核医学機器工学概論３

フィルタ処理、畳込み

画像処理において用いられる実空間、空間周波数フィルタ処理とその演算方法である畳込みについて学ぶ

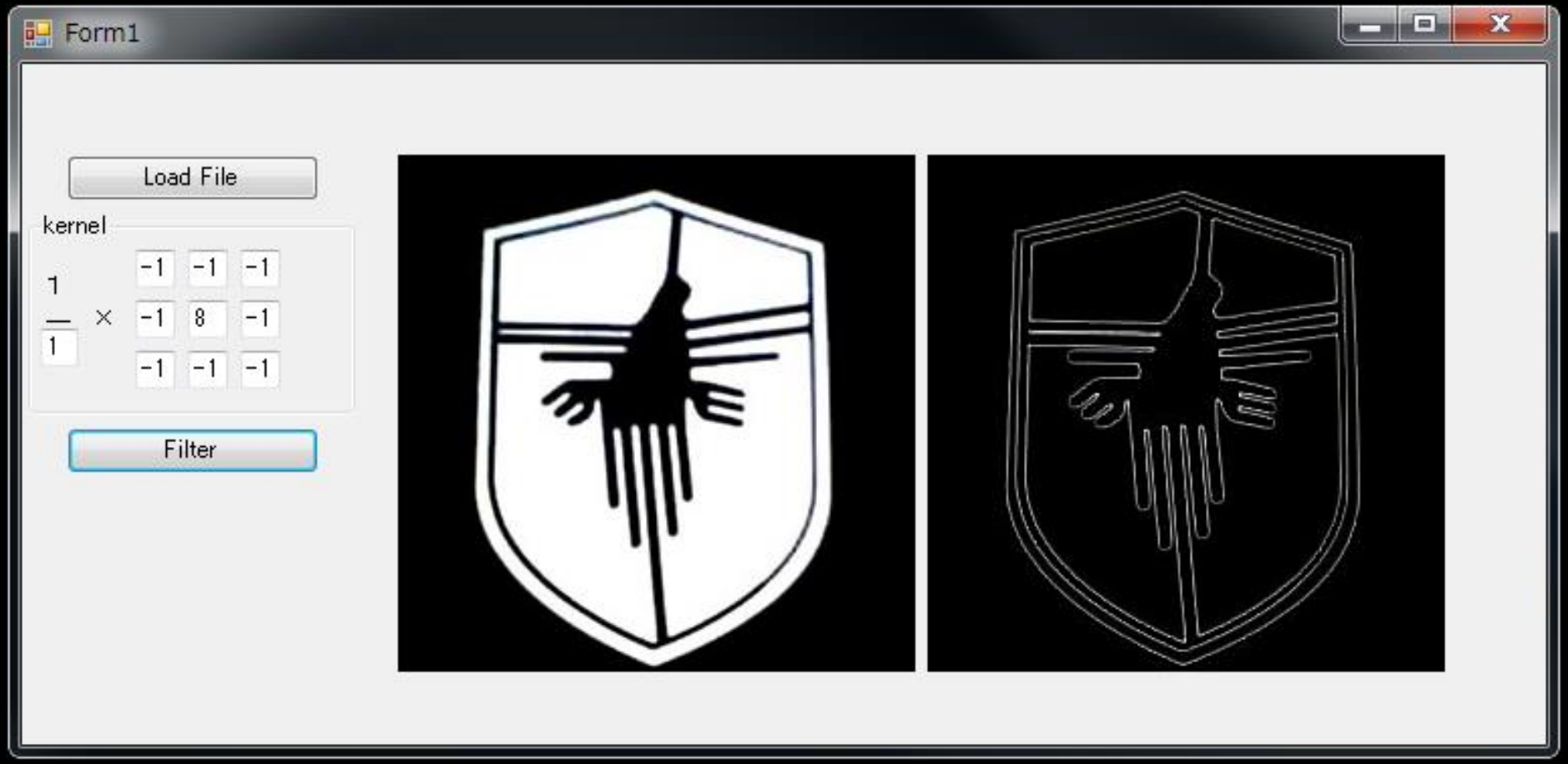
実空間フィルタ

フォルダFilter内にあるFilter.exeを起動し、「Load File」ボタンを押してBitmapまたはJPEG画像を選択。（プログラムフォルダ内にあるsample.jpgを使用）

「Kernel」の数値（９個のフィルタ行列の要素）を変更し、任意の3×3の実空間フィルタを作成する。

「Filter」ボタンを押し、作成したフィルタによりフィルタ処理を行う。

処理を行うと以下の様な画像が得られる



画像に平滑化フィルタ処理（smoothing）を行う

3x3 smoothing filter

$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{2}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

１画素の値を、その近傍にも
少し影響を及ぼすようにする。
合計が1になっている点に注目。
（カウント合計値は変化しない）

平滑化フィルタ　実空間での雑音除去フィルタ

フィルタ処理前後で画素値の総和が変わらないように
フィルタ内成分の和が1になっていることに注目。

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

移動平均フィルタ （成分値が全部同じ）

最も単純な 3x3 スムージングフィルタ。
中央画素の重み付けがない。
輪郭の鮮明さが損なわれる。

$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{2}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

0	0	$\frac{1}{13}$	0	0
0	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	0
$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$
0	$\frac{1}{13}$	$\frac{1}{13}$	$\frac{1}{13}$	0
0	0	$\frac{1}{13}$	0	0

荷重平均フィルタ
（成分値が中心部で大きい）

輪郭の鮮明さを維持するために
中心部に重みを付けた
3x3, 5x5 スムージングフィルタ。

平成18年　国家試験

問題97　3×3の空間フィルタを示す。画像の鮮鋭化に用いるのはどれか。
ただし、数字は重み係数を示す。

1.

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

2.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

3.

−1	−1	−1
0	0	0
1	1	1

4.

−1	0	1
−2	0	2
−1	0	1

5.

0	−1	0
−1	5	−1
0	−1	0

解答　5

```
void filter1( int x[ ][260], int y[ ][260] )
{                                        //  Weighted smoothing filter
        int      i, j ;

        for(j=2;j<=255;j++){ for(i=2;i<=255;i++){

            y[i][j] =  x[i-1][j-1] + x[i][j-1]*2 + x[i+1][j-1]
                     + x[i-1][j]*2 + x[i][j]*4     + x[i+1][j-1]*2
                     + x[i-1][j+1] + x[i][j+1]*2 + x[i+1][j+1]   ;
        }}
        for(j=2;j<=255;j++){ for(i=2;i<=255;i++){ y[i][j] /= 16; }}
}
```

輪郭の鮮明さを維持するために
中心部に重みを付けた
スムージングフィルタ。

1.

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

```
void filter2( int x[ ][260], int y[ ][260] )
{                                          // Simple smoothing filter
        int     i, j ;

        for(j=2;j<=255;j++){ for(i=2;i<=255;i++){

            y[i][j] =  x[i-1][j-1]  + x[i][j-1]  + x[i+1][j-1]
                     + x[i-1][j]    + x[i][j]    + x[i+1][j-1]
                     + x[i-1][j+1]  + x[i][j+1]  + x[i+1][j+1] ;
        }}
        for(j=2;j<=255;j++){ for(i=2;i<=255;i++){ y[i][j] /= 9; }}
}
```

2.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

最も単純なスムージングフィルタ。
中央画素の重み付けがない。
輪郭の鮮明さが損なわれる。

```
void filter3( int x[ ][260], int y[ ][260] )
{                                          //  Vertical Prewitt filter
      int    i, j ;

      for(j=2;j<=255;j++){ for(i=2;i<=255;i++){

         y[i][j] =   x[i-1][j-1]*-1 + x[i][j-1]*-1 + x[i+1][j-1]*-1
                   + x[i-1][j]*0      + x[i][j]*0      + x[i+1][j-1]*0
                   + x[i-1][j+1]      + x[i][j+1]      + x[i+1][j+1]    ;
      }}
}
```

Prewitt フィルタ （垂直方向）

1次微分（差分）エッジ検出フィルタ
上下方向に画素値が大きく変化する部位（輪郭）の抽出フィルタ

3.

−1	−1	−1
0	0	0
1	1	1

```
void filter4( int x[ ][260], int y[ ][260] )
{                                          // Horizontal Sobel filter
    int   i, j ;

    for(j=2;j<=255;j++){ for(i=2;i<=255;i++){

        y[i][j] =  x[i-1][j-1]*-1 + x[i][j-1]*0 + x[i+1][j-1]
                 + x[i-1][j]*-2    + x[i][j]*0     + x[i+1][j-1]*2
                 + x[i-1][j+1]*-1 + x[i][j+1]*0 + x[i+1][j+1]   ;
    }}
}
```

Sobel フィルタ （水平方向）

1次微分（差分）エッジ検出フィルタ
左右方向に画素値が大きく変化する部位（輪郭）の抽出フィルタに中央部の重み付けを加えている。

4.

−1	0	1
−2	0	2
−1	0	1

```
void filter5( int x[ ][260], int y[ ][260] )
{                                          //  Unsharp masking filter
    int    i, j ;

      for(j=2;j<=255;j++){ for(i=2;i<=255;i++){

         y[i][j] =   x[i-1][j-1]*0 + x[i][j-1]*-1 + x[i+1][j-1]*0
                   +  x[i-1][j]*-1   + x[i][j]*5      + x[i+1][j-1]*-1
                   +  x[i-1][j+1]*0  + x[i][j+1]*-1 + x[i+1][j+1]*0 ;
        }}
}
```

4近傍 先鋭化フィルタ

アンシャープマスキング。
ぼやけた部位をマスクする。
上下および左右方向に画素値が
大きく変化する部位（輪郭）の強調

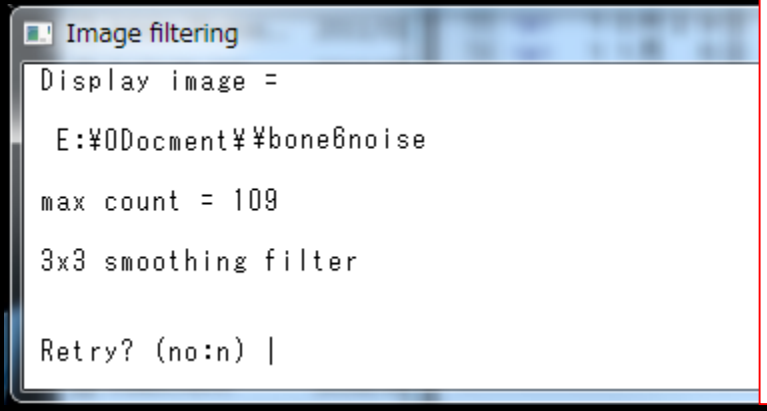
5.

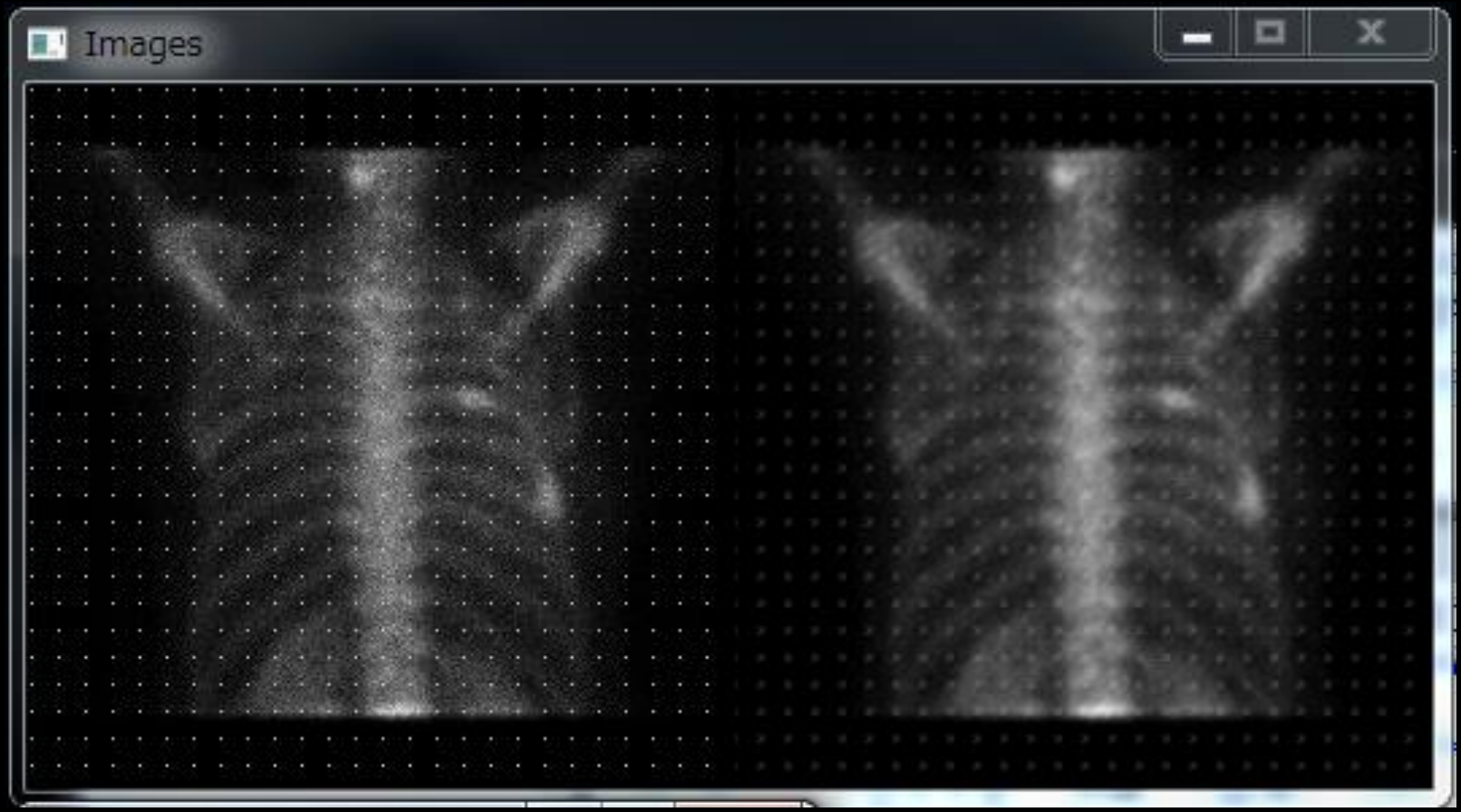
0	−1	0
−1	5	−1
0	−1	0

Smoothing.exe

実行結果

Bone6noise画像を選択。





画像に 中央値フィルタ処理（median）を行う

3x3 median filter

X[i−1][j−1] = med[1]	X[i][j−1] = med[2]	X[i+1][j−1] = med[3]
X[i−1][j] = med[4]	X[i][j] = med[5]	X[i+1][j] = med[6]
X[i−1][j+1] = med[7]	X[i][j+1] = med[8]	X[i+1][j+1] = med[9]

座標[i][j]の画素値を、その近傍を含む９画素の値 med[1]～med[9]の中央値に置き換える処理。

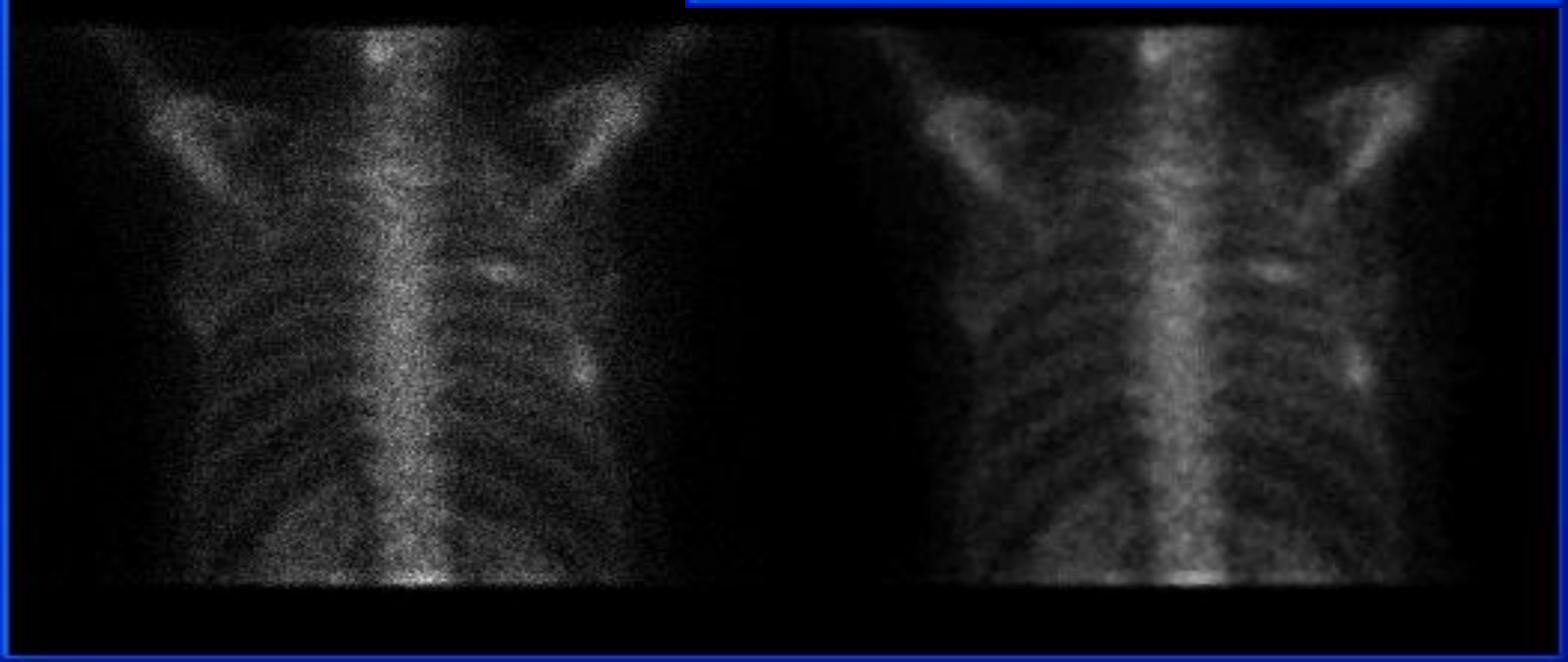
median.c 実行結果

Median filtering

```
Display image =
 C:¥jouhou¥program3¥bone2
max count = 46
3x3 median filter

Retry? (no:n) |
```

Quantified PET Image



median.c
メディアン フィルタ

```
void median( int x[ ][260], int y[ ][260] )
{
        int  i, j, k, mi, mj, ni, nj, yn, min, med[10];

        for( j=2; j<=255; j++){ for( i=2; i<=255; i++){

            k = 1;
            for( mi=-1; mi<=1; mi++){ for( mj=-1; mj<=1; mj++){

                    med[k] = x[i+mi][j+mj] ;   k++ ;
            }}

            for( ni=1; ni<=8; ni++){  for( nj=8; nj >= ni; nj--){

                if( med[nj+1] < med[nj] ){

                    min=med[nj+1];  med[nj+1]=med[nj];  med[nj]=min;
                }
            }}

          y[ i ][ j ] = med[ 5 ];
        }}
}
```

```
k = 1 ;
for( mi=-1; mi<=1; mi++){ for( mj=-1; mj<=1; mj++) {
        med[ k ] = x[ i+mi ][ j+mj ] ;    k++ ;
}}
```

座標 [i][j] と、その近傍の画素カウント値を
配列 med[1] ～ med[9] に代入している。

配列 med は、int med[10]; と宣言している。
int med[9]; でも良さそうに思えるが、

C言語では、配列数は 0番目から数えている ので
med[9]; と宣言すると、med[0] から med[8] の9個の
配列要素だけが宣言され、med[9] は宣言されない。

```
for ( ni=1; ni<=8; ni++) {   for ( nj=8; nj >= ni; nj−−) {
        if ( med[nj+1] < med[nj] ){
                min=med[nj+1];  med[nj+1]=med[nj];  med[nj]=min;
        }
} }
```

med[1] から med[9] の値を並べ替えている（ソート　sort）。この for ループが終了すると、med[1] から med[9] の中の最小値が med[1] に、最大値が med[9] に代入されている。

バブルソートのアルゴリズム　Bubble sort algorithm

コップ底の泡が、軽いものから先に順番に昇っていくような様子をプログラムで表現している。ソート法のアルゴリズムとしては最も非効率的な方法だが単純で理解しやすい。

はじめに ni が 1 の 値で nj の ループに入る。

nj の 値は 8 から1 までの値に変化しながら ループを回る。

最初は nj が 8 なので、if 文の中身を具体的に書くと

```
if ( med[ 9 ] < med[ 8 ] ){
    min=med[ 9 ];  med[ 9 ]=med[ 8 ];  med[ 8 ]=min;
}
```

もし med[9] が med[8] より小さければ med[9] と med[8] の値を入れ替える。変数の入れ替え作業を swap という。swap作業には 第三の変数（ここでは min）が必要になる。

ni が 1 の 状態で、nj が 8 から 1 まで変化し終わると、med[1]の値は、med[1]からmed[9]の最小値になっている。

小さい値が泡のように上に昇っていくので バブルソート法。

次に ni が 2 の 値で nj の ループに入る。

nj の 値は 8 から 2 までの値に変化しながら ループを回る。

（ med[1] は最小値が確定なので ループに入る必要がない。）

ni が 2 の 状態で、nj が 8 から 2 まで変化し終わると、
med[2] の値は、med[2] から med[9] の中の最小値に
なっている。

次に ni が 3 の 値で nj の ループに入る。
nj の 値は 8 から 3 までの値に変化しながら ループを回る。
（ med[1]、med[2] は 1番目、2番目に小さい値が確定なので
ループに入る必要がない。）

同様の操作を ni が 8 になるまで繰り返すと、
med[1] に最小値、med[2] に2番目に小さい値、med[3]に
3番目に小さい値、 ・・、med[9]に9番目に小さい値が入る。

中央値は med[5] なので、 y[i][j] = med[5] ; となる。

smoothing filter は、輪郭が不明瞭化し画像がぼやける。

median filter は smoothing filter より輪郭が保たれる。

median filter は、微細なノイズを除去する効果がある。

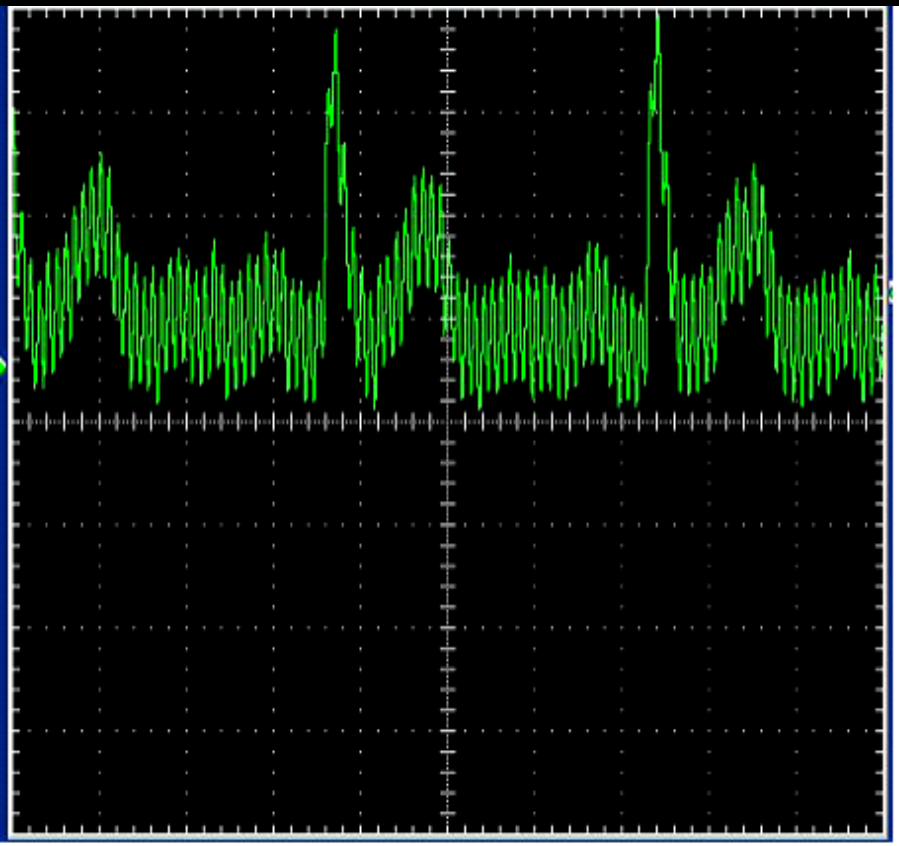
bone6noise は、10ピクセルおきに画素値 100 を入れた像。

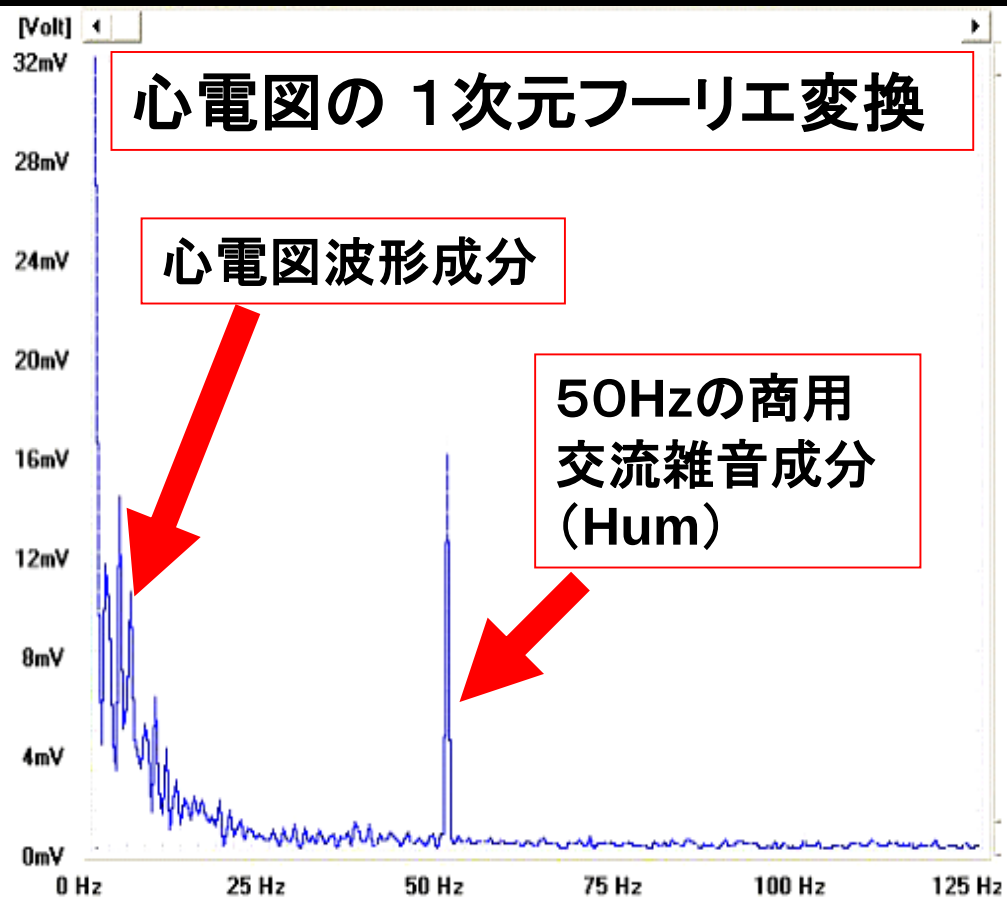
median では、このノイズを完全に消すことができる。

smoothing ではノイズを完全には消せない。

フーリエ変換（Fourier Transform）は、波形データなどが含む**周波数成分（スペクトル）**を分析する方法。

交流雑音（ハム）を含む心電図をフーリエ変換すると、50Hzの周波数成分が多く含まれていることが判る。





フーリエ変換（Fourier Transform）の基本式は、波形データが 連続データであることに基づいている。

Fourier 変換

ある時間信号 $g(t)$ $(0 \leqq t \leqq T)$ を

周期 $\frac{T}{n}$（周波数 $\frac{n}{T}$）$(n = 1, 2, 3, \cdots)$ の

cos 成分 a_n，sin 成分 b_n と

直流成分 a_0 で表現すると

$$g(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos\left(2\pi \frac{n}{T} t\right) + \sum_{n=1}^{\infty} b_n \sin\left(2\pi \frac{n}{T} t\right)$$

Fourier 変換 $(t \rightarrow f)$

$$G(f) = \int_{-\infty}^{\infty} g(t)\, e^{-j(2\pi f t)}\, dt$$

逆 Fourier 変換 $(f \rightarrow t)$

$$g(t) = \int_{-\infty}^{\infty} G(f)\, e^{j(2\pi f t)}\, df$$

2次元 Fourier 変換

- 1次元 Fourier 変換

  波形 $g(t)$ の 周波数分布 $G(f)$ を求める.

- 2次元 Fourier 変換

  平面上に分布する量 $g(x,y)$ の
  周波数分布 $G(f_x, f_y)$ を求める式は,

$$G(f_x,f_y)=\iint g(x,y)\, e^{-j(2\pi f_x x)} e^{-j(2\pi f_y y)}\, dx\,dy$$

- 画像の周波数成分

  低周波成分は 大まかな濃度変化の部分の情報,
  高周波成分は 急激な濃度変化の部分の情報
  を含む.

  (画像がボケる = 高周波成分が消失する)

高速フーリエ変換（ FFT：Fast Fourier Transform ）

フーリエ変換を高速に計算するアルゴリズム。

1942年に Danielson と Lanczos が発明。

プログラムによるフーリエ変換は、**波形データが離散的**なので、基本式に現れる $e^{(2\pi f t)i}$ の項を、W^{nk} と変形して巧みに解いている。

高速逆フーリエ変換（ IFFT：Inverse FFT ）も ほとんど同じアルゴリズムなので、両方可能な関数が作れる。

FFT 、IFFT におけるデータの制限としては、

データ数が 2の階乗でなければならない。

FFT　Fast Fourier Transform

FFT.exeを起動し、open fileから同フォルダ内にあるboneを選択する。
※bone：^{99m}Tc-MDPのSPECT画像。骨シンチ。

FFTをクリックして2次元フーリエ変換を行い、replaceをクリックして、パワースペクトル画像を再配置する。

discoverをクリックすると、逆フーリエ変換された画像が表示される。

coverをクリックすると、右側の画像が消える。

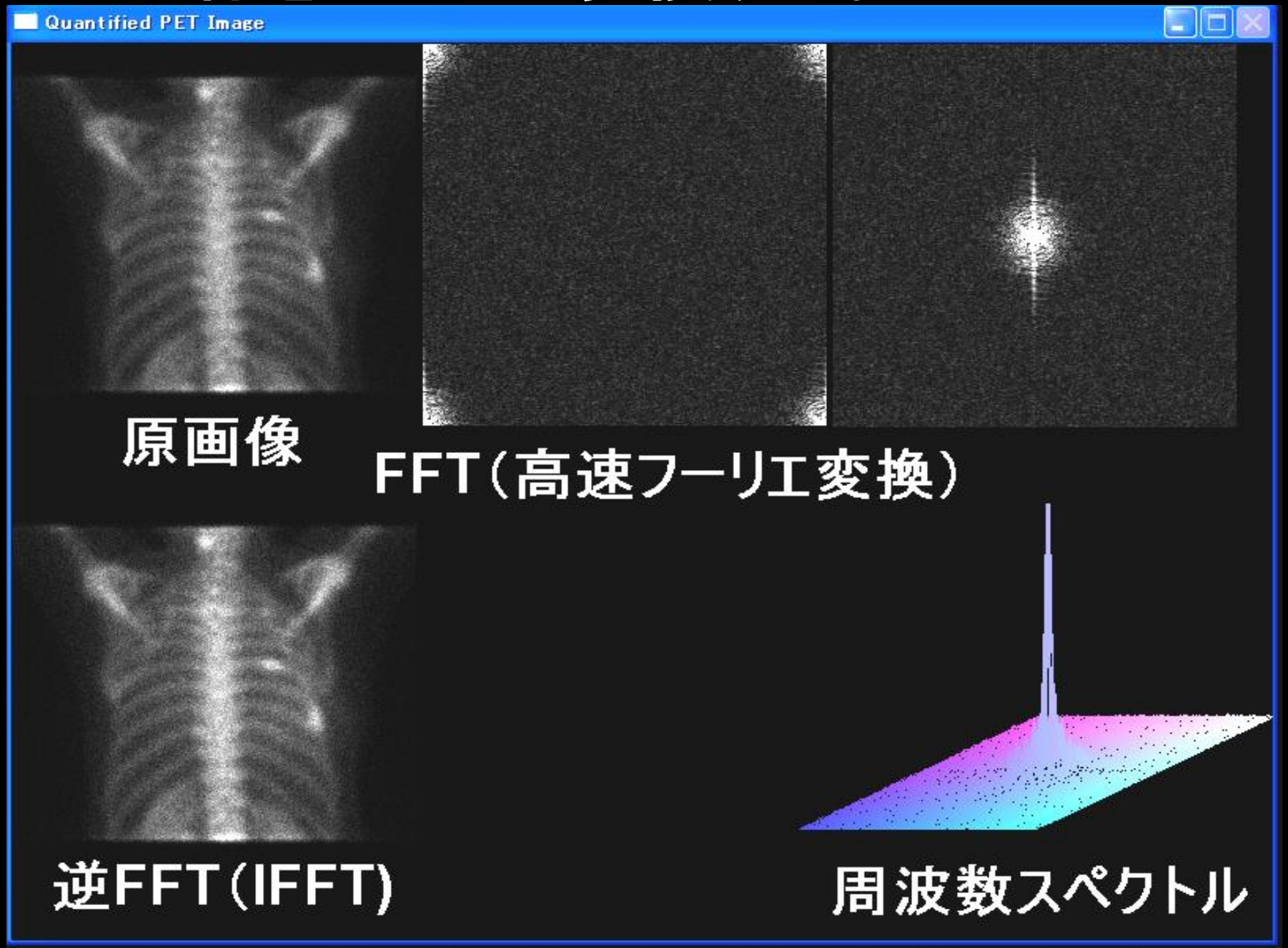
空間周波数フィルタ

フォルダButterworth内にあるButterworth.exeを起動し、「Load File」ボタンを押して、
プログラムフォルダ内にあるboneファイルを読み込む。

「display sin cos component」ボタンを押し、原画像のフーリエ変換を行い、のsin成分とcos成分の画像を表示させる。

「FFT spectrum」ボタンを押し、パワースペクトルを表示する。

画像を フーリエ変換する。　FFT.c

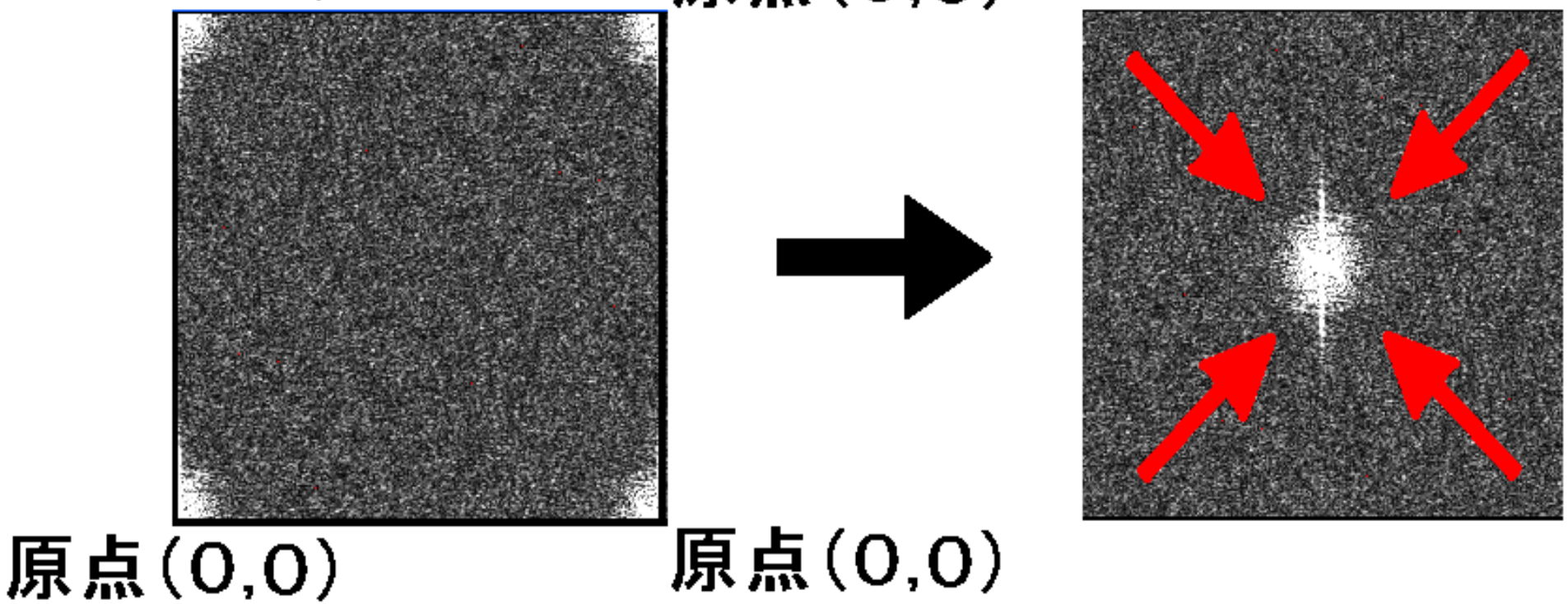


２次元 FFT の出力である ２次元周波数分布

データは、４隅が原点 (0,0) （= 直流成分）に

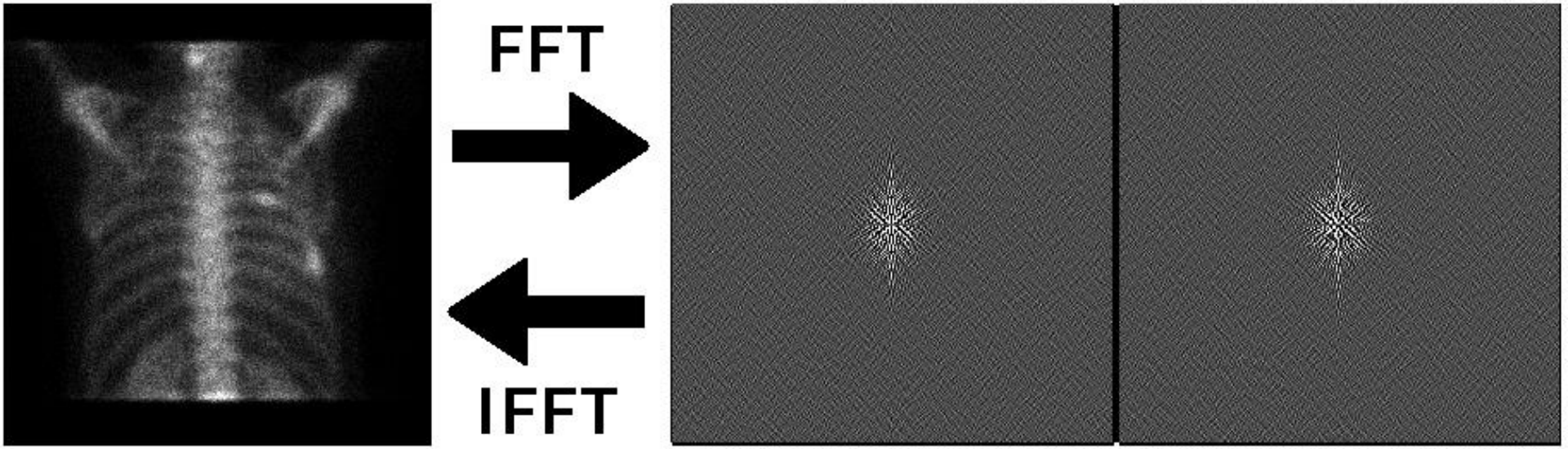
なっているのが面倒な点である。

中心が原点になるように各々 1/4 領域を反転する。



原画像 img を x[ ] に入れて２次元 FFT を行うと、2次元周波数分布のsin成分が x[ ] に、cos成分が y[ ] に出力される。

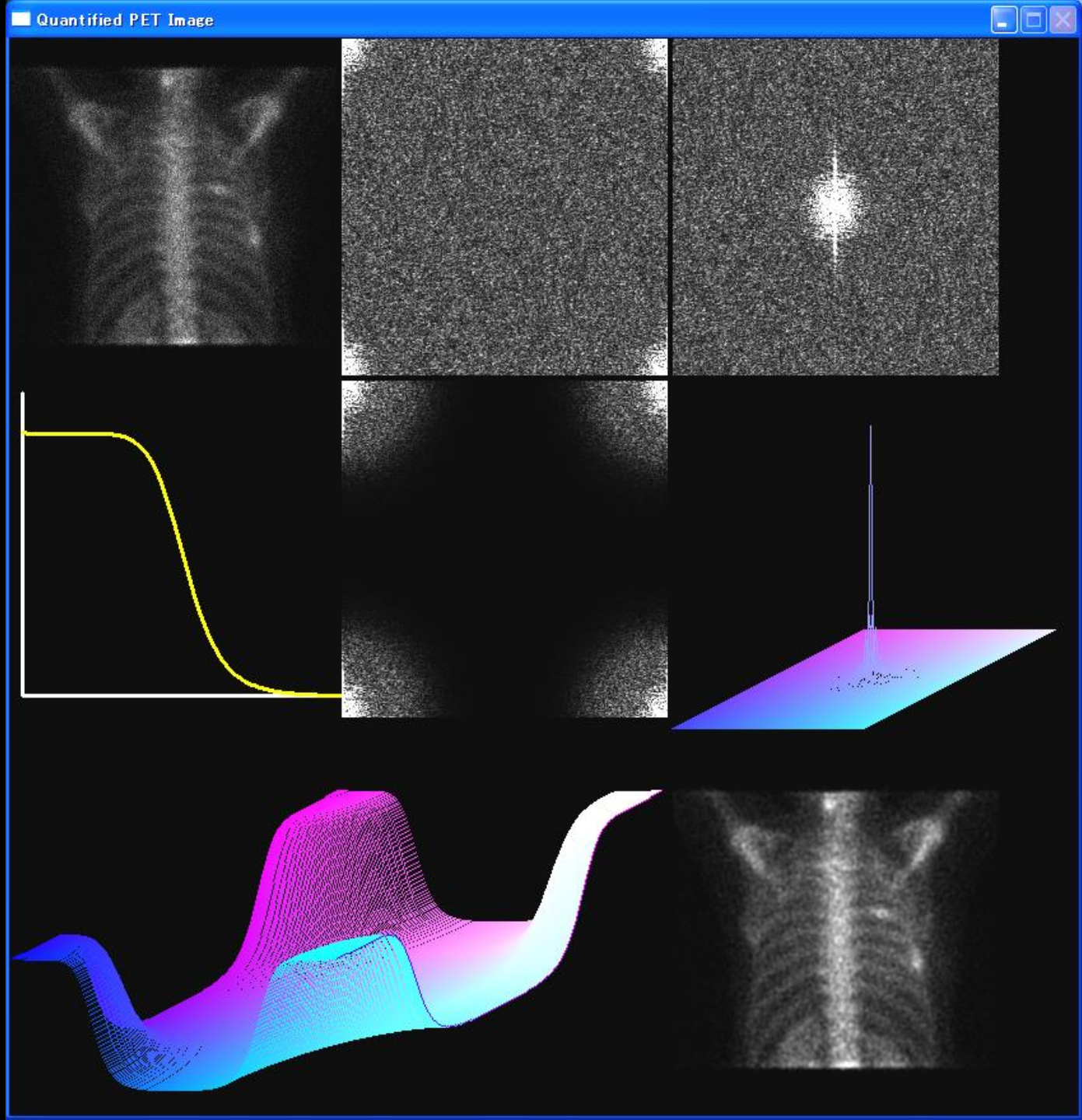
x[ ] と y[ ] を、４隅が原点 (0,0) になっている状態で ２次元逆フーリエ変換 (IFFT) の関数に入力すると、x[ ] に 元に戻った像が出力される。



Butterworth.c

周波数空間で
フィルタ処理を行う

バターワース
フィルタ

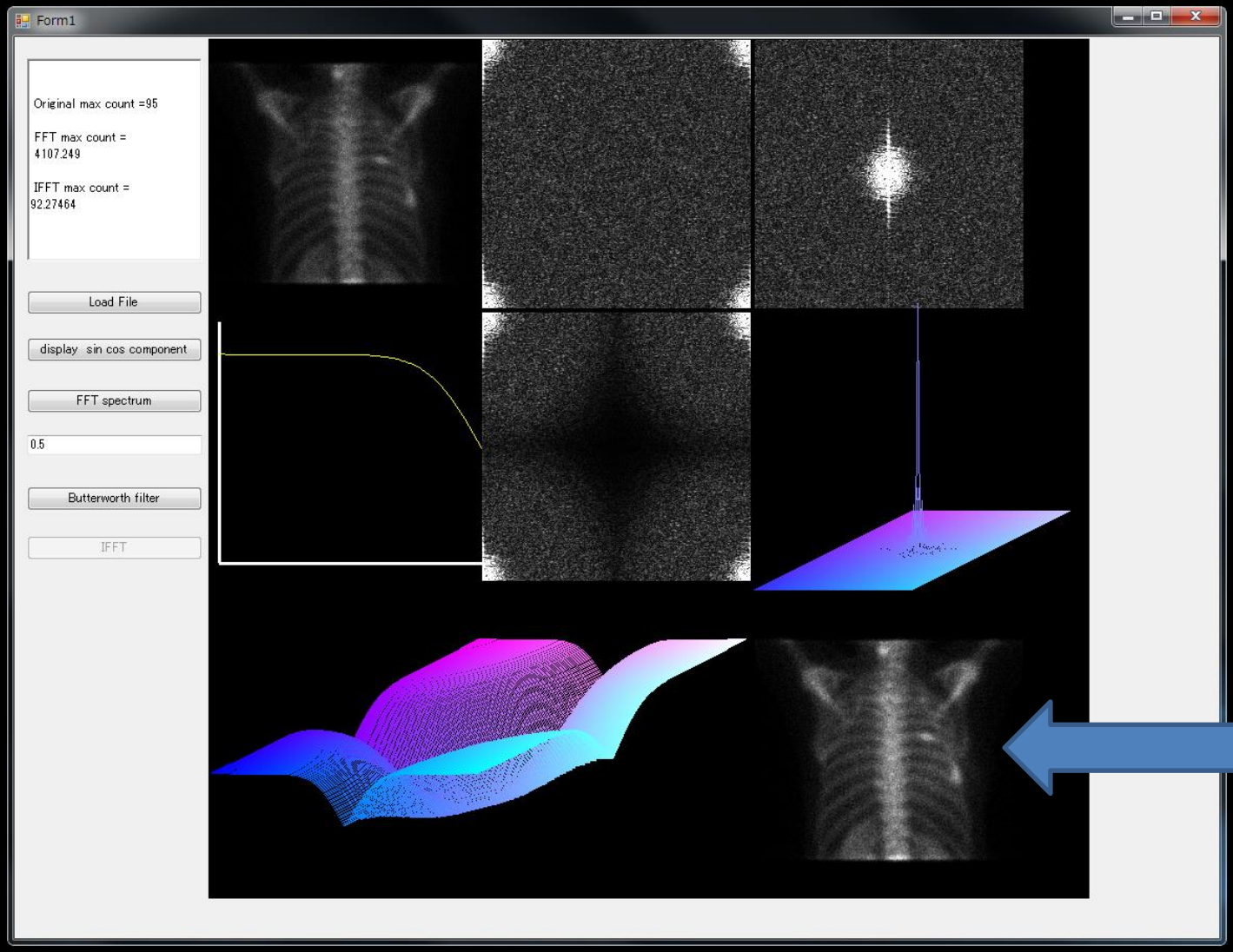


テキストボックス内に0.5～0.01の任意の遮断周波数を設定する。

「Butterworth filter」ボタンを押し、設定した遮断周波数からバターワースフィルタを作成し、空間周波数フィルタ処理を行う。

「IFFT」ボタンを押し、逆フーリエ変換を行い、フィルタ処理後の画像を表示する。

結果として、以下の様な画像が得られる。

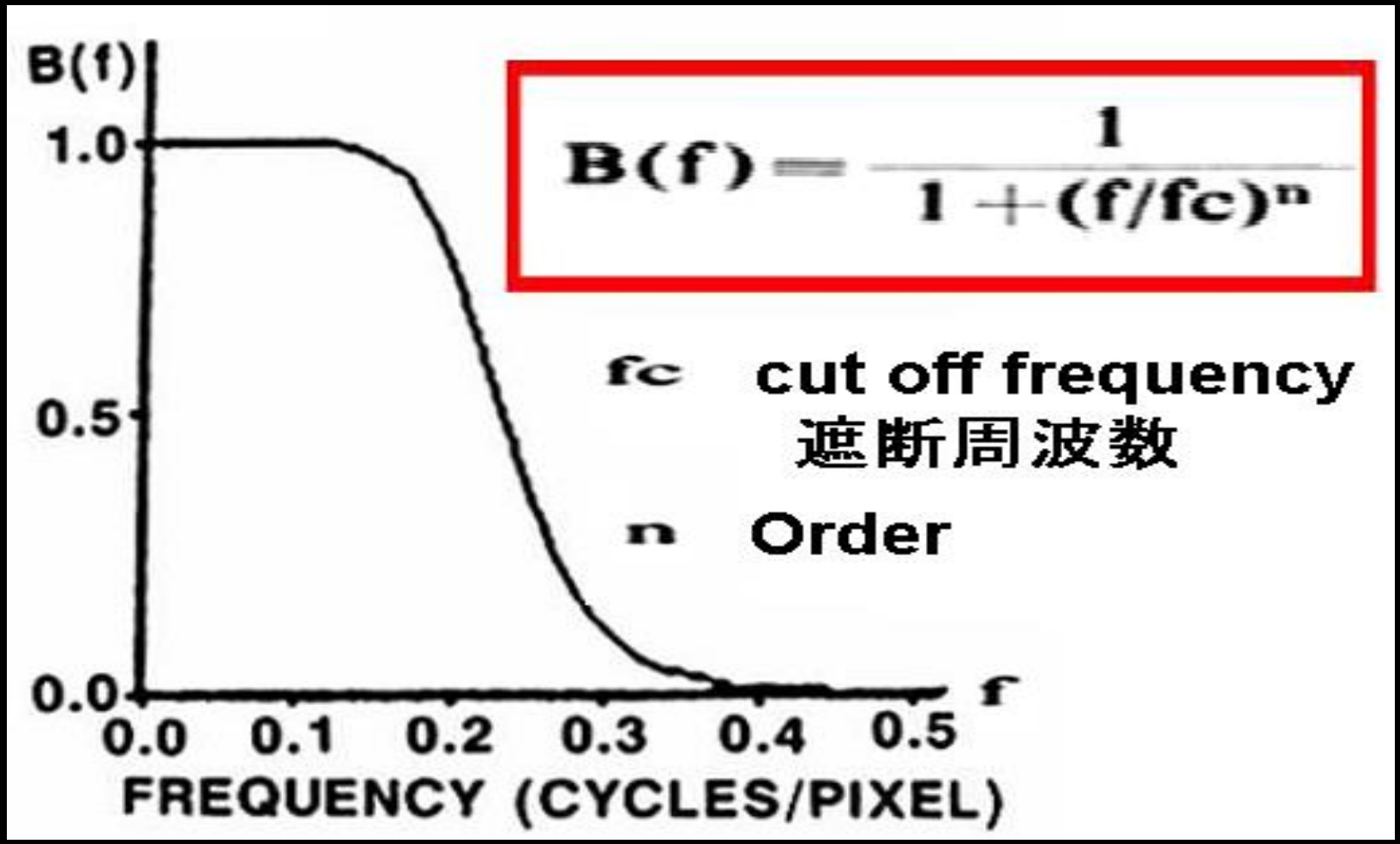


出力画像

画像 を 周波数空間でフィルタ処理する。

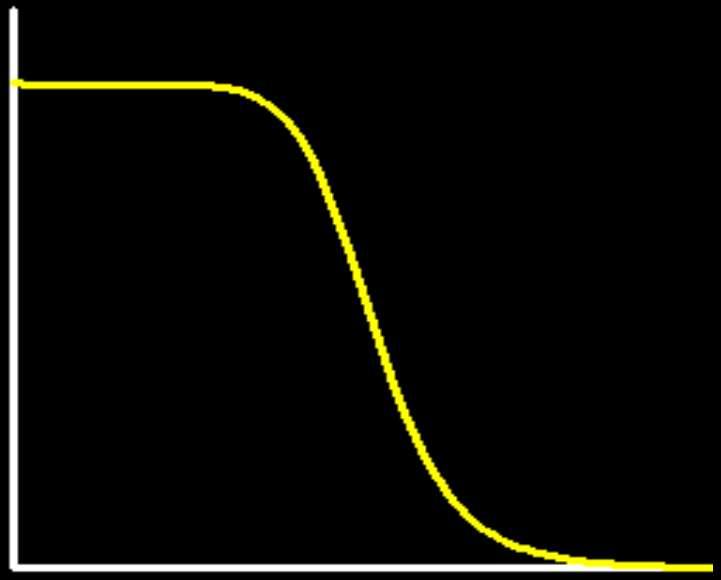
バターワースフィルタ　Butterworth filter

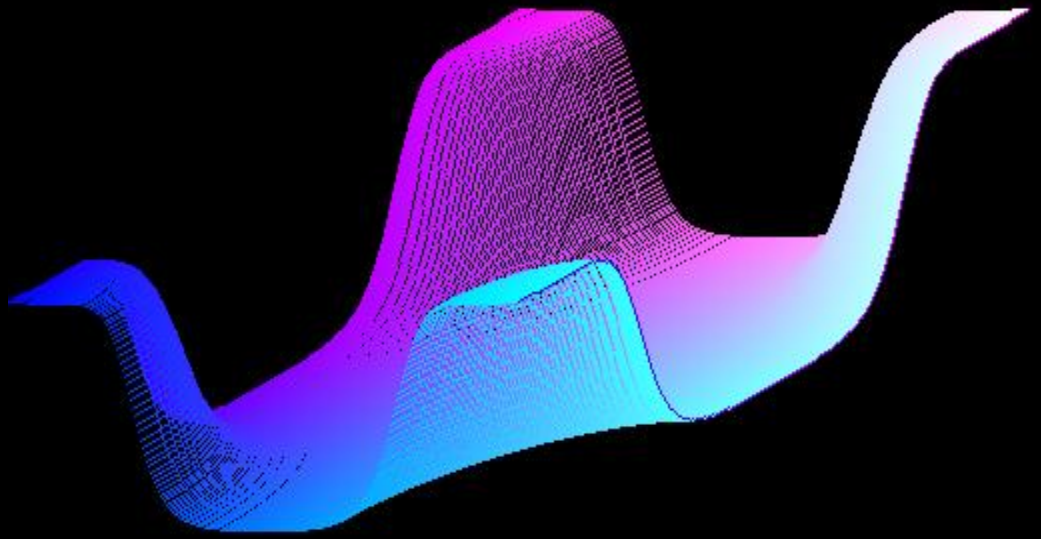
ノイズ成分が相対的に多い高周波成分を抑制。



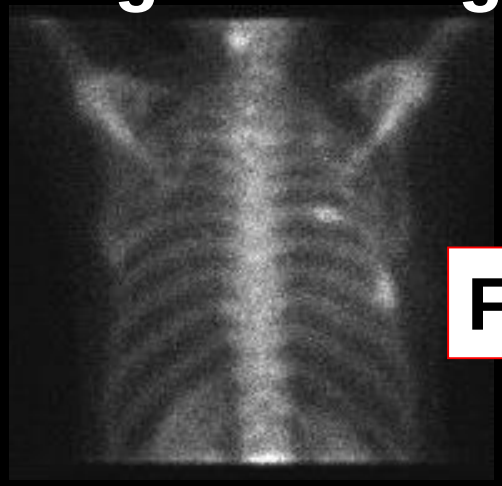
１次元の Butterworth filter を ２次元にしている。

２次元周波数分布のsin成分 x[ ] 、cos成分 y[ ] は
４隅が原点 (0,0) （ = 直流成分）になっている。
４隅近傍が低周波成分の領域。
したがって、２次元の Butterworth filter B2[ ][ ]
を、x[ ]、y[ ] の４隅を残すように作成している。



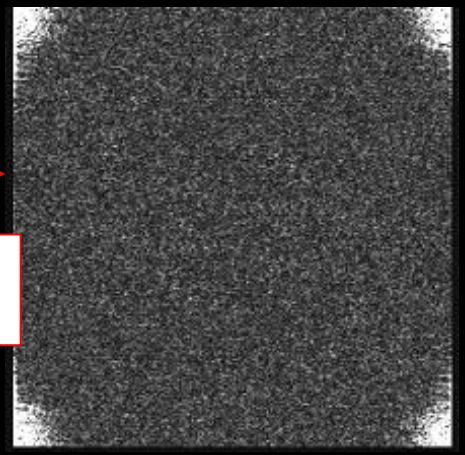


Original Image

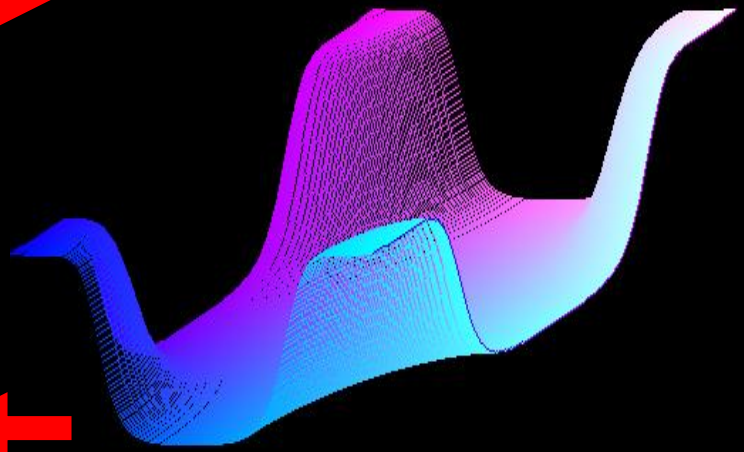


FFT

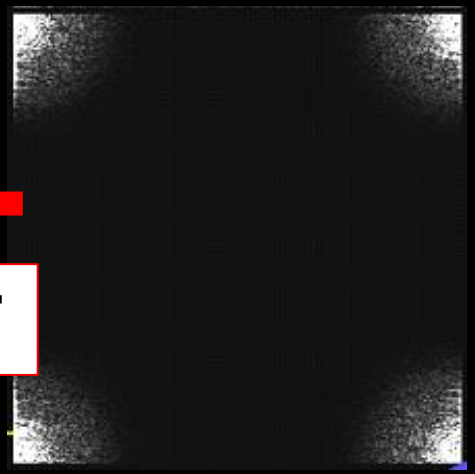
Frequency spectrum



Frequency space filtering

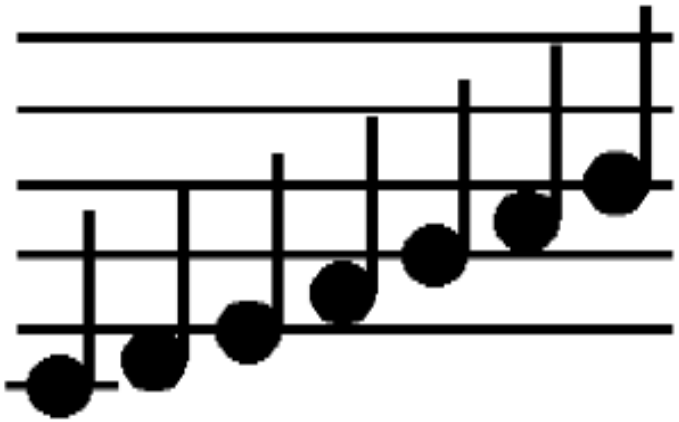


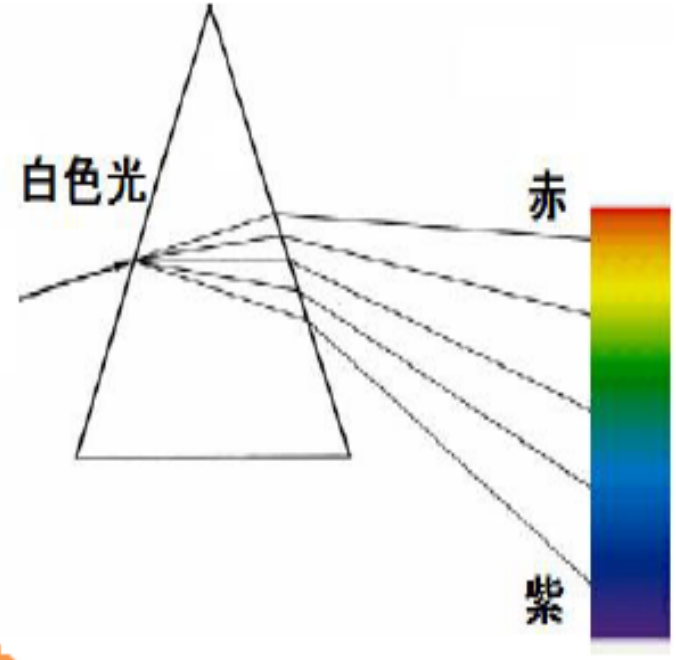
Filtered Frequency spectrum



IFFT

Filtered Image





周波数空間を理解することは難しい印象を持つが、周波数空間や周波数分布を表現しているものは一般にも結構ある。

たとえば楽譜は、音楽の周波数分布、曲のフーリエ変換とも解釈できる。五線譜の下側の音符は低音、低周波数成分を表し、上側の音符は高音、高周波成分を示している。

虹は白色光に含まれる色の周波数分布を示している。雨粒中で屈折しやすい高周波の紫色成分から、屈折しにくい低周波の赤色成分まで、周波数の順序で色が並んでいる。