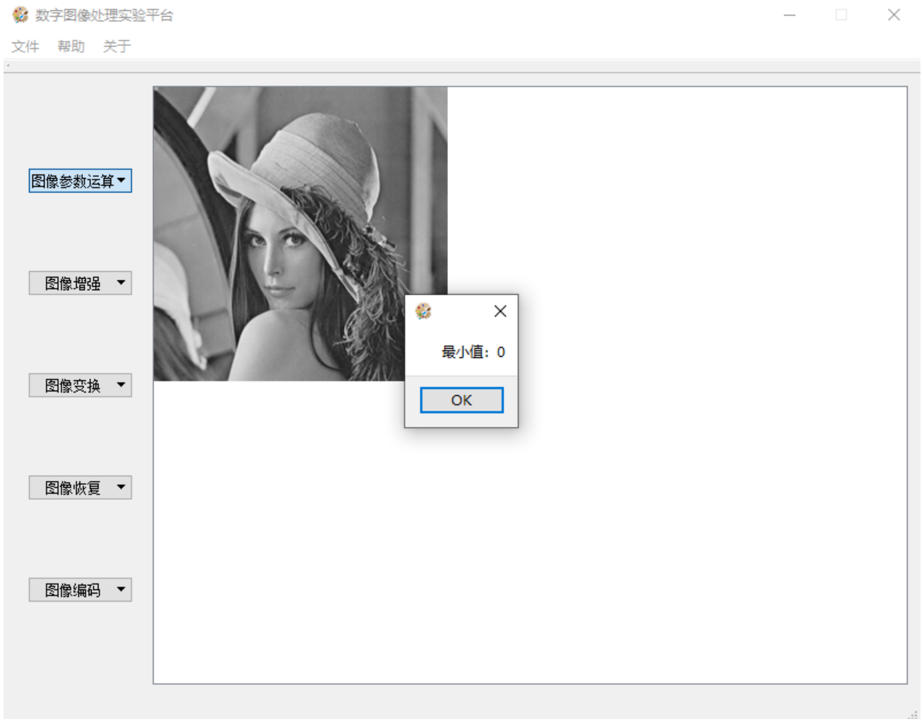
```
{
  std::complex<double>** freq;
  std::complex<double>** pixelComp = new std::complex<double> * [mheight];
  for (int i = 0; i < mheight; i++)
  {
    pixelComp[i] = new std::complex<double>[mwidth];
    for (int j = 0; j < mwidth; j++)
    {
      pixelComp[i][j] = pixelmat[i][j];
    }
  }
  // row
  freq = matprod(dftmat(mheight), mheight, mheight, pixelComp, mheight, mwidth);
  // column
  freq = matprod(freq, mheight, mwidth, dftmat(mwidth), mwidth, mwidth);
  return freq;
}

std::complex<double>** idft2(std::complex<double> * *pixelComp, int mheight, int mwidth)
{
  // column
  pixelComp = matprod(pixelComp, mheight, mwidth, dftmat(mwidth, true), mwidth, mwidth);
  // row
  pixelComp = matprod(dftmat(mheight, true), mheight, mheight, pixelComp, mheight, mwidth);
  return pixelComp;
}

float** haar2(int** pixelmat, int mheight, int mwidth)
{
  float** freq;
  float** pixelf = new float* [mheight];
  for (int i = 0; i < mheight; i++)
  {
    pixelf[i] = new float[mwidth];
    for (int j = 0; j < mwidth; j++)
    {
      pixelf[i][j] = pixelmat[i][j];
    }
  }
  // row
  freq = matprod(haarmat(mheight), mheight, mheight, pixelf, mheight, mwidth);
  // column
  freq = matprod(freq, mheight, mwidth, haarmat(mwidth), mwidth, mwidth, true);
  return freq;
}
```

{# -1 }

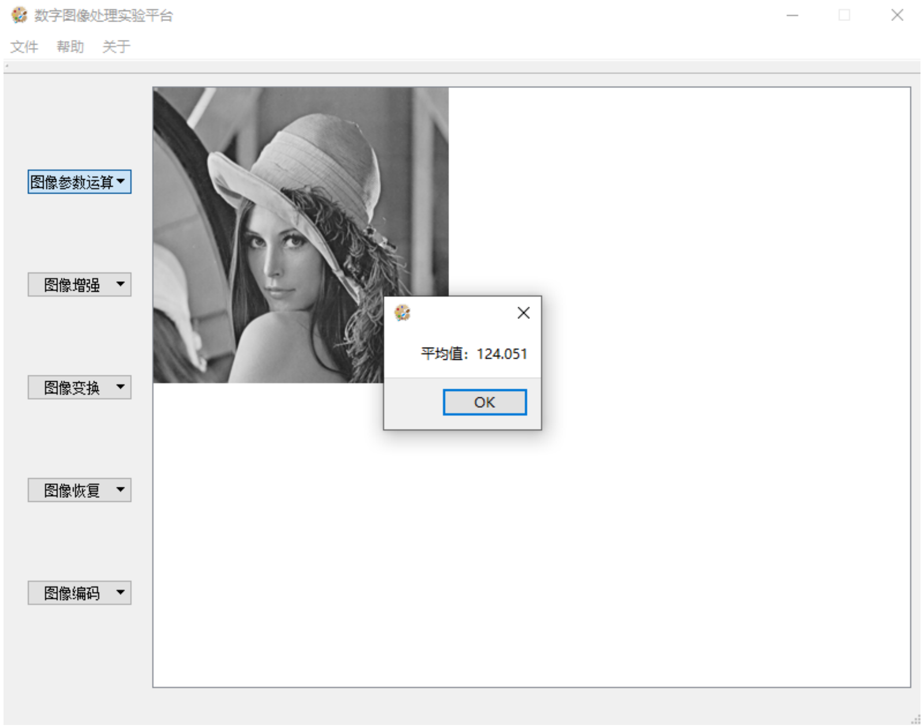
```
//
int maxvalue(int** pixelmat, int mheight, int mwidth)
{
  int max = 0;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      if (pixelmat[i][j] > max)
      {
        max = pixelmat[i][j];
      }
    }
  }
  return max;
}

//
int minvalue(int** pixelmat, int mheight, int mwidth)
{
  int min = 0;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      if (pixelmat[i][j] < min)
      {
        min = pixelmat[i][j];
      }
    }
  }
  return min;
}

//
float avgvalue(int** pixelmat, int mheight, int mwidth)
{
  float avg = 0;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      avg += pixelmat[i][j];
    }
  }
  return avg / (mheight * mwidth);
}

//
float varvalue(int** pixelmat, int mheight, int mwidth)
{
  float v2a = 0, va = 0;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      v2a += pixelmat[i][j] * pixelmat[i][j];
      va += pixelmat[i][j];
    }
  }
  v2a /= mheight * mwidth;
  va /= mheight * mwidth;
  return v2a - va * va;
}

//   ,    256 1
int* histogram(int** pixelmat, int mheight, int mwidth)
{
  // :         ;
  int* hist = new int[256]();
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      hist[pixelmat[i][j]]++;
    }
  }
  return hist;
}
```


数字图像处理实验平台
文件
帮助
关于
图像参数运算
图像增强
图像变换
图像恢复
图像编码
OK

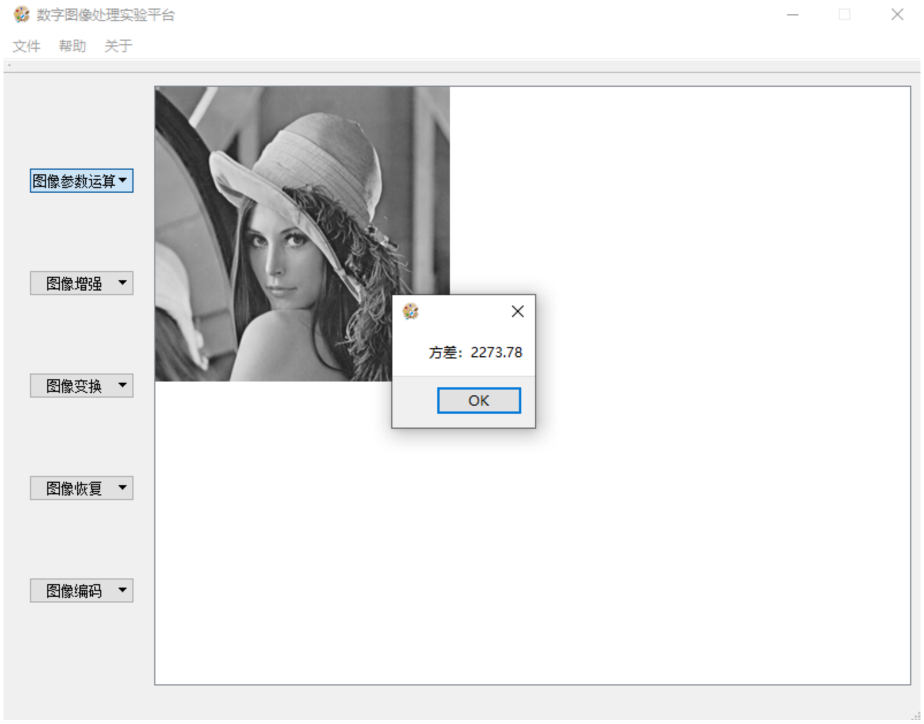
最大值：245


数字图像处理实验平台
文件
帮助
关于
图像参数运算
图像增强
图像变换
图像恢复
图像编码
OK

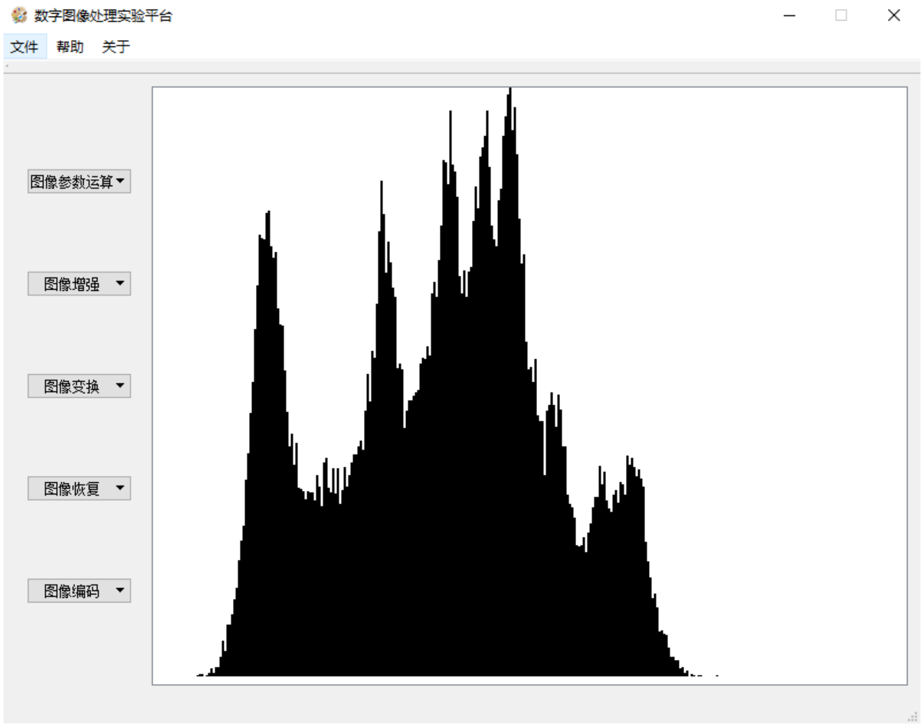
最小值：0



平均值：124.051



方差：2273.78



log

```
//walsh ,
int** walsh(int** pixelmat, int mheight, int mwidth)
{
#if WALSH ...
#else
  //TODO walsh => this is minusPic
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = 255 - pixelmat[i][j];
    }
  }
  return pixelmat;
#endif
}
```

pixelmat[i][j] = 255 - pixelmat[i][j]

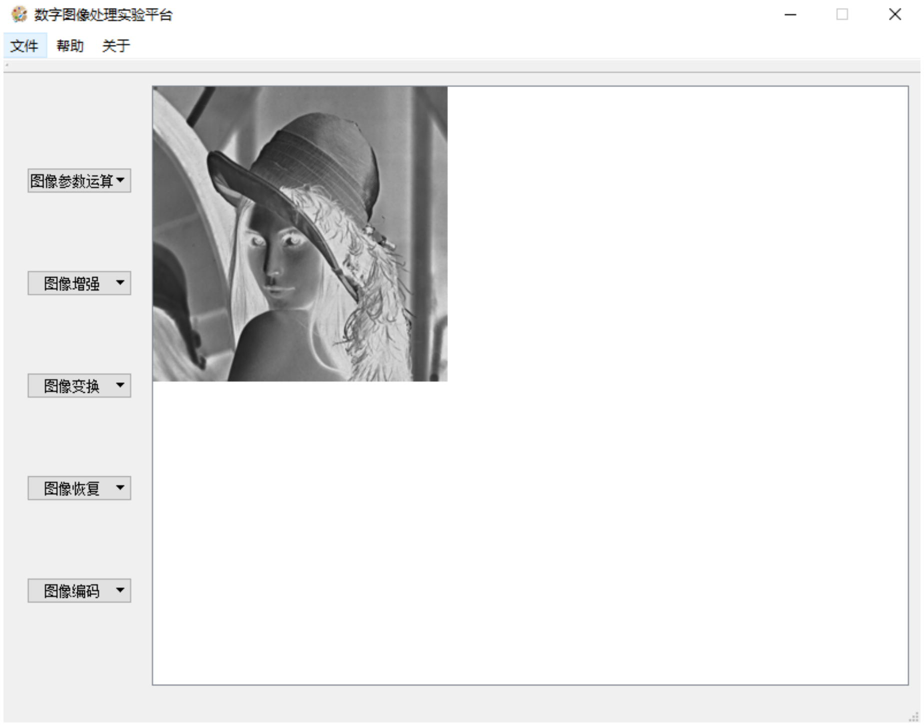


Figure 1: minus

log

```
//DCT ,
int** DCT(int** pixelmat, int mheight, int mwidth)
{
#if DCTTRANS ...
#else
  //TODO DCT => this is log
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = 31.875 * log2(1.0 + pixelmat[i][j]);
    }
  }
  return pixelmat;
#endif
}
```

pixelmat[i][j] = 31.875 * log2(1.0 + pixelmat[i][j]) scale 31.875=255/log2(1+255)

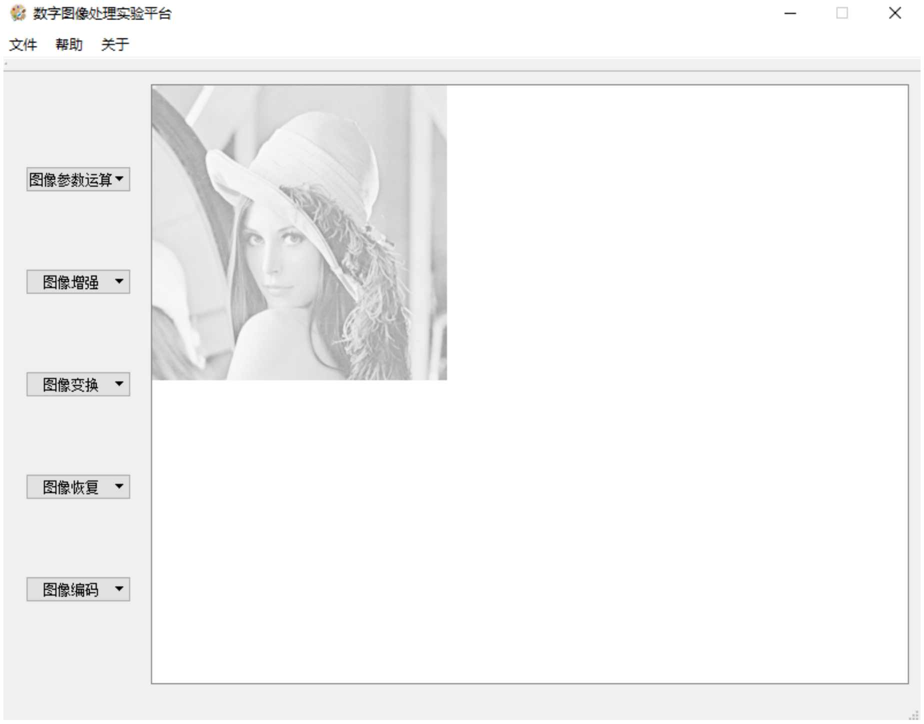


Figure 2: log

```
//   ,
int** histogramequ(int** pixelmat, int mheight, int mwidth)
{
  int* pdf = histogram(pixelmat, mheight, mwidth);
  int cdf[1 << 8], sum = 0;
  for (int i = 0; i < (1 << 8); i++)
  {
    sum += pdf[i];
    cdf[i] = sum;
  }

  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = cdf[pixelmat[i][j]] * 255 / sum;
    }
  }

  delete[] pdf;

  return pixelmat;
}
```

histogram() pdf cdf pixelmat[i][j] = cdf[pixelmat[i][j]] * 255 / sum

```
//  ,
int** graystretch(int** pixelmat, int mheight, int mwidth)
{
  int x1, y1, x2, y2;
  x1 = 100, y1 = 50, x2 = 200, y2 = 220;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      if (pixelmat[i][j] < x1)
      {
        pixelmat[i][j] = pixelmat[i][j] * y1 / x1;
      }
      else if (pixelmat[i][j] < x2)
      {
```

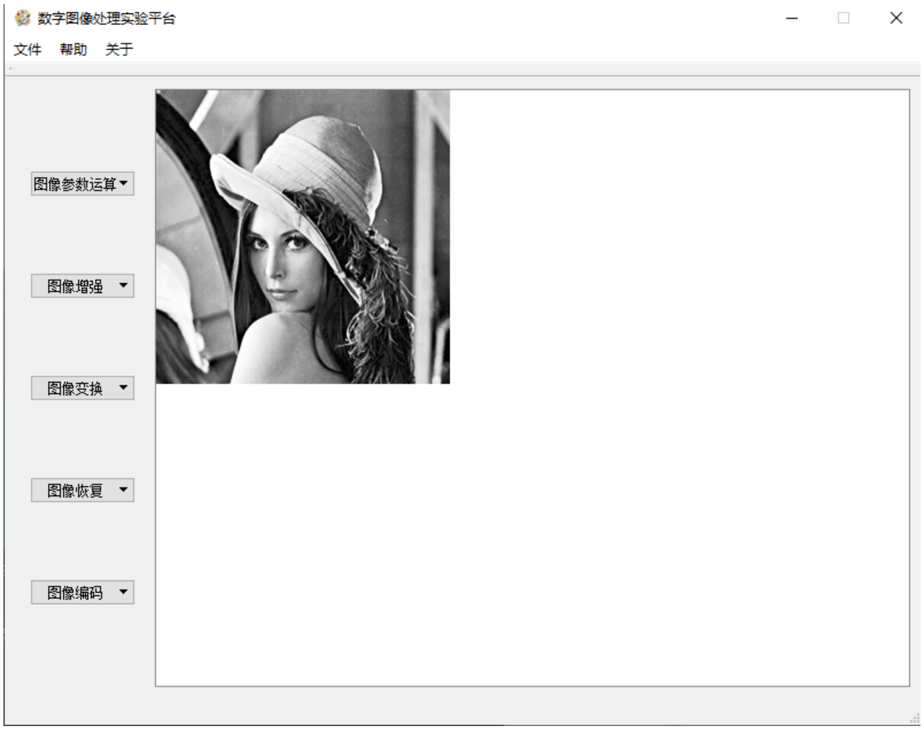


Figure 3: histEqu

```
        pixelmat[i][j] = (pixelmat[i][j] - x1) * (y2 - y1) / (x2 - x1) + y1;
      }
      else
      {
        pixelmat[i][j] = (pixelmat[i][j] - x2) * (255 - y2) / (255 - x2) + y2;
      }
    }
  }
  return pixelmat;
}
```

(x1,y1) (x2,y2)

Gaussian

```
//   ,     ;
int** randomnoise(int** pixelmat, int mheight, int mwidth)
{
  std::default_random_engine generator;
  std::normal_distribution<double> distribution(0.0, 5.0);
  auto noise = std::bind(distribution, generator);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] += noise();
    }
  }
  return pixelmat;
}
```

0 5

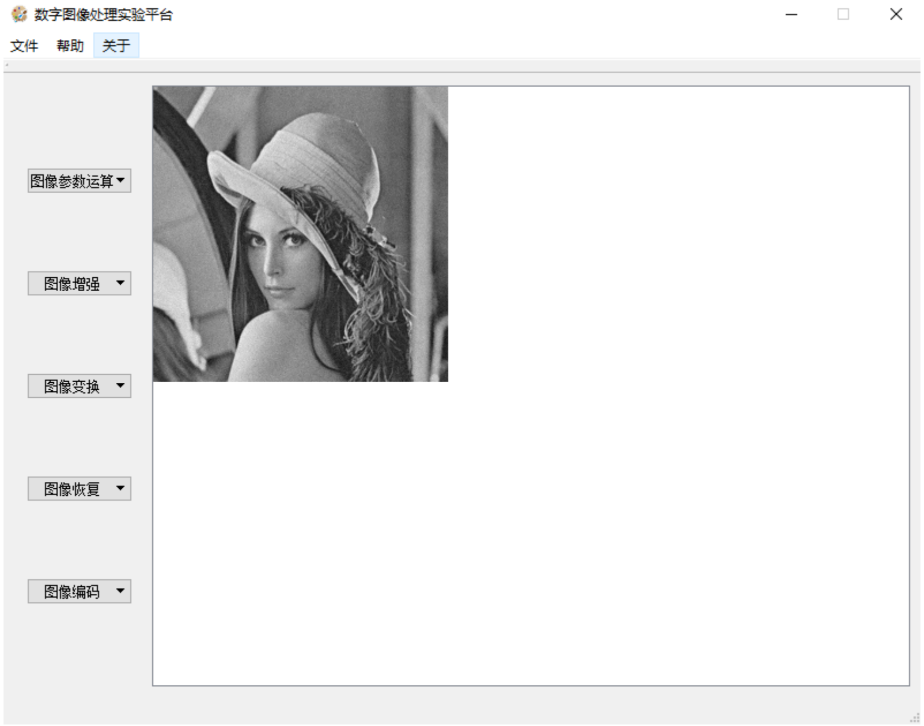


Figure 4: gaussian

```
//   ,
int** impulsenoise(int** pixelmat, int mheight, int mwidth)
{
  std::default_random_engine generator;
  std::uniform_real_distribution<double> distribution(0, 1);
  auto noise = std::bind(distribution, generator);
  const double prob = 0.05;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      double n = noise();
      if (n < prob)
      {
        if (n < prob / 2)
        {
          pixelmat[i][j] = 0;
        }
        else
        {
          pixelmat[i][j] = 1 << 8 - 1;
        }
      }
    }
  }
  return pixelmat;
}
```

[0,1]   n=noise() n>1-prob   n<prob/2

```
//  ,
int** medianfit(int** pixelmat, int mheight, int mwidth)
{
  int** imgWithPadding = paddingImg2(pixelmat, mheight, mwidth);
```

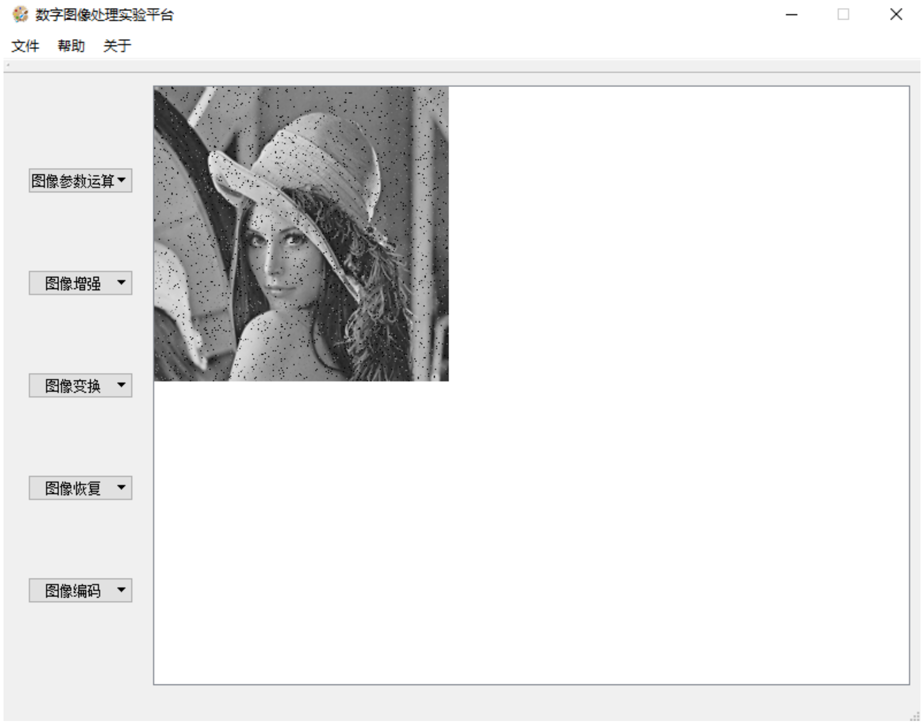
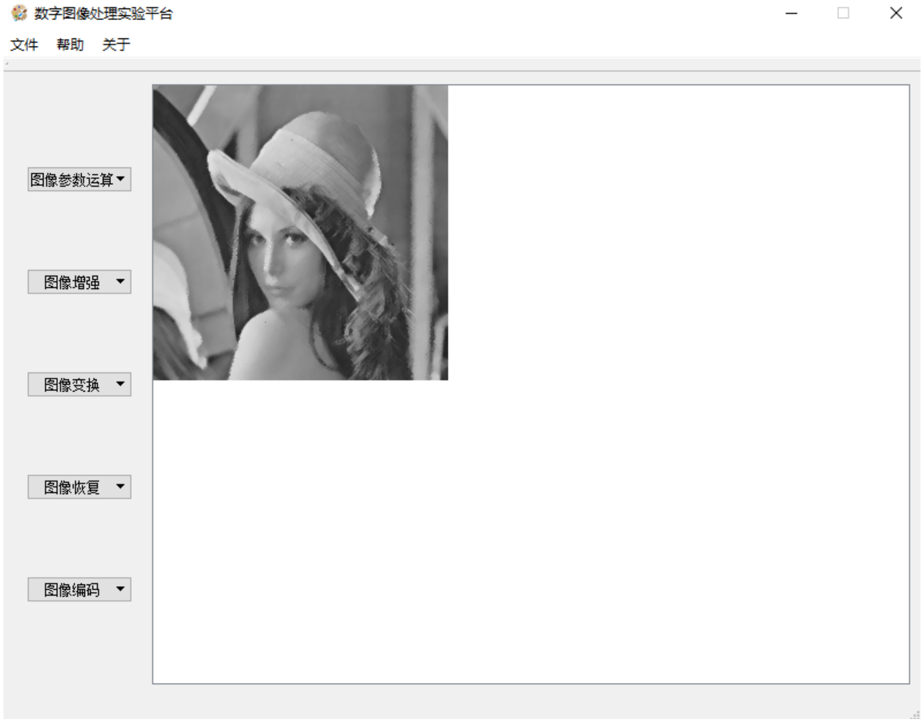
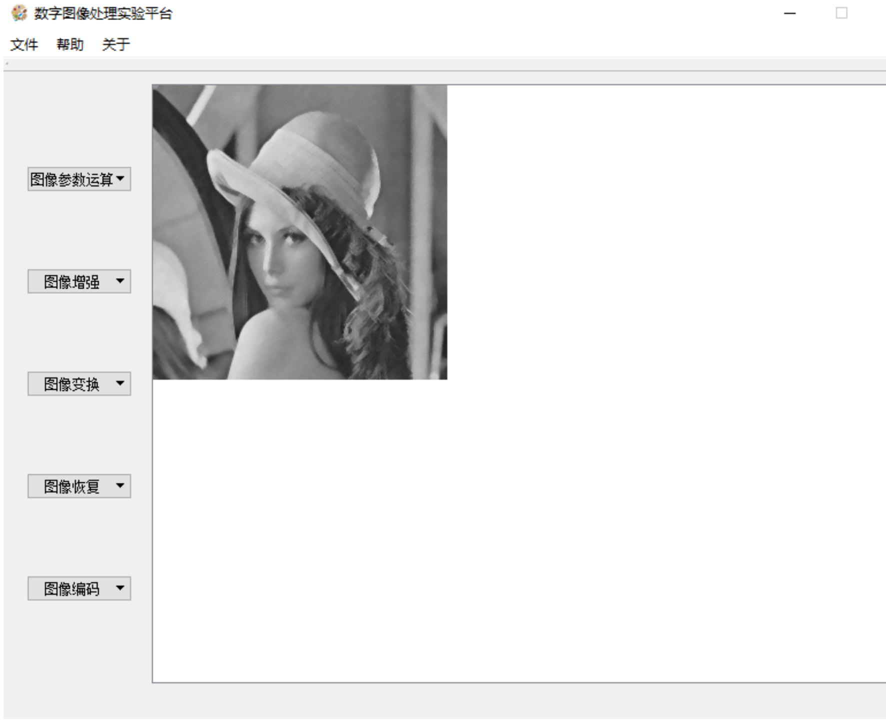


Figure 5: salt

```
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int pattern[9];
      pattern[0] = imgWithPadding[i][j];
      pattern[1] = imgWithPadding[i][j + 4];
      pattern[2] = imgWithPadding[i + 1][j + 1];
      pattern[3] = imgWithPadding[i + 1][j + 3];
      pattern[4] = imgWithPadding[i + 2][j + 2];
      pattern[5] = imgWithPadding[i + 3][j + 1];
      pattern[6] = imgWithPadding[i + 3][j + 3];
      pattern[7] = imgWithPadding[i + 4][j];
      pattern[8] = imgWithPadding[i + 4][j + 4];

      std::sort(&pattern[0], &pattern[8]);
      pixelmat[i][j] = pattern[4];
    }
  }
  for (int i = 0; i < mheight + 4; i++)
  {
    delete[] imgWithPadding[i];
  }
  delete[] imgWithPadding;
  return pixelmat;
}
```

padding 5x5 X stl sort

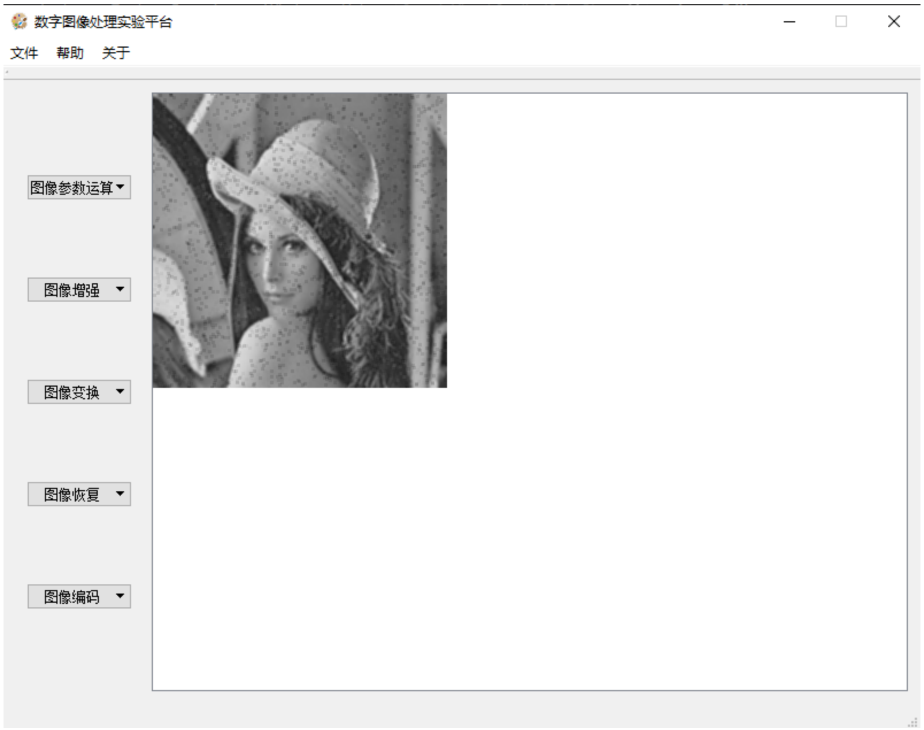


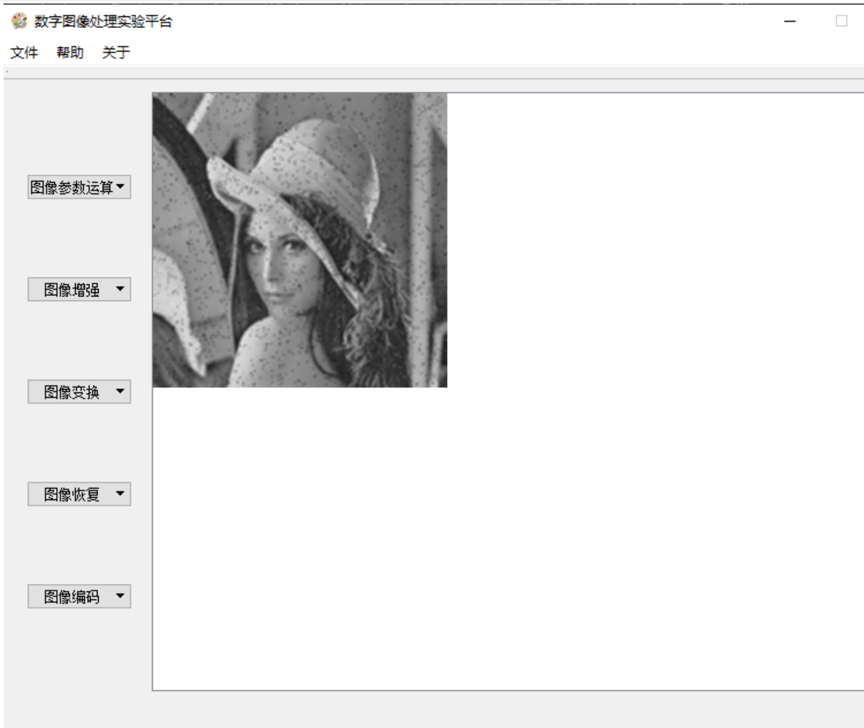


Gaussian {#gaussian }

```
//  ,
int** averagefit(int** pixelmat, int mheight, int mwidth)
{
  const int filter[3][3] = { {1,1,1},{1,1,1},{1,1,1} };
  int** imgWithPadding = paddingImg2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int sum = 0;
      for (int r = 0; r < 3; r++)
      {
        for (int c = 0; c < 3; c++)
        {
          sum += imgWithPadding[i + r + 1][j + c + 1] * filter[r][c];
        }
      }
      pixelmat[i][j] = sum / 9;
    }
  }
  for (int i = 0; i < mheight + 4; i++)
  {
    delete[] imgWithPadding[i];
  }
  delete[] imgWithPadding;
  return pixelmat;
}
```

padding 3x3

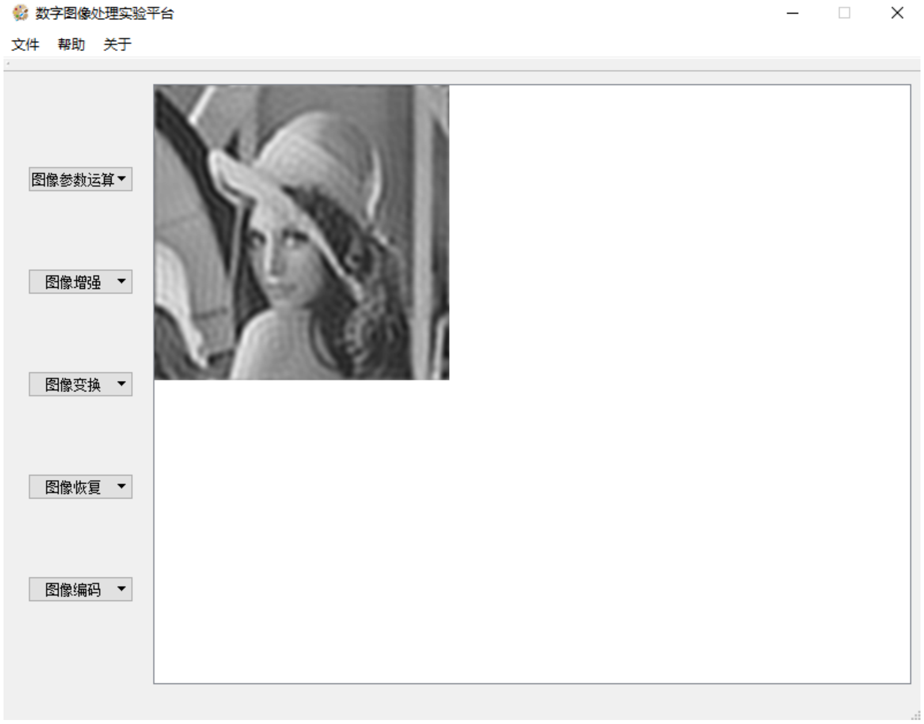


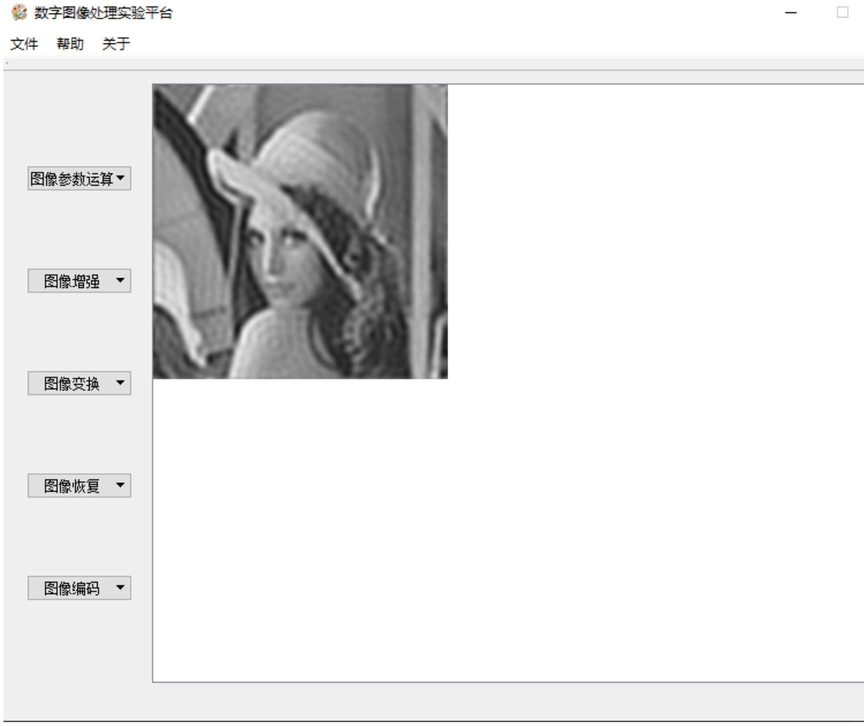


Gaussian {#gaussian -1 }

```cpp
//   ,
int** lowpassfit(int** pixelmat, int mheight, int mwidth)
{
  std::complex<double>** freq = dft2(pixelmat, mheight, mwidth);
  constexpr int D = 30;
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int absx, absy;
      absx = min(abs(i), abs(i - mheight));
      absy = min(abs(j), abs(j - mwidth));
      if (absx * absx + absy * absy > D * D)
      {
        freq[i][j] = 0;
      }
    }
  }
  std::complex<double>** pixelComp = idft2(freq, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = clip3(0, 255, int(abs(pixelComp[i][j]) / (mheight * mwidth)));
    }
  }

  return pixelmat;
}
```

dft





Gaussian {#gaussian -2 }

Sobel

```
//sobel ,
int** sobel(int** pixelmat, int mheight, int mwidth)
{
  const int sobel[3][3] = { {1,2,1},{0,0,0},{-1,-2,-1} };
  int** imgWithPadding = paddingImg2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int sum = 0;
      for (int r = 0; r < 3; r++)
      {
        for (int c = 0; c < 3; c++)
        {
          sum += imgWithPadding[i + r + 1][j + c + 1] * sobel[r][c];
        }
      }
      pixelmat[i][j] = sum;
    }
  }
  for (int i = 0; i < mheight + 4; i++)
  {
    delete[] imgWithPadding[i];
  }
  delete[] imgWithPadding;

  imgWithPadding = paddingImg2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int sum = 0;
      for (int r = 0; r < 3; r++)
      {
        for (int c = 0; c < 3; c++)
        {
          sum += imgWithPadding[i + r + 1][j + c + 1] * sobel[c][r];
        }
      }
      pixelmat[i][j] = abs(sum);
    }
  }
  for (int i = 0; i < mheight + 4; i++)
  {
    delete[] imgWithPadding[i];
  }
  delete[] imgWithPadding;
  return pixelmat;
}
```

padding Sobel

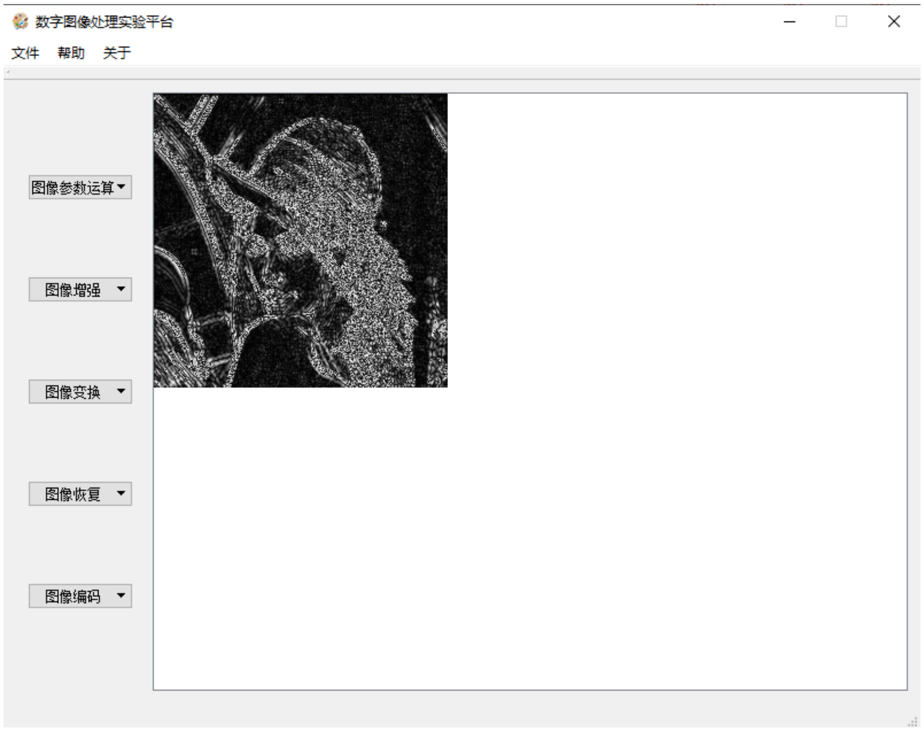


Figure 6: sobel

Laplace

```
//laplace ,
int** laplace(int** pixelmat, int mheight, int mwidth)
{
  const int laplacian[3][3] = { {0,-1,0},{-1,4,-1},{0,-1,0} };
  int** imgWithPadding = paddingImg2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      int sum = 0;
      for (int r = 0; r < 3; r++)
      {
        for (int c = 0; c < 3; c++)
        {
          sum += imgWithPadding[i + r + 1][j + c + 1] * laplacian[c][r];
        }
      }
      pixelmat[i][j] = abs(sum);
    }
  }
  for (int i = 0; i < mheight + 4; i++)
  {
    delete[] imgWithPadding[i];
  }
  delete[] imgWithPadding;
  return pixelmat;
}
```

padding laplace

```
//    ,
int** highpassfit(int** pixelmat, int mheight, int mwidth)
{
  std::complex<double>** freq = dft2(pixelmat, mheight, mwidth);
  constexpr int D = 30;
  for (int i = 0; i < mheight; i++)
```

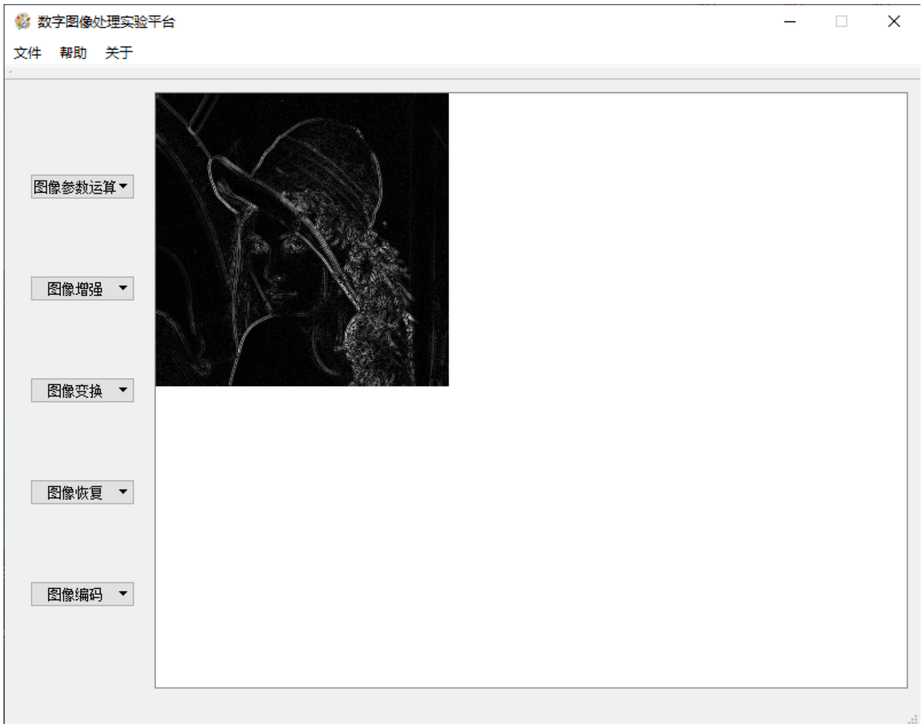


Figure 7: laplace

```
  {
    for (int j = 0; j < mwidth; j++)
    {
      int absx, absy;
      absx = min(abs(i), abs(i - mheight));
      absy = min(abs(j), abs(j - mwidth));
      if (absx * absx + absy * absy < D * D)
      {
        freq[i][j] = 0;
      }
    }
  }
  std::complex<double>** pixelComp = idft2(freq, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = clip3(0, 255, int(abs(pixelComp[i][j]) / (mheight * mwidth)));
    }
  }

  return pixelmat;
}
```

dft

(20)

```
//  ,    ( )
int** rotation(int** framemat, int** pixelmat, int mheight, int mwidth)
{
  const float theta = 20. / 180 * _Pi;
  auto rotate = [](float& x, float& y, const float& theta) { float tmpx = x * cos(theta) + y

  float lt[2] = { -mwidth / 2,mheight / 2 };
  float rt[2] = { mwidth / 2 - 1,mheight / 2 };
  float lb[2] = { -mwidth / 2,1 - mheight / 2 };
  float rb[2] = { mwidth / 2 - 1,1 - mheight / 2 };
```

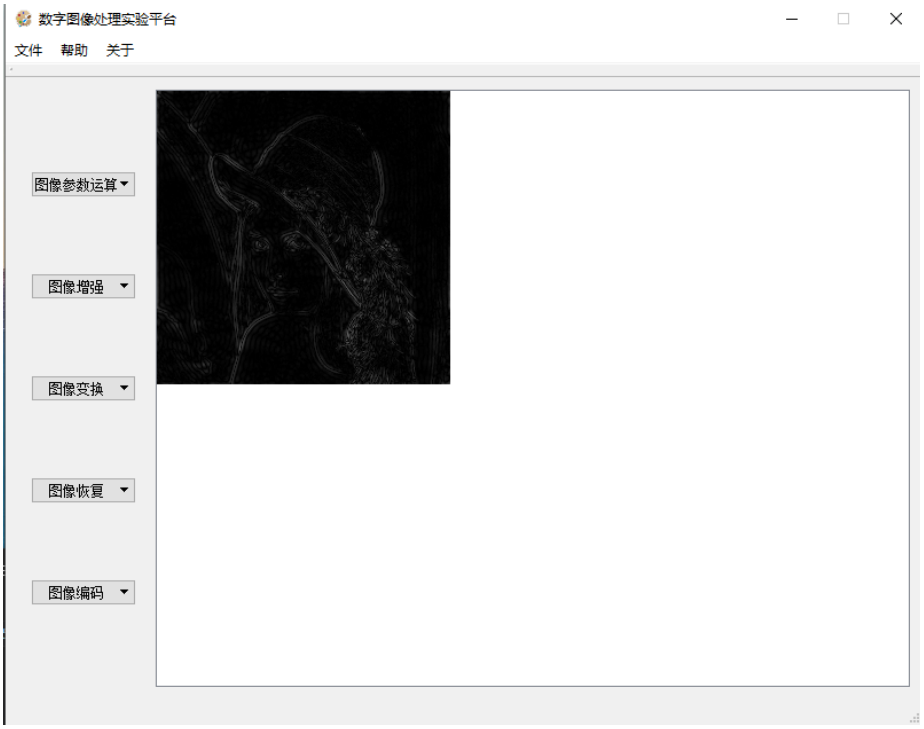


Figure 8: highPass

```
 rotate(lt[0], lt[1], theta);
 rotate(rt[0], rt[1], theta);
 rotate(lb[0], lb[1], theta);
 rotate(rb[0], rb[1], theta);

 const float lm = min({ lt[0],rt[0],lb[0], rb[0] });
 const float rm = max({ lt[0],rt[0],lb[0], rb[0] });
 const float tm = max({ lt[1],rt[1],lb[1], rb[1] });
 const float bm = min({ lt[1],rt[1],lb[1], rb[1] });

 for (int y = 0; y <= tm - bm; y++)
 {
  for (int x = 0; x <= rm - lm; x++)
  {
   float tmpx = x + lm, tmpy = tm - y;
   rotate(tmpx, tmpy, -theta);

   float orix = tmpx + mwidth / 2, oriy = mheight / 2 - tmpy;
   if (orix >= 0 && orix < mwidth - 1 && oriy >= 0 && oriy < mheight - 1)
   {
    framemat[y][x]
     = pixelmat[int(oriy)][int(orix)] * (1 + floor(oriy) - oriy) * (1 + floor(orix) - o
     + pixelmat[int(oriy)][int(orix) + 1] * (1 + floor(oriy) - oriy) * (orix - floor(or
     + pixelmat[int(oriy) + 1][int(orix)] * (oriy - floor(oriy)) * (1 + floor(orix) - o
     + pixelmat[int(oriy) + 1][int(orix) + 1] * (orix - floor(orix)) * (oriy - floor(or
   }
  }
 }

 return framemat;
}
```

1.
2. [lm,rm]x[tm,bm]
3.

-

DFT

```
//DFT ,      ,    0~255
int** DFT(int** pixelmat, int mheight, int mwidth)
{
  complex<double>** freq = dft2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = clip3(0., 255., 20 * log(std::abs(freq[i][j]) + 1));
    }
  }
  return pixelmat;
}
```

wrapper DFT DFT ->

DCT

```
//DCT ,
int** DCT(int** pixelmat, int mheight, int mwidth)
{
#if DCTTRANS
  float dctmat[8][8];
  for (int i = 0; i < 8; i++)
  {
    for (int j = 0; j < 8; j++)
    {
      if (i == 0)
      {
        dctmat[i][j] = 1 / sqrt(8);
      }
      else
      {
        dctmat[i][j] = cos(_Pi * i * (2 * j + 1) / 16) / 2;
      }
    }
```

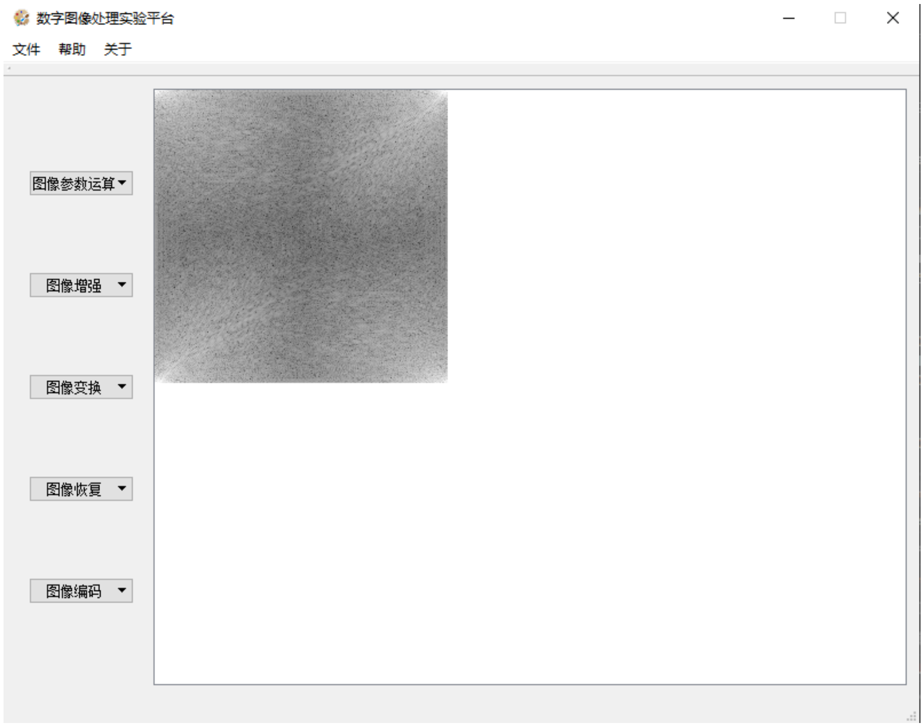


Figure 9: dft

```
  }
  for (int ydiv = 0; ydiv < mheight / 8; ydiv++)
  {
    for (int xdiv = 0; xdiv < mwidth / 8; xdiv++)
    {
      // for every 8x8 block
      float block[8][8] = { 0 };
      // col trans
      for (int row = 0; row < 8; row++)
      {
        for (int col = 0; col < 8; col++)
        {
          for (int k = 0; k < 8; k++)
          {
            block[row][col] += dctmat[row][k] * pixelmat[ydiv * 8 + k][xdiv * 8 + col];
          }
        }
      }
      // row trans
      for (int row = 0; row < 8; row++)
      {
        for (int col = 0; col < 8; col++)
        {
          float sum = 0;
          for (int k = 0; k < 8; k++)
          {
            sum += block[row][k] * dctmat[col][k];
          }
          pixelmat[ydiv * 8 + row][xdiv * 8 + col] = abs(sum);
        }
      }
    }
  }
  return pixelmat;
#else ...
#endif
}
```

1. DCT
2.

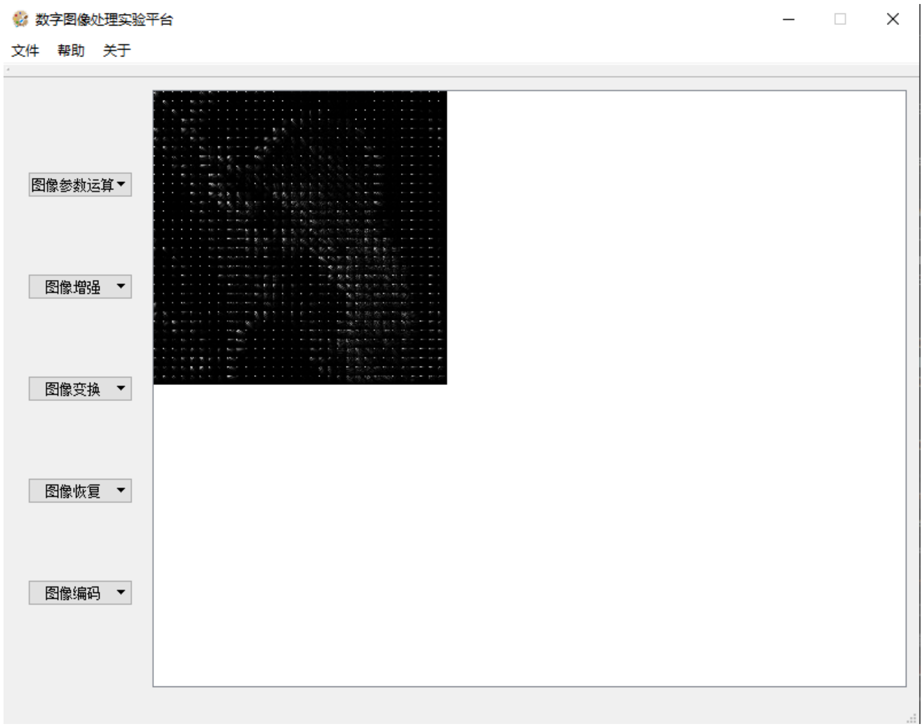


Figure 10: dct

Walsh

```
//walsh ,
int** walsh(int** pixelmat, int mheight, int mwidth)
{
#if WALSH
  constexpr int walshmat[8][8] = {
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,-1,-1,-1,-1},
    {1,1,-1,-1,1,1,-1,-1},
    {1,1,-1,-1,-1,-1,1,1},
    {1,-1,1,-1,1,-1,1,-1},
    {1,-1,1,-1,-1,1,-1,1},
    {1,-1,-1,1,1,-1,-1,1},
    {1,-1,-1,1,-1,1,1,-1}
  };
  for (int ydiv = 0; ydiv < mheight / 8; ydiv++)
  {
    for (int xdiv = 0; xdiv < mwidth / 8; xdiv++)
    {
      // for every 8x8 block
      int block[8][8] = { 0 };
      // col trans
      for (int row = 0; row < 8; row++)
      {
        for (int col = 0; col < 8; col++)
        {
          for (int k = 0; k < 8; k++)
          {
            block[row][col] += walshmat[row][k] * pixelmat[ydiv * 8 + k][xdiv * 8 + col];
          }
        }
      }
      // row trans
      for (int row = 0; row < 8; row++)
      {
        for (int col = 0; col < 8; col++)
        {
          pixelmat[ydiv * 8 + row][xdiv * 8 + col] = 0;
          for (int k = 0; k < 8; k++)
          {
            pixelmat[ydiv * 8 + row][xdiv * 8 + col] += block[row][k] * walshmat[col][k];
          }
        }
      }
    }
  }
  // post-processing
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = abs(pixelmat[i][j]) >> 3;
    }
  }
  return pixelmat;
#else ...
#endif
}
```

DCT dct walsh 8

Haar

```
//haar ,
int** haar(int** pixelmat, int mheight, int mwidth)
{
  float** freq = haar2(pixelmat, mheight, mwidth);
  for (int i = 0; i < mheight; i++)
  {
    for (int j = 0; j < mwidth; j++)
    {
      pixelmat[i][j] = clip3(0.f, 255.f, abs(freq[i][j]));
    }
  }
  return pixelmat;
}
```

dft wrapper haar

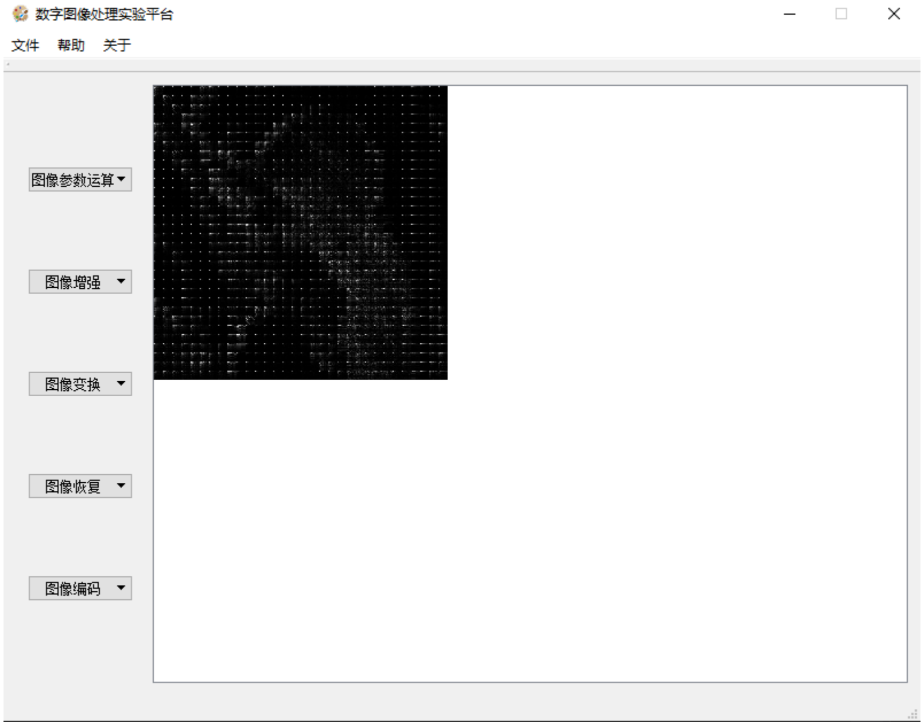


Figure 11: walsh

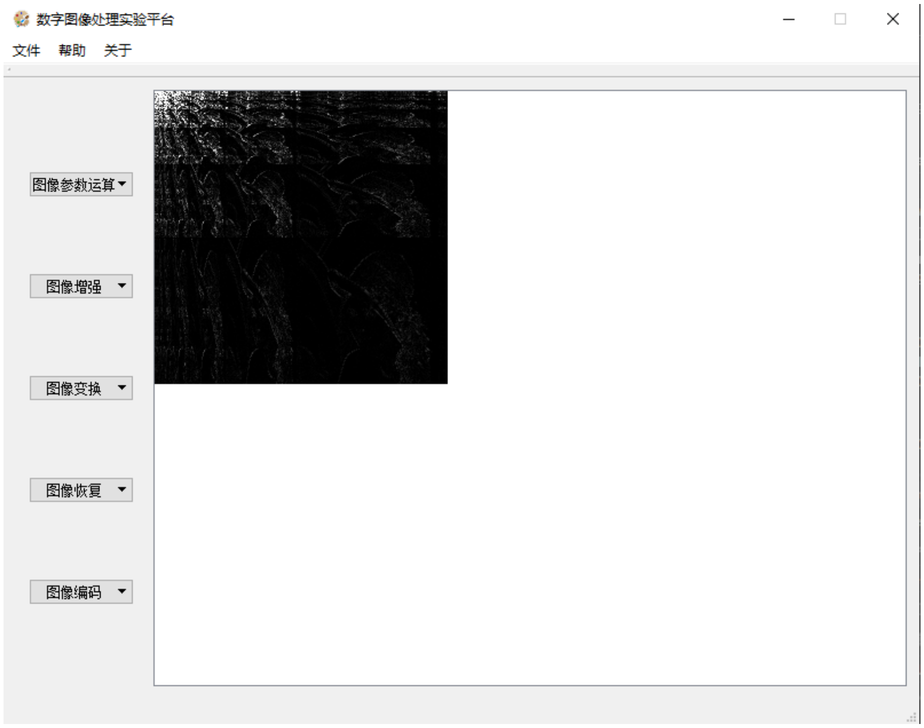


Figure 12: haar

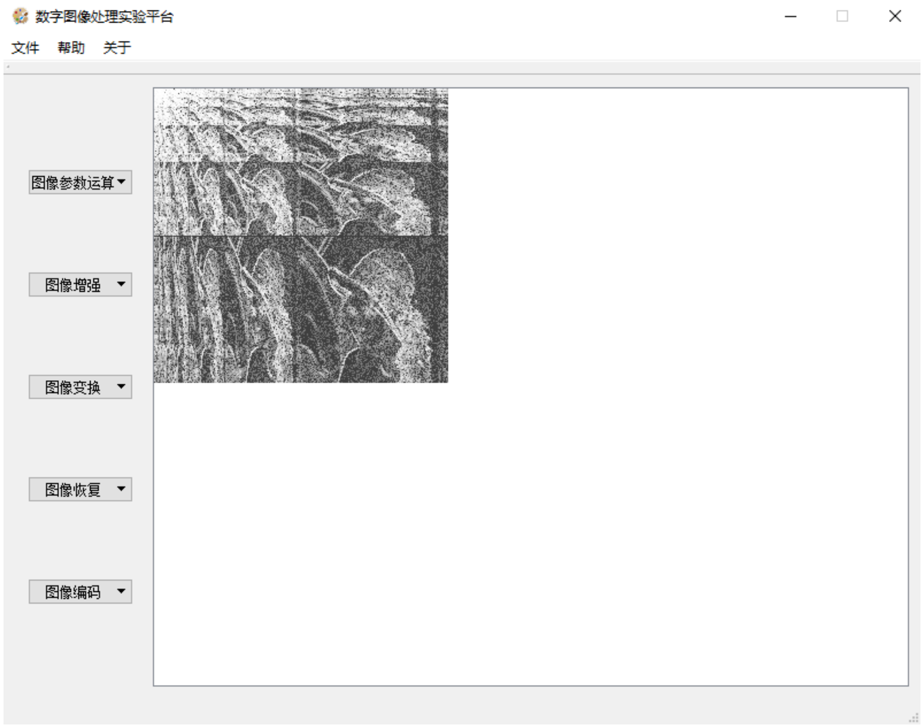


Figure 13: haar+equ

+

( )

```
#define XS 10

// : JPEG
int** jpeg(int** pixelmat, int mheight, int mwidth)
{
  // This is motion blur
  const int xs = XS;
  int** pixMov = new int* [mheight];
  for (int r = 0; r < mheight; r++)
  {
    pixMov[r] = new int[mwidth];
    for (int c = 0; c < mwidth; c++)
    {
      int sum = 0;
      for (int x = 0; x < xs; x++)
      {
        sum += pixelmat[r][uint8_t(c - x)];
      }
      pixMov[r][c] = sum / xs;
    }
  }
  for (int r = 0; r < mheight; r++)
  {
    for (int c = 0; c < mwidth; c++)
    {
      pixelmat[r][c] = pixMov[r][c];
    }
  }
  return pixelmat;
}
```

x

```
//
int** inversefit(int** pixelmat, int mheight, int mwidth)
{
#if INVFIT
  const int xs = XS;
  const double threshold = 0.01;
  complex<double>** freqPel = dft2(pixelmat, mheight, mwidth);
  complex<double>* freqFit = new complex<double>[mwidth];
  for(int i = 0; i < mwidth; i++)
  {
    for(int j = 0; j < xs; j++)
    {
      freqFit[i] += exp( -complex<double>(0, 2. * _Pi / mwidth * i * j)) / double(xs);
    }
```

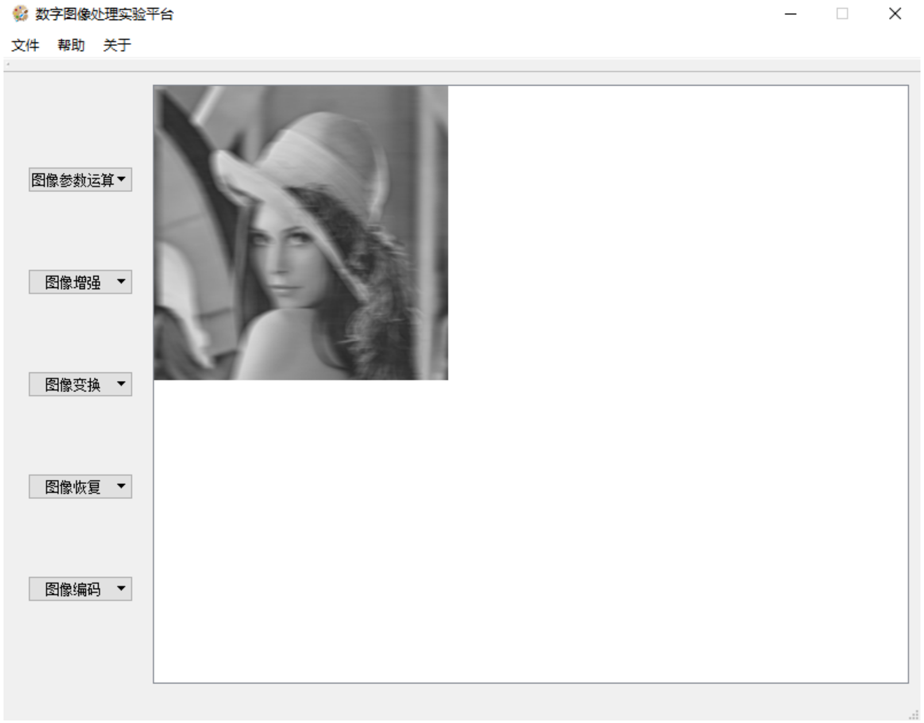


Figure 14: motionBlur

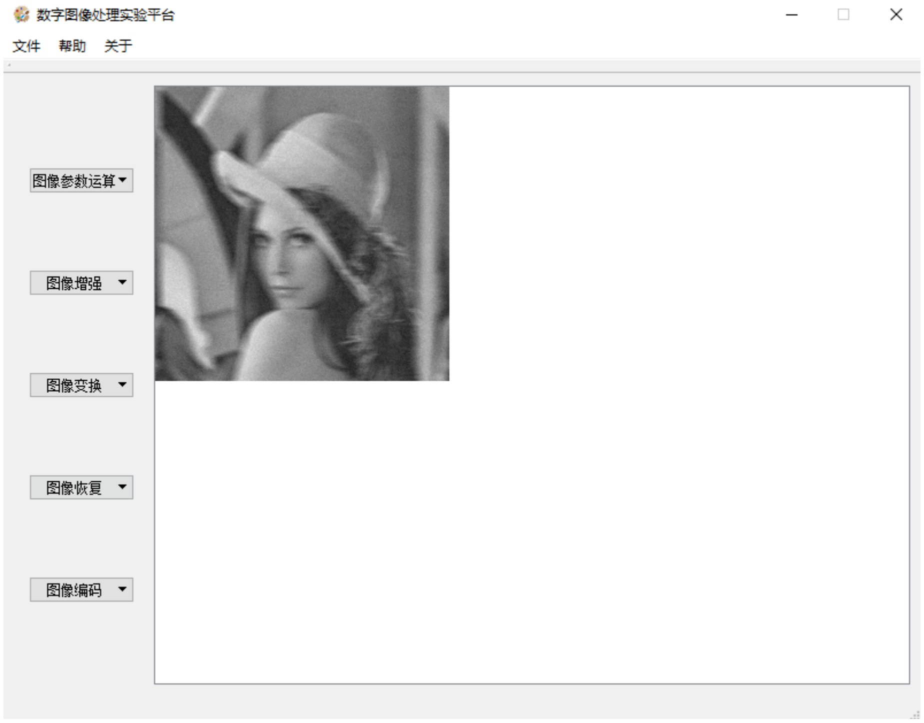


Figure 15: motionBlurG

```
  }
  for(int r = 0; r < mheight; r++)
  {
    for(int c = 0; c < mwidth; c++)
    {
      if(abs(freqFit[c]) > threshold)
      {
        freqPel[r][c] /= freqFit[c];
      }
    }
  }

  complex<double>** pelOri = idft2(freqPel, mheight, mwidth);
  for(int r = 0; r < mheight; r++)
  {
    for(int c = 0; c < mwidth; c++)
    {
      pixelmat[r][c] = clip3(0., 255., abs(pelOri[r][c]) / (mheight * mwidth));
    }
  }
  return pixelmat;
#else ...
#endif
}
```

```
//
int** wienerfit(int** pixelmat, int mheight, int mwidth)
{
  const int xs = XS;
  const double K = 0.01;
  complex<double>** freqPel = dft2(pixelmat, mheight, mwidth);
  complex<double>* freqFit = new complex<double>[mwidth];
```

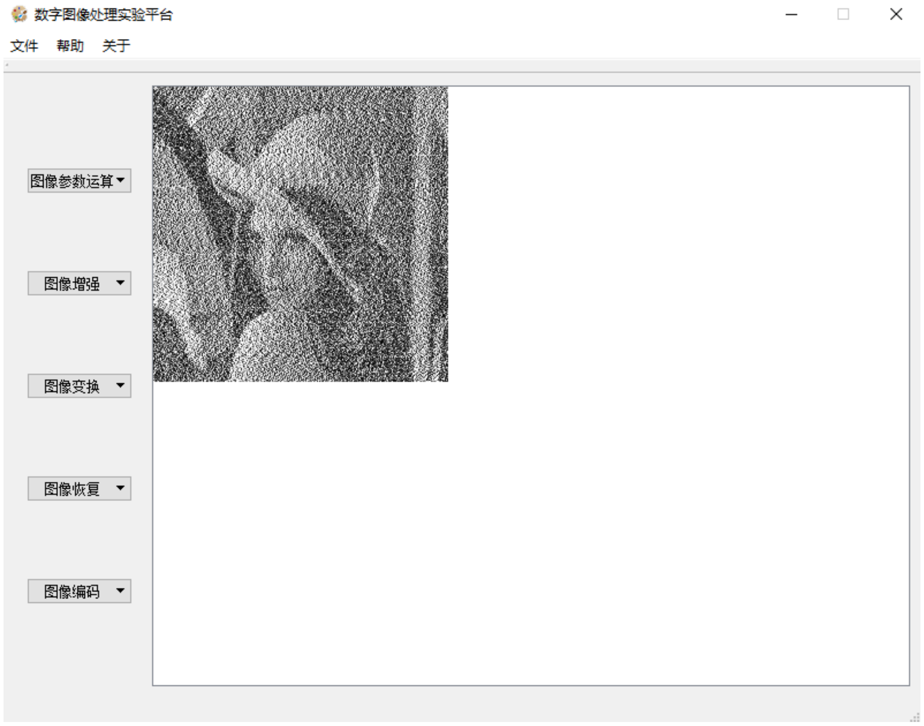


Figure 16: invfitG

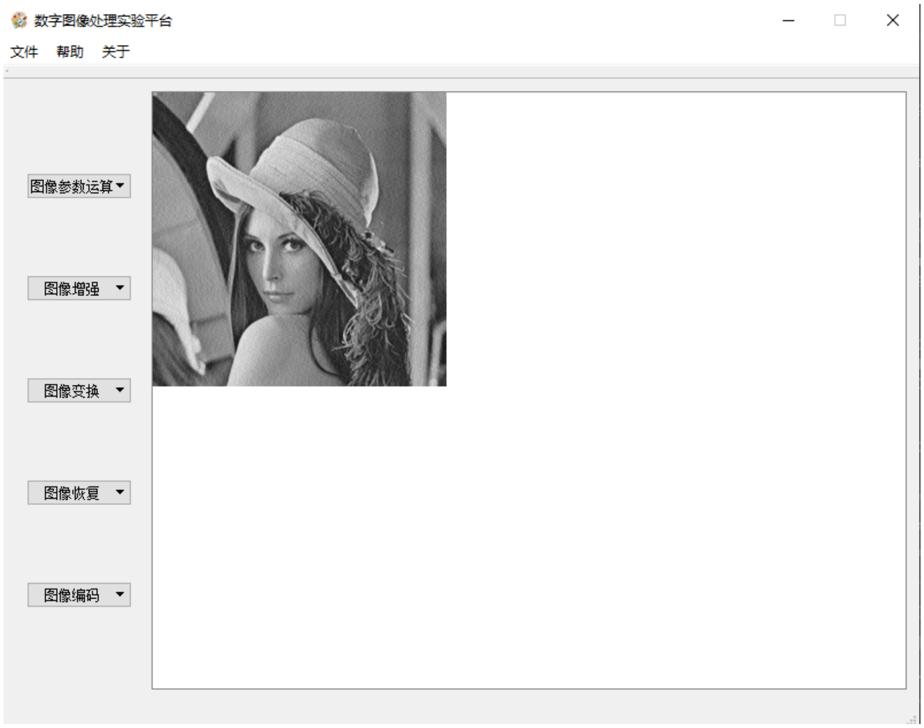


Figure 17: invfit

```
  for (int i = 0; i < mwidth; i++)
  {
    for (int j = 0; j < xs; j++)
    {
      freqFit[i] += exp(-complex<double>(0, 2. * _Pi / mwidth * i * j)) / double(xs);
    }
  }
  for (int r = 0; r < mheight; r++)
  {
    for (int c = 0; c < mwidth; c++)
    {
      freqPel[r][c] *= conj(freqFit[c]) / (pow(abs(freqFit[c]),2)+K);
    }
  }
  complex<double>** pelOri = idft2(freqPel, mheight, mwidth);
  for (int r = 0; r < mheight; r++)
  {
    for (int c = 0; c < mwidth; c++)
    {
      pixelmat[r][c] = clip3(0., 255., abs(pelOri[r][c]) / (mheight * mwidth));
    }
  }
  return pixelmat;
}
```

wenier K freqPel[r][c] *= conj(freqFit[c]) / (pow(abs(freqFit[c]),2)+K)

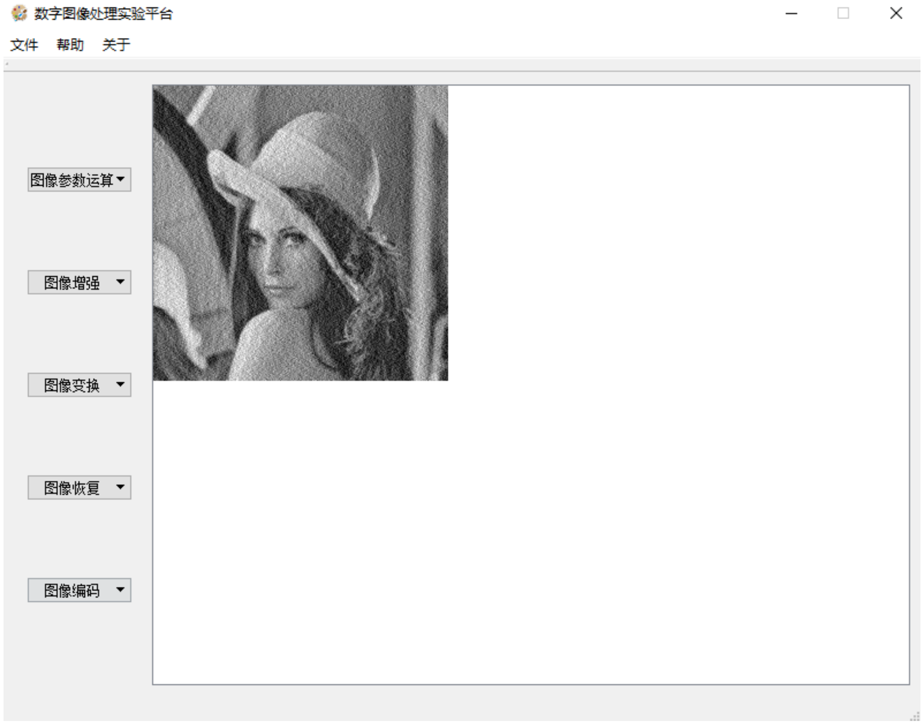


Figure 18: wenier