```
min = img[1][1] ;
for( j = 1;  j <= 256; j++ ) { for( i = 1; i <= 256; i++ ) {
   if ( img[ i ][ j ] < min ) min = img[ i ][ j ] ;
} }
```

最小値を求める常套手段。

forループが終了すると変数 min に最小値が入る。

始めに min を 適当な値に設定しておく。
データの最初の値を入れておくのが通常の方法。

次々と画像内の画素カウントと min を比較し、
画素カウントのほうが小さい場合は、
その画素カウントを 変数 min に代入する。

```
//------- Calculate count density ---    計数密度の計算
    total_count = 0 ;
    for(j=1;j<=256;j++){ for(i=1;i<=256;i++){
            total_count += img[i][j];
    }}
     density = (double)total_count / (43.4 * 43.4) ;
    printf("¥n¥n Count density = %.2lf (count/cm2) ", density );
```

実数表示 %lf は、無駄に長い小数点以下表示が邪魔。
%.2lf と記述すると、小数点以下２桁までの表示になる。

この計算法では、カウントが 0 の領域も
計数密度を測る面積に加えられる問題が生じる。

画素値が 0 より大きい部位の計数密度算出

```
//------- Calculate count density ---   program3c.c 計数密度

    total_count = pixel = 0 ;

    for(j=1;j<=256;j++){ for(i=1;i<=256;i++){

            if( img[i][j] > 0 ){ total_count += img[i][j];  pixel++ ; }
    }}

    area = (double)pixel * 0.1695 * 0.1695 ;
                                                    //  0.1695  1画素の実長
    density = (double)total_count / area ;

printf("¥n¥n Count density = %.2lf (count/cm2) ", density );
printf("¥n¥n Total count     = %d (counts) ", total_count );
printf("¥n¥n Total pixels    = %d (pixels) ", pixel );
printf("¥n¥n                 = %.1lf (cm2) ", area );
```

total_count = pixel = 0 ;　は、

total_count = 0 ;　pixel = 0 ;　と同じ。

C言語では、複数の同じ型の変数に、
一度に同じ値を代入する文を記述できる。

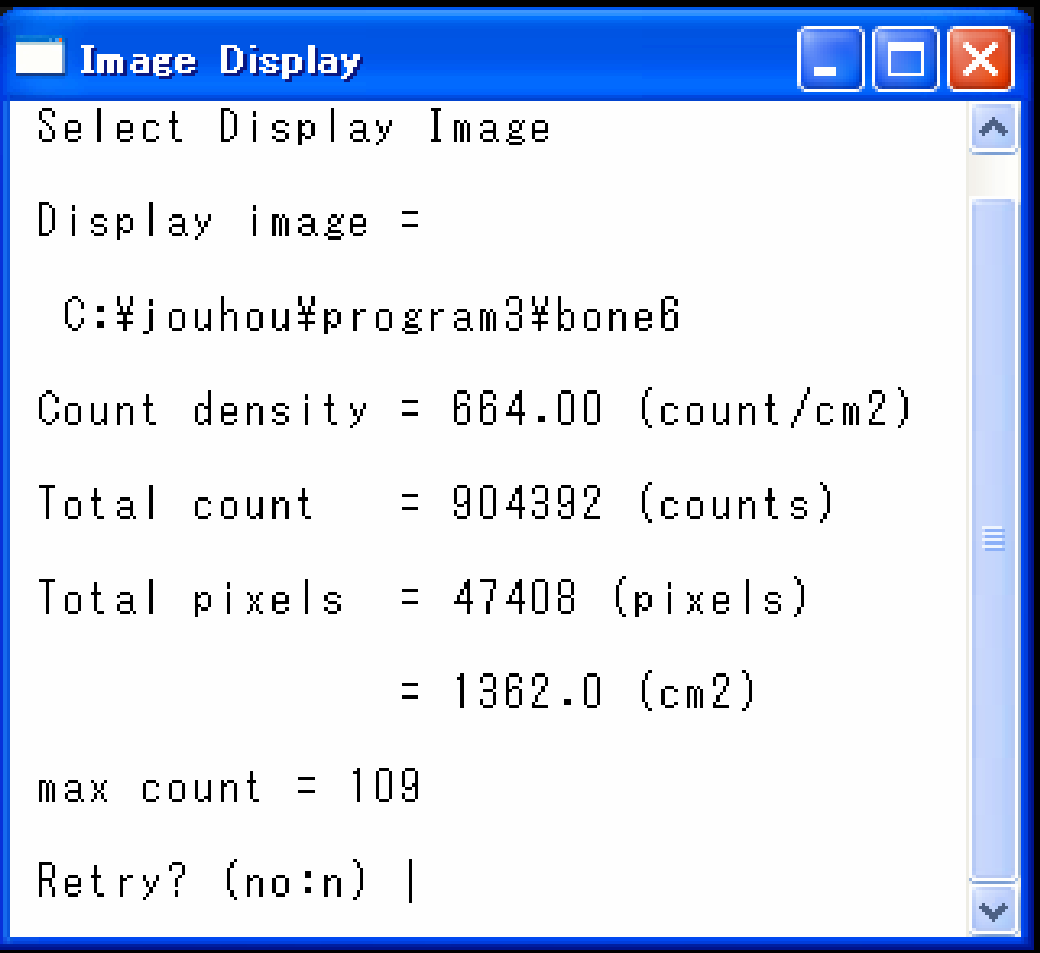
```
for(j=1;j<=256;j++){ for(i=1;i<=256;i++){
    if( img[i][j] > 0 ){ total_count += img[i][j];  pixel ++ ; }
}}
```

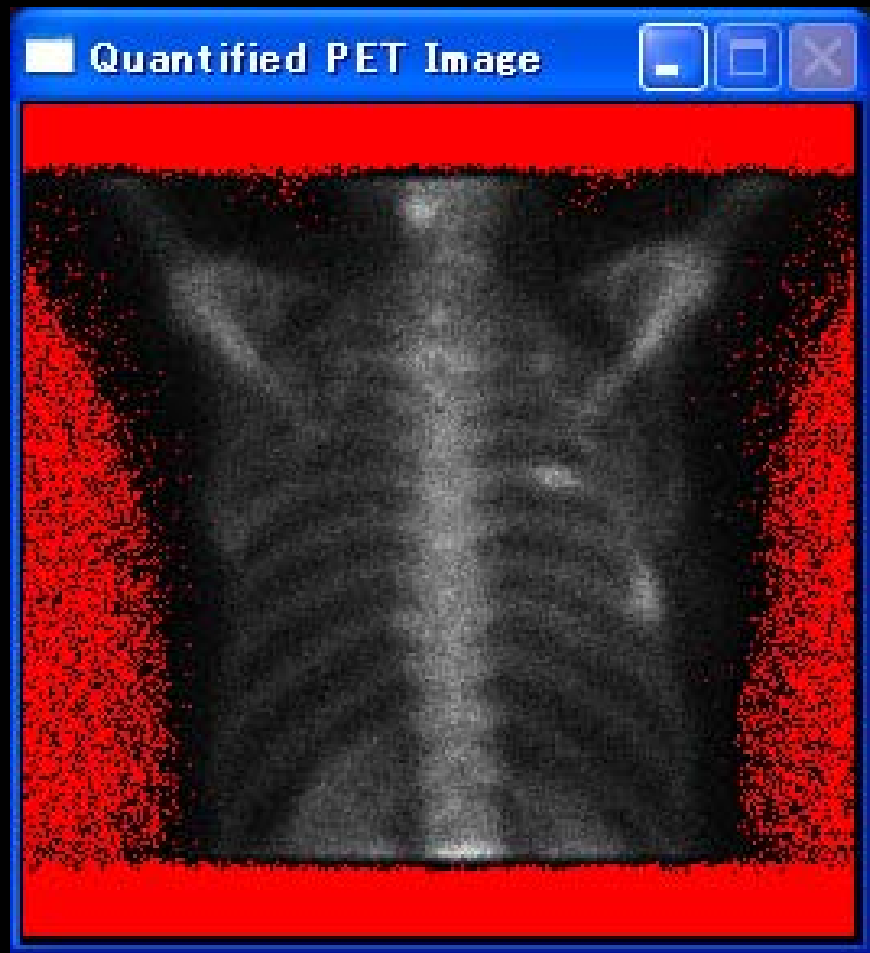
画素値が 0 より大きい部位の
カウント合計 と ピクセル合計を 求めている。

```
//------------ Display image -----    program3c.c
  GraphicWindow(260,260,BK_BLACK);
      for(j=1;j<=256;j++){ for(i=1;i<=256;i++){
          count = img[i][j];
              count *= 255 / maxcount;
                    if(count < 0   ) count = 0  ;
                    if(count > 255 ) count = 255;
              SetColor( RGB(count, count, count) );
              if( count == 0 ){ SetColor( RED ); }
              SetPixel( i , j );
  }}
```

program3c.c 実行結果

カウントが 0 の領域を赤く描画している。
この領域は、計数密度の算出には入れない。





```
if( count == 0 ){ SetColor( RED ); }
```

画素値 count が 0 ならば、赤色を描画する。
== は、C言語では等号（等しいこと）を意味する。

SetColor関数は、カッコ内に色を指定する
COLORREF変数を RGB関数を介して入力するが、
単色を指定する場合は、このような簡便法が使える。
（ RED は、RGB(255, 0, 0) と同じ。）
HU. h に、下記の単色設定が用意されている。

BLACK	: 黒色	RED	: 赤色
GREEN	: 緑色	BLUE	: 青色
PURPLE	: 紫色	SKYBLUE	: 水色
YELLOW	: 黄色	WHITE	: 白色

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。

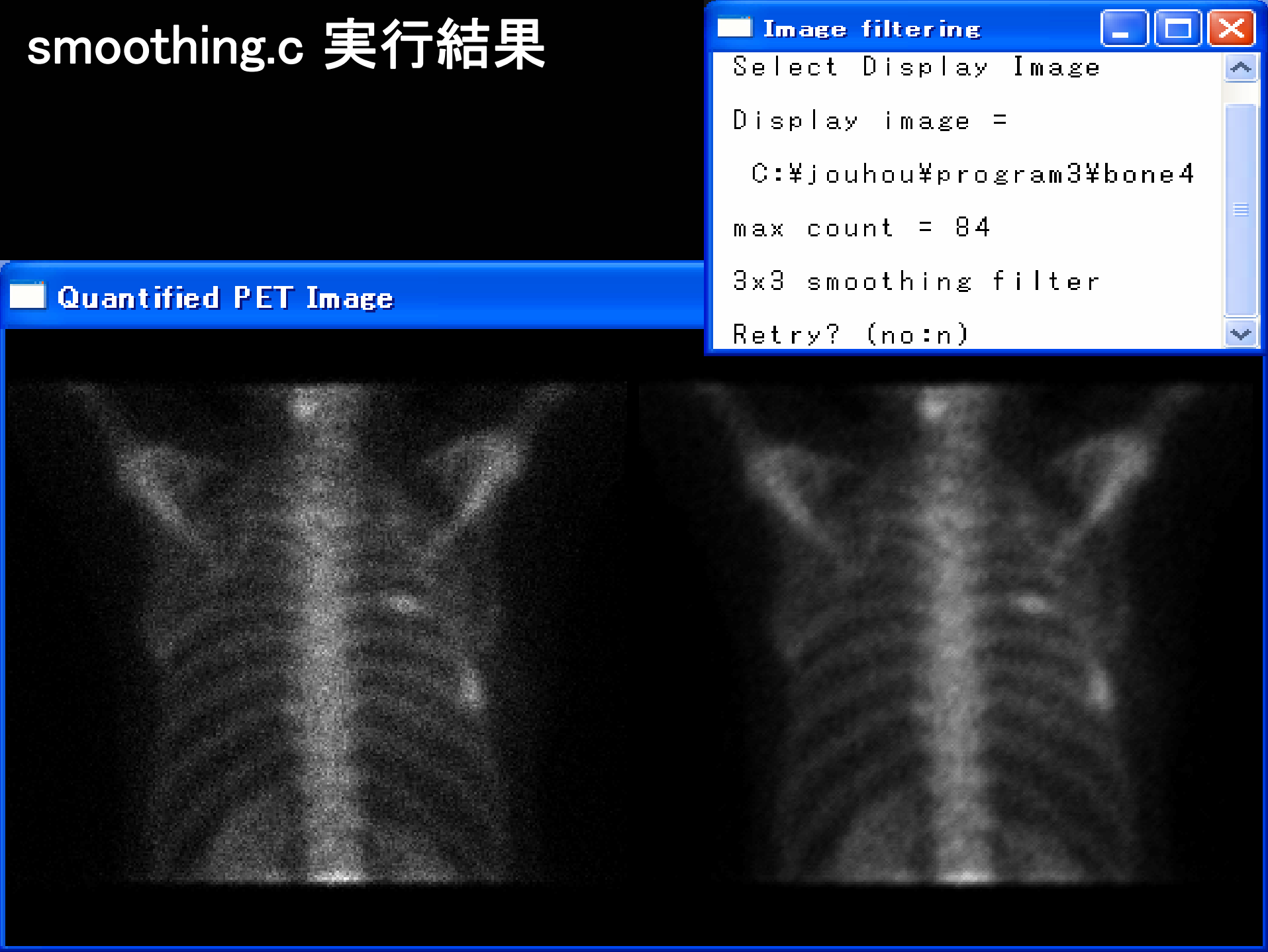
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{2}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

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。

smoothing.c 実行結果



```
// smoothing.c      画像の平滑化フィルタ処理
#include <HU.h>
void Main(void)
{
        char            a, b, yn, f[100];
        int             i, j, count, maxcount, xshift = 260 ;
        static  int     img[260][260],  img2[260][260] ;
        FILE *fp;

TextWindow(10,10,300,200); Title("Image filtering");
GraphicWindow(520,260,BK_BLACK);
START:      printf("¥n Select Display Image ¥n");
                ClearWindow();
```

```
int    xshift = 260 ;
```

C言語では、変数宣言時に、
変数の内容を代入することができる。

```
ClearWindow();
```

GraphicWindow 関数で作ったグラフィック画面を
クリア（何も描画していない状態）にする。

カッコ内には何も記述しない（入力値のない関数）。

```
static  int    img[260][260],  img2[260][260] ;
```

変数の static 宣言　　画像を扱う時の重要な裏技

画像を入れる配列は巨大配列になる。普通に変数を宣言すると、変数はプログラムを動かすメモリ空間に記憶される。

プログラムを実行する重要なメモリ空間に巨大配列が記憶されるとプログラム実行メモリが不足して障害を起こす。

そこで、巨大な配列を別のメモリ空間（静的メモリ）に記憶させる方法が、static 宣言。

本当の static 宣言の意義は、変数の記憶場所（ポインタ）を固定したい場合に用いるが 画像配列宣言にも重宝する。

```
//------------ Get image data -------------

        GetFileName(f,0);

        printf("¥n Display image = ¥n¥n  %s", f);

        fp=fopen(f,"rb");

        for(j=1;j<=256;j++){for(i=1;i<=256;i++){

                a = fgetc(fp);  b = fgetc(fp);

                 img[i][j] = a * 256 + b;
        }}
        fclose(fp);
```

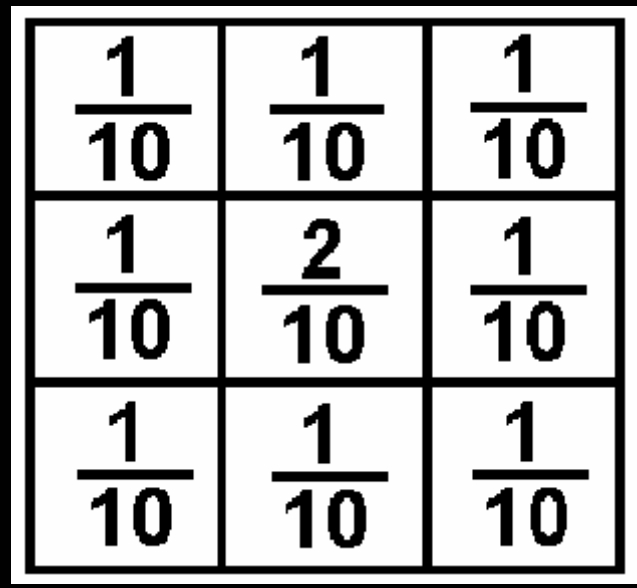
```
//------------------- Get Max count ----------------------
   maxcount = 0 ;

        for(j=1;j<=256;j++){ for(i=1;i<=256;i++){

            if(img[i][j] > maxcount) maxcount = img[i][j];
        }}

        printf("¥n¥n max count = %d ", maxcount);

//-------------------- Display image ---------------------

        for(j=1;j<=256;j++){ for(i=1;i<=256;i++){

            count = img[i][j];

                count *= 255 / maxcount;

                        if(count < 0   )  count = 0  ;
                        if(count > 255 ) count = 255;

                SetColor( RGB(count, count, count) );

                SetPixel(i,j);
  }}
```

```
//------------- 3x3 smoothing filtering ---------------------

printf("¥n¥n 3x3 smoothing filter "); scanf("%c",&yn);


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

  img2[ i ][ j ]  =  img[ i-1 ][ j-1 ] + img[ i ][ j-1 ]   + img[ i+1 ][ j-1 ]

                   + img[ i-1 ][ j ]    + img[ i ][ j ] * 2  + img[ i+1 ][ j-1 ]

                   + img[ i-1 ][ j+1 ] + img[ i ][ j+1 ]   + img[ i+1 ][ j+1 ] ;
} }

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

        img2[ i ][ j ] /= 10 ;
} }
```

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

img2[ i ][ j ]  =  img[ i-1 ][ j-1 ] + img[ i ][ j-1 ]   + img[ i+1 ][ j-1 ]

                + img[ i-1 ][ j ]    + img[ i ][ j ] * 2  + img[ i+1 ][ j-1 ]

                + img[ i-1 ][ j+1 ] + img[ i ][ j+1 ]  +img[ i+1 ][ j+1 ] ;
} }
```

```
for( j=2; j<=255; j++ ) { for( i=2; i<=255; i++ ) {
        img2[ i ][ j ] /= 10 ;
} }
```

img[i][j] に 3x3 smoothing フィルタをかけて、img2[i][j] に代入している。



フィルタをかけるときの for ループの変数 i, j が、

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

で、i、j が 2 から 255 までになっていることに注目。

ループ内で img[i-1][j-1] や img[i+1][j+1] があるが、変数 i , j に 1 や 256 が代入されてしまうと img[0][0] や img[257][257] には数字が入っていないので誤った計算をしてしまう。

printf(″¥n¥n 3x3smoothing filter″); scanf(″%c″,&yn);

scanf で 入力された文字 yn はプログラムでは利用しないが、この文でプログラムが入力待ちの状態になる。
プログラム実行を途中で一旦停止させたいときに使う文。

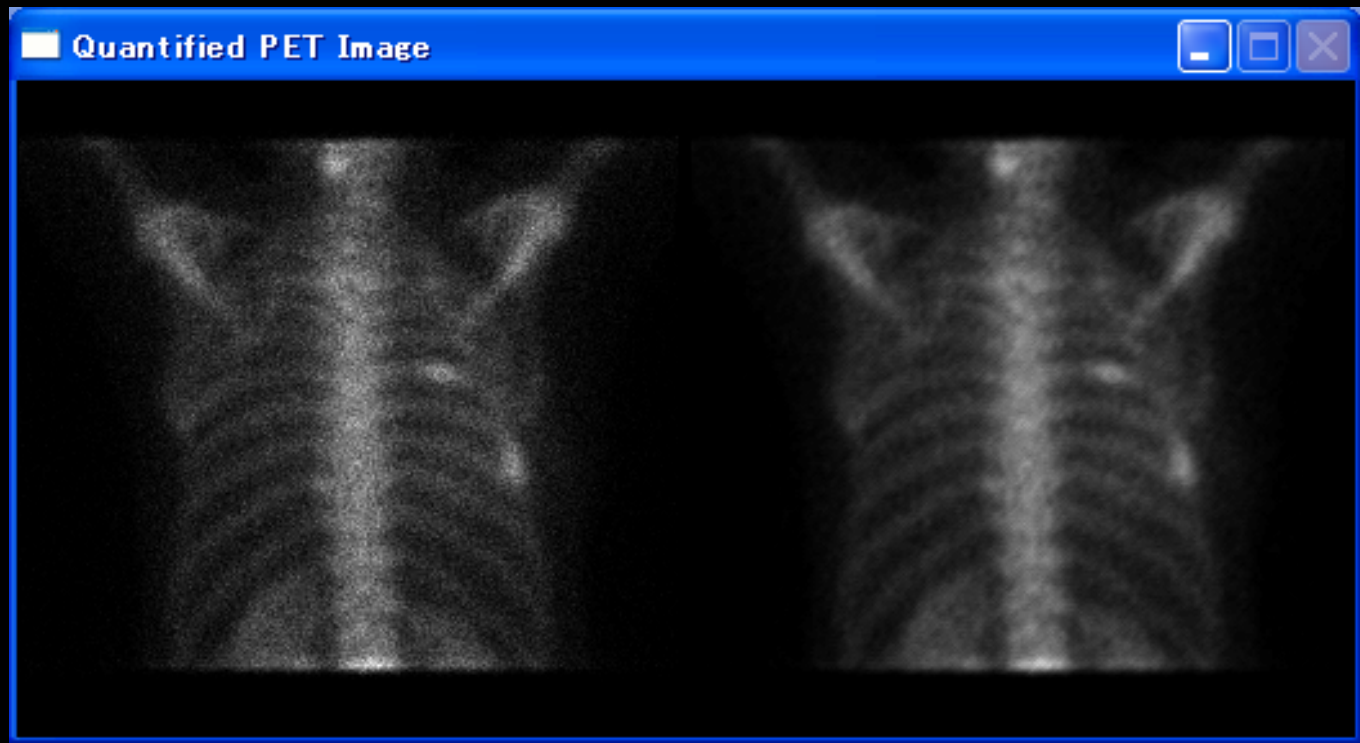
```
//-------------- Display filtered image --------------------
   for( j=2; j<=255; j++ ){ for( i=2; i<=255; i++ ){

        count = img2[ i ][ j ];     count *= ( 255 / maxcount );

        if(count < 0 ) count = 0 ;    if(count > 255 ) count = 255 ;

        SetColor( RGB(count, count, count) );

        SetPixel( i + xshift , j );
   } }

//--------------------- End ----------------------------

        printf("¥n¥n Retry? (no:n) ");scanf("%c",&yn);
        if(yn!='n')goto START;
        exit(0);
}
```

SetPixel(i + xshift , j);

img2[i][j] の描画部位を、img[i][j] より

右側に xshift 画素シフトさせている。

整数変数 xshift の値は、宣言時に 260 が代入されているので、260画素分だけ右側に描画される。



```
// smoothing2.c      平滑化フィルタ処理関数の作成
#include <HU.h>
void smoothing ( int x[ ][ 260 ],  int y[ ][ 260 ] )
{
        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 ] * 2+ 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 ] /= 10 ; } }
}

void Main(void)
{
```

関数を作る

```
//---------- 3x3 smoothing filter ----------------
printf("¥n¥n 3x3 smoothing filter "); scanf("%c",&yn);
smoothing( img , img2 );
//------------------------------------------------
```

smoothing関数を作って、プログラムを見やすくする。

ひとまとまりのプログラムを、ひとつの関数にして
main関数（ここでは Main(void) ）の前に置く。

main関数内に、プログラム文を全て記述していくと
非常に長く、みにくいプログラムになり、
プログラムミスの発見も困難になる。

こまめに関数を作って、main関数内のプログラム文を
減らしてすっきりさせるのが正しい Cプログラム記述法。

どのような変数を持つ関数にするかは自由である。

ここでは、関数名を smoothing として、
関数に渡す変数は 配列 img[260][260] に記録された画像データ と、処理画像をしまう配列 img2[260][260] とした。

大量の画素データを、ひとつづつ関数に渡すわけにはいかないので、画像データが記録されているメモリ上の先頭アドレス（ポインタ）を関数に渡す。

関数は、受け取ったポインタの示すアドレスからメモリ上のデータを読み込むと、それが 画像データになっているので、ポインタを受取るだけで画像を読み込める。

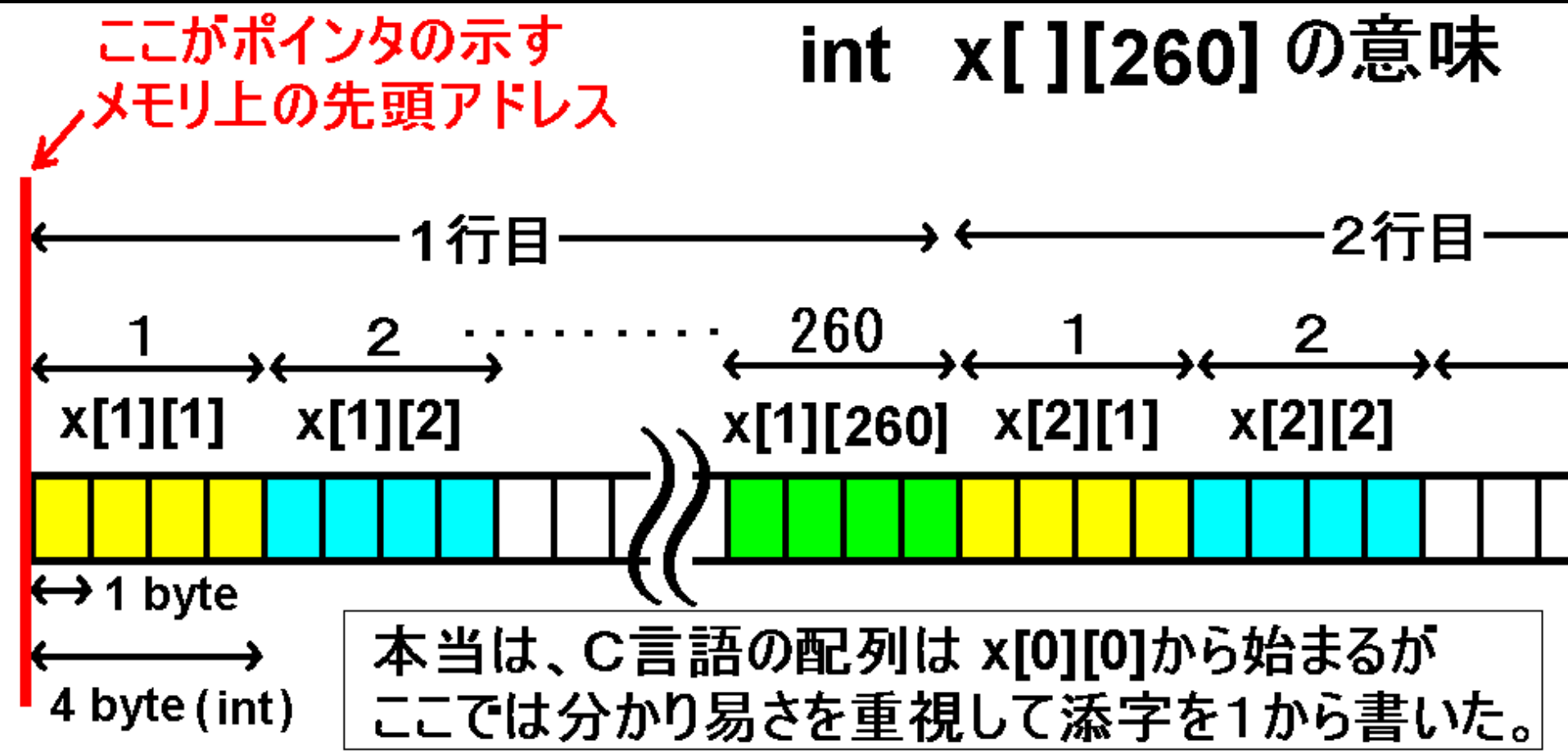
しかし、画像のような2次元配列では、メモリ上の先頭アドレスだけでは、１画素が何バイトなのか、横幅が何ピクセルあるのかが判らないと困る。

そこで、関数側で受取る変数の型を記述する所で、まず１画素が4byteの整数であることを示すint を明記する。次に1行が260画素の2次元配列であることを明記するために x[ ][260] と記述する。

前半の[ ] 内を空白にすることで、2次元配列 x[ ][260] という記述は、配列の内容を具体的に示すものではなく、単に配列の名前を示しているのと同じ扱いになる （ 配列名 ＝ 配列の内容が記録されているメモリ上の先頭アドレス（ポインタ））

配列 の内容が記録されているメモリ上の先頭アドレス（ポインタ）と、１画素のデータ長（ここでは4バイト）と、配列 の１行の長さが判れば、関数側はそれで十分。

配列名と、列が何列あるかは、関数は知る必要がない。



関数 smoothing で注目すべき点は、
この関数の中では配列を宣言していないことである。

main関数の中で宣言した配列 img、img2 を
間借りしているだけである。

main関数が用意したメモリ上の数値を勝手に使って
計算を行っている。

C言語の関数は、このようなことが出来るため、
メモリ上の数値の行き来が他言語より少なく、
処理速度の速いプログラムが作成でき、
最も普及する言語になった。

ポインタの扱い方がC言語の最も重要な点である。

main関数の中で　smoothing(img , img2);　と書いて
smoothing関数に、1行 260 ピクセルの2次元配列
img と img2 のメモリ上の先頭アドレス(ポインタ)を渡す。

関数側は、smoothing (int x[ ][260],　int y[ ][260])
と書いて、メモリ上では img、img2 と全く同じ配列 を
x、y という名前で、1行 260 ピクセルで、1画素4バイト
の2次元配列として宣言しているが、

1行260ピクセルの2次元配列 x、y の先頭アドレスを
img、img2 と同じにすると宣言しているだけで、
新たな配列は作っていない。

したがって、関数側で用意する配列などの変数名は
メモリ上の数値を受け渡しするためだけのものなので、
main関数で使っていた変数名とは別名でも構わない。

smoothing関数の中で、画像 x を平滑化処理した画像を配列 y に書き込んでいるが、

配列 y の メモリ上の先頭アドレスは img2 と同じなので、この関数が実行された後では、main関数内で img2 の内容を表示すると平滑化処理された画像が現れる。

プログラム文の中では異なる名前の変数だが、
メモリ上での**実体が同じ（変数の型とポインタが同じ）ものを、エイリアス（alias）**という。

エイリアスを扱う鍵が、ポインタ。

エイリアスをよく理解し、上手に使えるようになると
C言語が急に簡単に扱えるようになる。

関数の記述法

main関数内では

　関数名(引数1 , 引数2 , ・・・・) ; と書く。

引数(ひきすう　argument)とは、関数と受け渡しする変数のこと。引数の個数は自由。

関数側では、main関数の上に

```
出力変数の型　関数名( 引数1の型　引数1のエイリアス名、・・　)
{
　　関数のプログラム文
}
```

と書いて、main関数で宣言した引数と同じ型を記述する。
main関数側の引数の型や個数が異なるとエラーになる。

出力値をもつ関数を作ったら、その値の型を関数名の左側に記述する。　出力値の無い関数では、void と書く。

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]

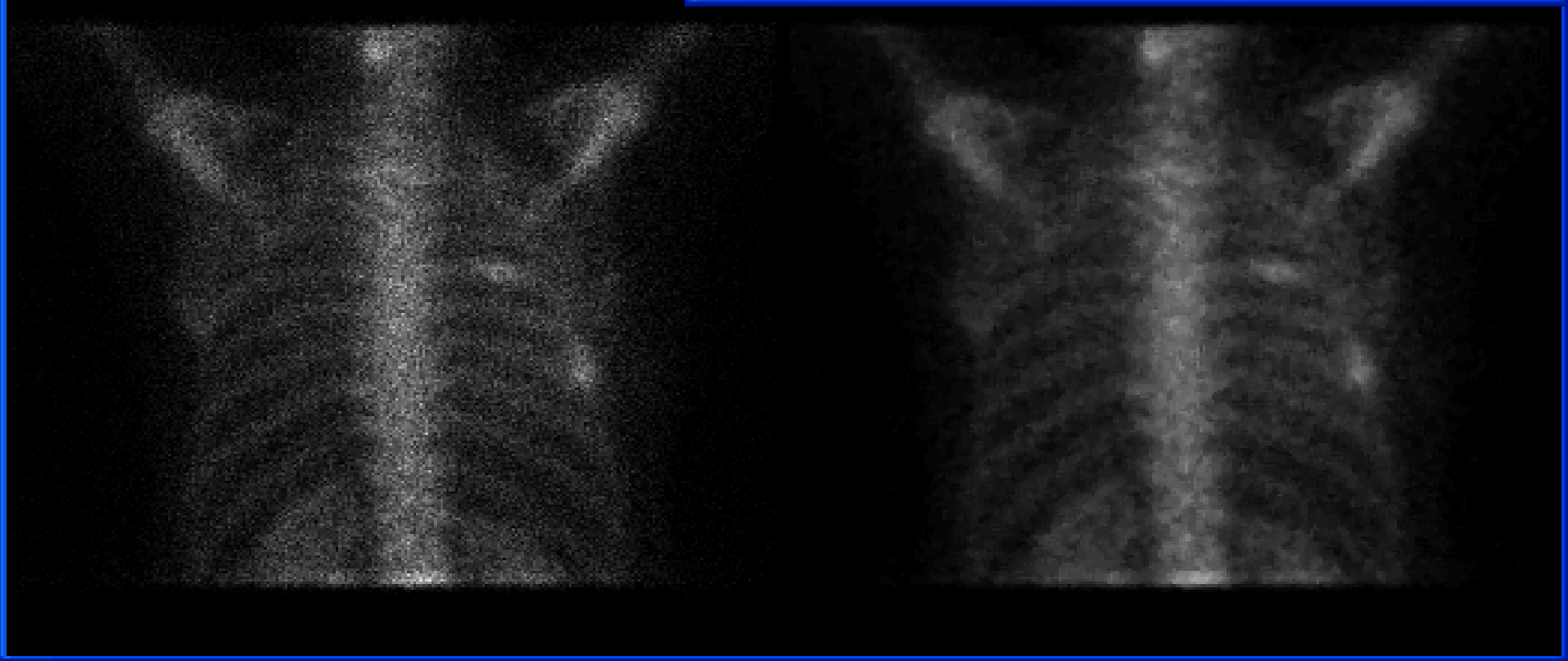
座標[i][j]の画素値を、その近傍を含む９画素の値 med[1]～med[9]の中央値に置き換える処理。

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。

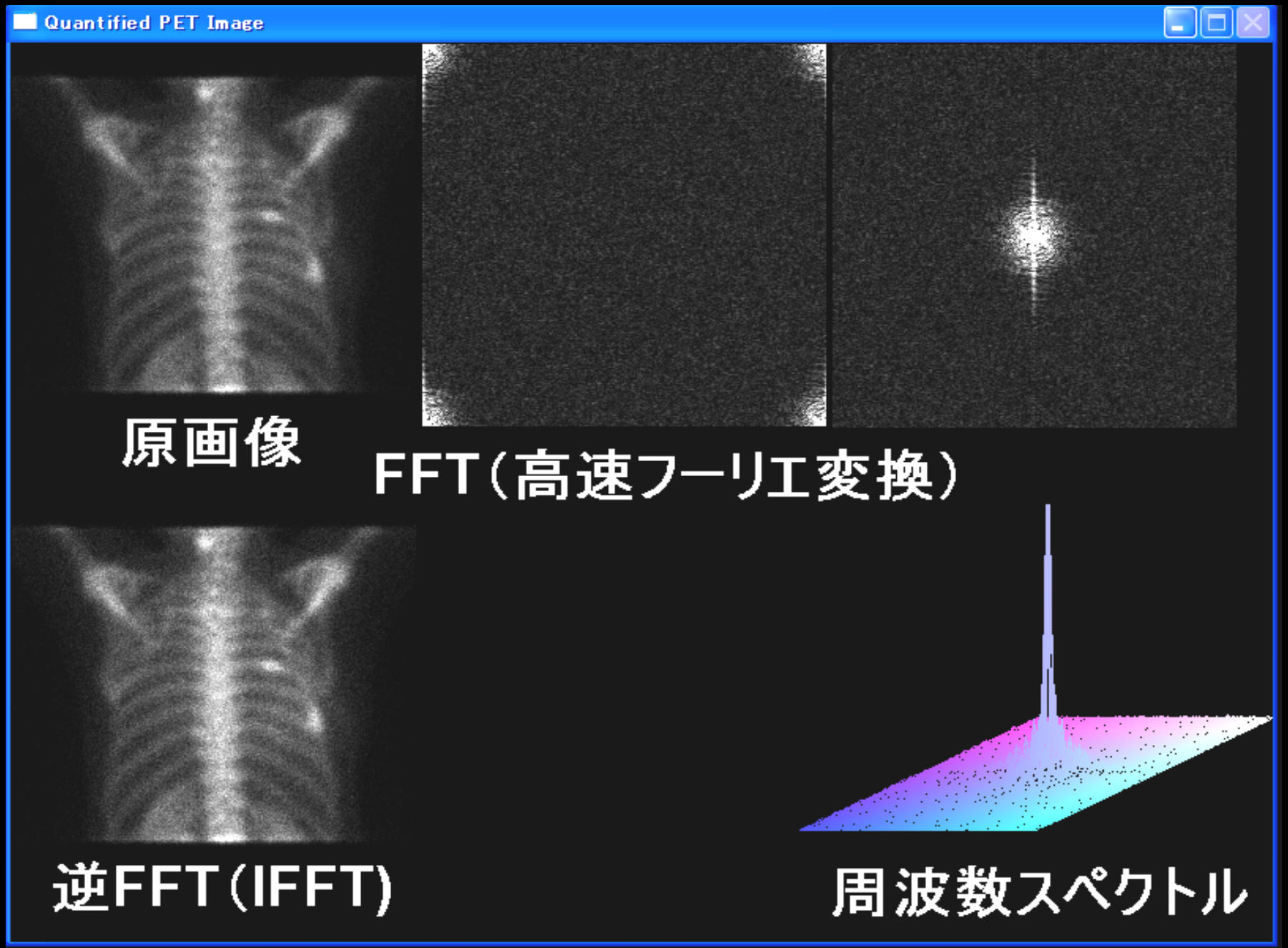
median filter は、微細なノイズを除去する効果がある。

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。

画像を フーリエ変換する。　FFT.c



2次元高速フーリエ変換（FFT : Fast Fourier Transform）

```
void FFT2D ( float x[ ] , float y[ ] , int data_n , int invert )
{
        // Numerical Recipes in C (p440-442) より引用（一部変更）。

        // invert が 1 ならば、画像x を2次元フーリエ変換して
        // 結果を実数成分x , 虚数成分y の配列に入れる。
        // data_n は、画像の一辺の画素数（2の階乗の必要あり）。

        // invert が -1 ならば、実数成分x , 虚数成分yを
        // 2次元逆フーリエ変換して、結果を画像x の配列に入れる。

        int       i,idim,i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
        int       ibit,k1,k2,n,nprev,nrem,dimension;
        float     tempi,tempr,temp;
        double theta,wi,wpi,wpr,wr,wtemp, PI = 3.141592653589793 ;

        float     *data;

        data = calloc( (data_n+1)*(data_n+1) * 2, sizeof(float) ) ;
```

```
float  *data;
data = calloc( (data_n+1)*(data_n+1) * 2, sizeof(float) ) ;
```

動的メモリ 宣言。 C言語はプログラム中で 配列の発生、消去ができ、配列の大きさを変数で宣言できる。

関数 FFT2Dは、内部で1次元配列 data (各要素の型はfloatで、(data_n +1) * (data_n +1) * 2 個の要素が必要) を宣言するが、単純に float data[(data_n + 1)*(data_n +1)*2]; と書くとエラーになる。

普通の配列宣言では要素数に変数を書くことはできない。

そこで変数宣言では、とりあえず配列名 data だけ宣言し、(配列名はポインタなので、float *data ; と宣言)

calloc関数を使って、先頭アドレスを dataとして、必要な配列のメモリ領域を確保している。これは配列の宣言と同じこと。

calloc関数の文法

配列名 = calloc(要素数、sizeof (変数の型)) ;

calloc関数は出力値をもつ。確保したメモリ領域の先頭アドレスを出力するので、配列名 = calloc と書くことで、配列のポインタに、確保したメモリ領域先頭アドレスが代入される。

callocで確保したメモリ領域は 普通の配列と同等に扱える。プログラム中で data[i] などと記述できる。

使用済みの動的メモリは、メモリ領域の無駄使いを避けるために、最後に free(配列名) ; で開放する必要がある。

FFT2D関数の最後のプログラム文は　free(data) ;
と書かれていることを確認して下さい。

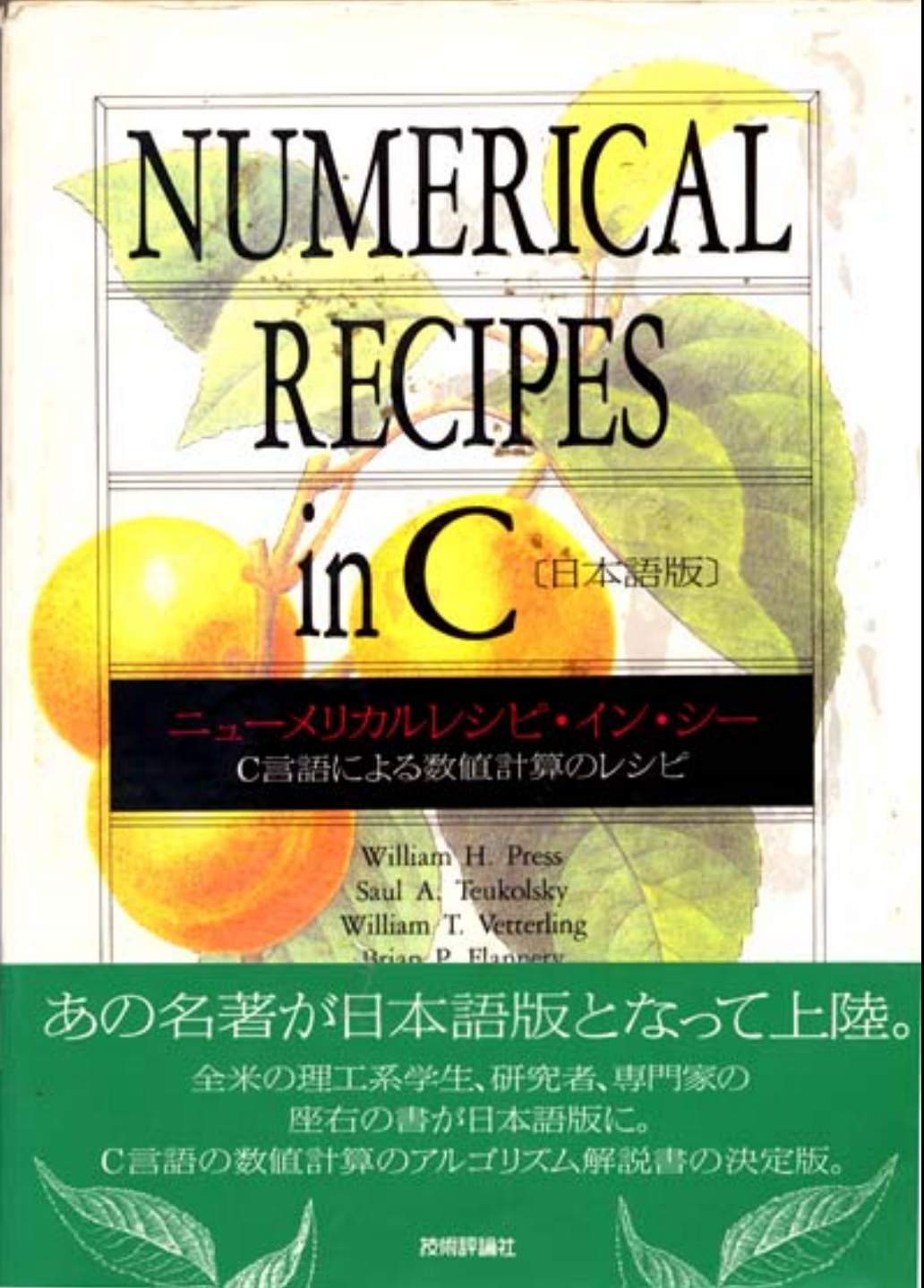
```
for (i=1; i<=data_n * data_n ; i++ ){ data[2*i-1]=x[i];   data[2*i]=y[i];    }
nprev=1;

for(idim=dimension; idim>=1; idim--){ Event95();
                    n = data_n;
                    nrem = data_n * data_n / (n*nprev) ;
                    ip1 = nprev << 1;
                    ip2 = ip1 * n;
                    ip3 = ip2 * nrem;
                    i2rev = 1;

                    for(i2=1; i2<=ip2; i2+=ip1){

                                        if(i2 < i2rev){

                                                            for( i1=i2; i1<=i2+ip1-2; i1+=2 ){

                                                                                for( i3 =i1; i3 <= ip3; i3 += ip2 ){

                                                                                                    i3rev = i2rev +i3 -i2;
                                                                                                    SWAP(data[i3],data[i3rev]);
                                                                                                    SWAP(data[i3+1],data[i3rev+1]);

                                                                                }
                                                            }
                                        }

                                        ibit = ip2 >> 1;

                                        while( ibit >=ip1 && i2rev > ibit ){

                                                            i2rev -= ibit;
                                                            ibit >>= 1;
                                        }

                                        i2rev += ibit;
                    }

                    ifp1 = ip1 ;

                    while( ifp1 < ip2){

                                        ifp2 = ifp1 << 1;
                                        theta = invert * 2. * PI / (ifp2 / ip1);
                                        wtemp = sin(0.5*theta);
                                        wpr = -2.0 * wtemp * wtemp;
                                        wpi = sin(theta);
                                        wr = 1.0; wi = 0.0;

                    // Danielson-Lanczos のFFT アルゴリズム

                                        for( i3=1; i3<=ifp1; i3+=ip1 ){

                                                            for( i1=i3; i1<=i3+ip1-2; i1+=2 ){

                                                                                for( i2=i1; i2<=ip3; i2+=ifp2){

                                                                                                    k1 = i2;
                                                                                                    k2 = k1+ifp1;
                                                                                                    tempr = (float)wr*data[k2]-(float)wi*data[k2+1];
                                                                                                    tempi = (float)wr*data[k2+1]+(float)wi*data[k2];
                                                                                                    data[k2] = data[k1]-tempr;
                                                                                                    data[k2+1] = data[k1+1]-tempi;
                                                                                                    data[k1] += tempr;
                                                                                                    data[k1+1] += tempi;
```

Danielson-Lanczos の FFT アルゴリズム

非常に難解な

プログラムコード。

理解できたら天才！



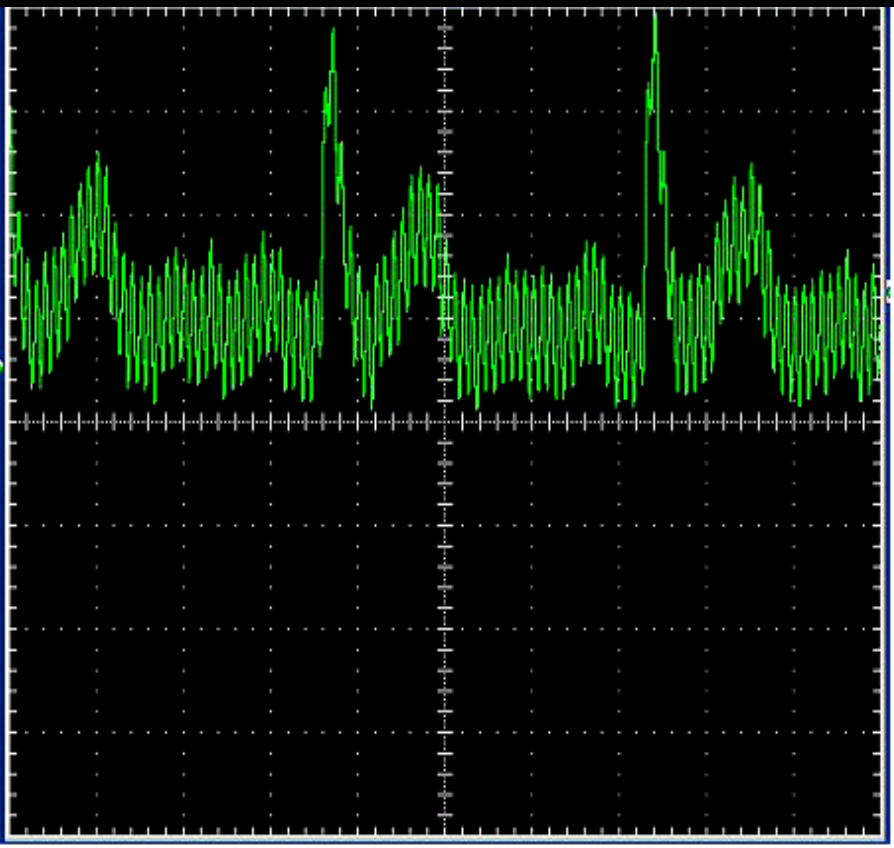
NUMERICAL RECIPES in C

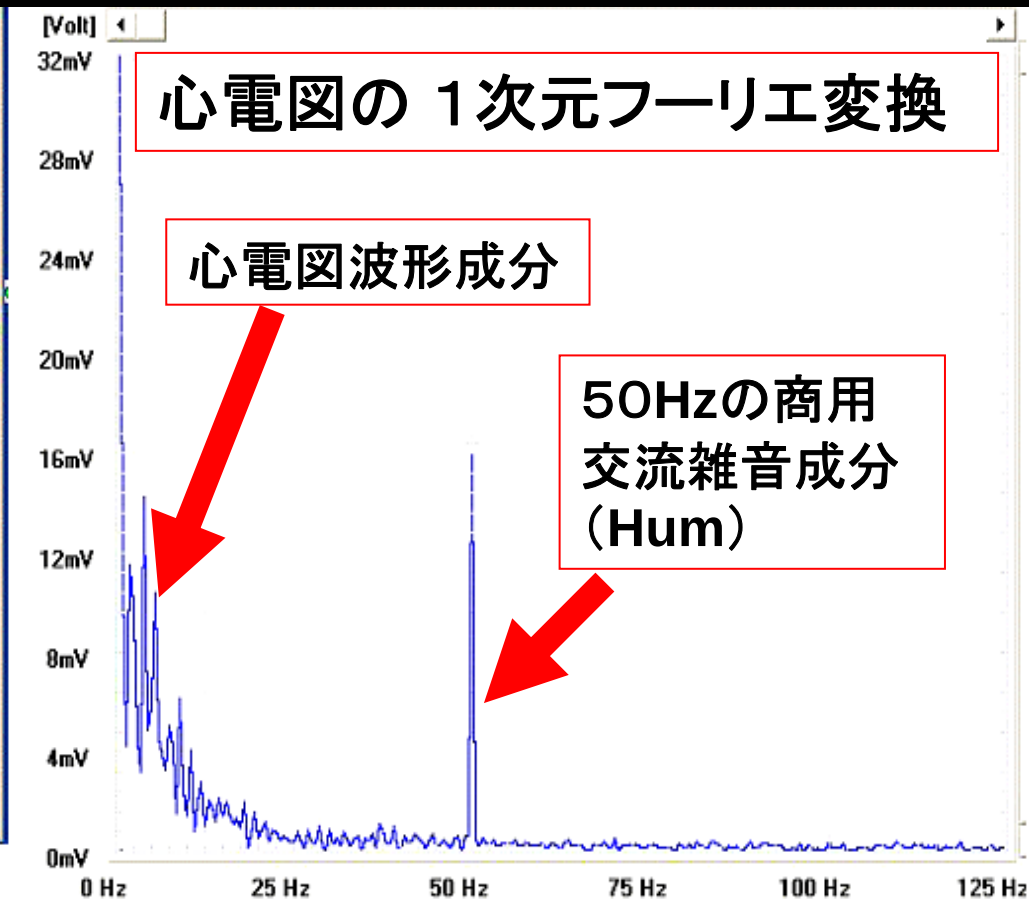
技術評論社

科学技術計算に有用な
高速フーリエ変換や
最小二乗法などの関数が
C言語プログラム として
多数収録され、詳細な
解説も記述されている。

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 $e^{(2\pi ft)i}$ 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 ----------------------------

for( j=0; j<=255; j++ ){ for( i=1; i<=256; i++ ){

        x[ i + 256*j ] = (float) img[ i][ j+1] ;  y[ i + 256*j ] = 0.0 ;
}}

FFT2D( x, y, 256, 1 );

for( j=0; j<=255; j++){ for( i=1; i<=256; i++){

imgFFT[i][j+1] = sqrt( x[i+256*j]*x[i+256*j] + y[i+256*j]*y[i+256*j]);

}}
```

main関数側の 関数 FFT2D の呼出し。
3番目の引数は、画像の1辺のピクセル数、
4番目の引数は、1 を入れているので、FFTを実行。

```
x[ i + 256*j ] = (float) img[ i][ j+1] ;  y[ i + 256*j ] = 0.0 ;
```

関数 FFT2D の 1、2番目の引数は、float x[ ]、float y[ ] と指定しているので、画像データをfloat型にキャストして１次元配列に並べ換える必要がある。
配列 x は実数成分データ。FFTでは 画像を入れる。
配列 y は虚数成分データ。FFTでは 全て 0 を入れる。

FFT2D(x, y, 256, 1); が終了すると、
配列 x には周波数スペクトルの sin成分 が入っている。
配列 y には周波数スペクトルの cos成分 が入っている。

```
imgFFT[ i][j+1] = sqrt( x[i+256*j]*x[i+256*j] + y[i+256*j]*y[i+256*j]);
```

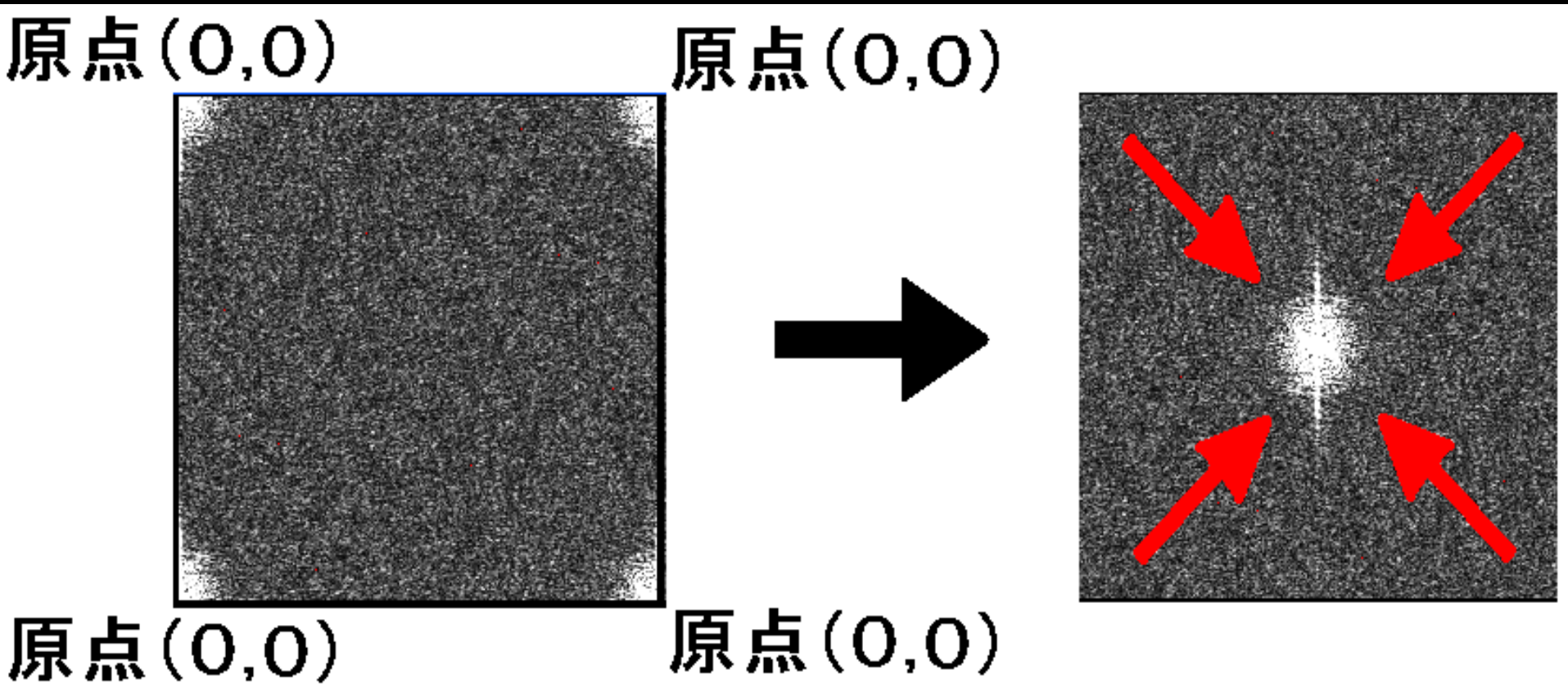
配列 imgFFT には、sin、cos成分の2乗和の平方根（スペクトルの大きさになる）を入れている。
sqrt() は、平方根を計算する関数。

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。



```
//------------------- Display FFT spectrum ---------------
    for(j=1;j<=128;j++){ for(i=1;i<=128;i++){

        g[i][j] = imgFFT[129-i][129-j];
        g[i+128][j] = imgFFT[257-i][129-j];
        g[i][j+128] = imgFFT[129-i][257-j];
        g[i+128][j+128] = imgFFT[257-i][257-j];
    }}

      for(j=1;j<=256;j++){ for(i=1;i<=256;i++){

        countFFT = g[i][j];

        countFFT *= (255. / maxcountFFT);  count = (int)countFFT;

              if(count < 0) count=0;  if(count > 255) count=255;

        SetColor95( RGB(count, count, count) );

        SetPixel95(i+520,j);
  }}
```

4隅の原点を
中央に折り返し、
配列 g[ ][ ] に
入れ直している。

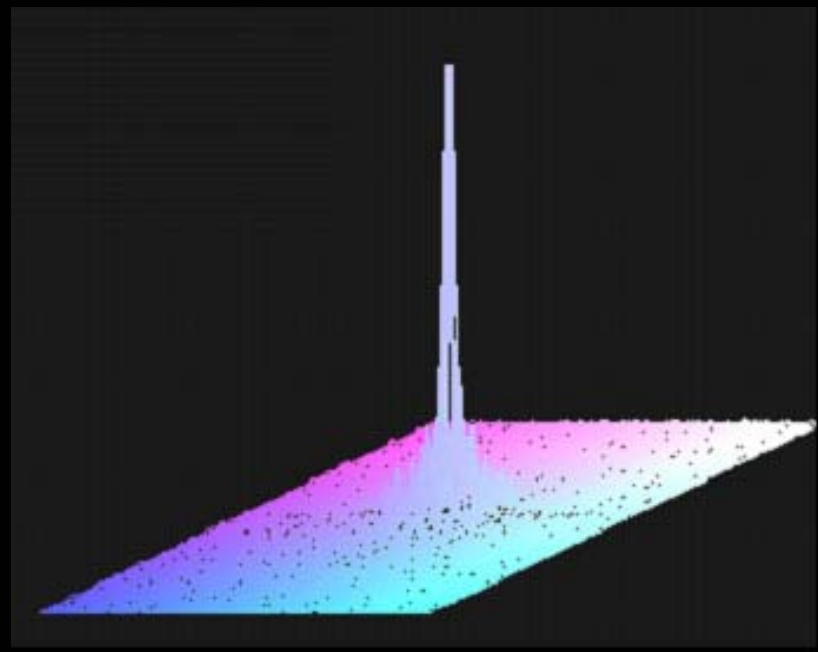
```
//--------------- Display FFT spectrum curve -------------
for( j=50; j<=200; j++ ) { for( i=50; i<=200; i++ ) {
countFFT = g[ i ] [ j ]  *(255./maxcountFFT); count = (int)countFFT;
countFFT = g[ i+1][ j ] *(255./maxcountFFT); count2= (int)countFFT;
 ip = 420 + i + j ;                    ip2 = 420 + i+1 + j;
 jp = 550 - count/200 - /2 ;      jp2 = 550 - count2/200 - (i+1)/2 ;
SetColor( RGB( i, j, 255 ) ) ; Line( ip, jp, ip2, jp2 ) ;
}}
```

Line(x1, y1, x2 , y2);

座標(x1, y1)から(x2, y2)に

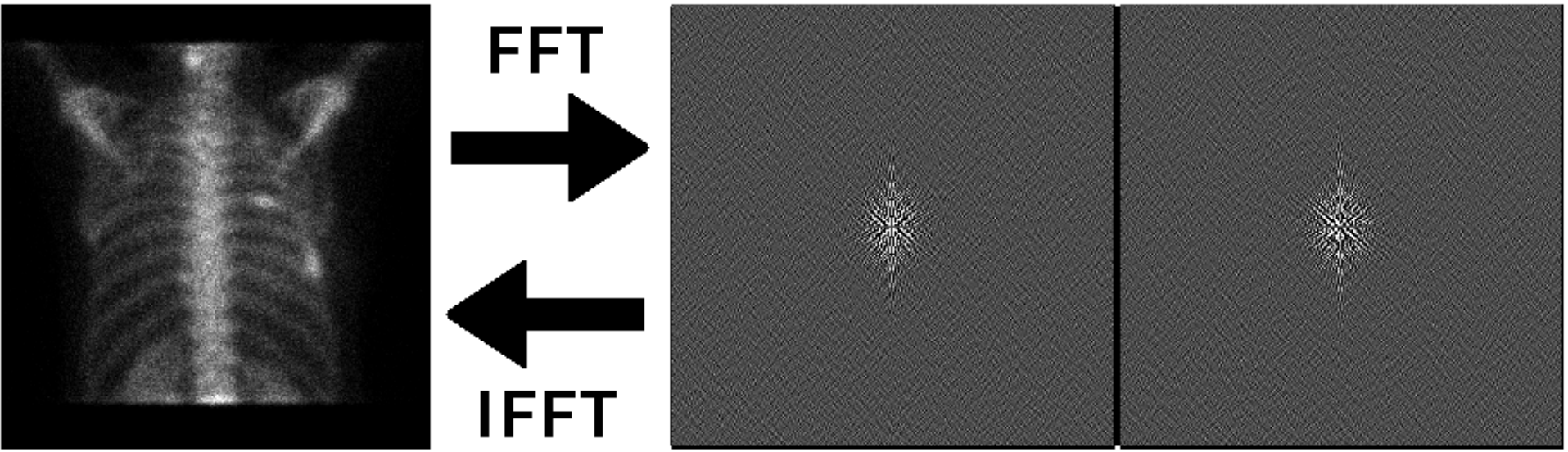
SetColor で設定した色の

線を描く関数。



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。



```
//---------------------- IFFT ----------------------
  printf("¥n¥n Perform IFFT (OK: enter) ");scanf("%c",&yn);

  FFT2D( x, y, 256, -1 ) ;

  for( j=0; j<=255; j++ ) { for( i=1; i<=256; i++ ) {
            imgIFFT[ i ][ j+1] = x[ i+256*j ] ;
  } }
```

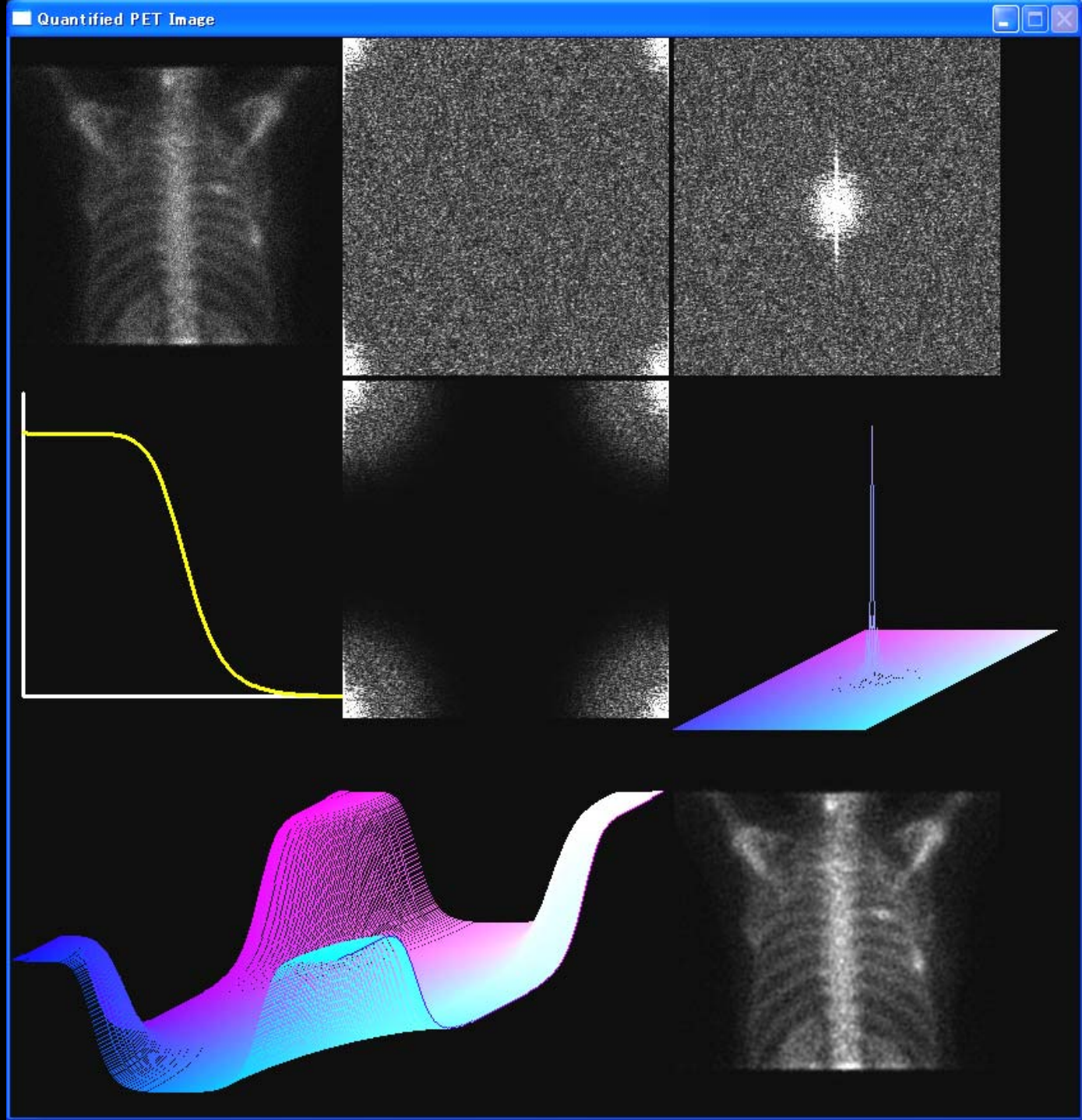
関数 FFT2D の４番目の変数を -1 にすると、
２次元 逆FFT を行う関数に変わる。

配列 x[ ] に実空間に戻った画像が１次元配列 x[ ] に入っているので、２次元配列 imgIFFT[ ][ ] に入れ換えている。

Butterworth.c

周波数空間で
フィルタ処理を行う

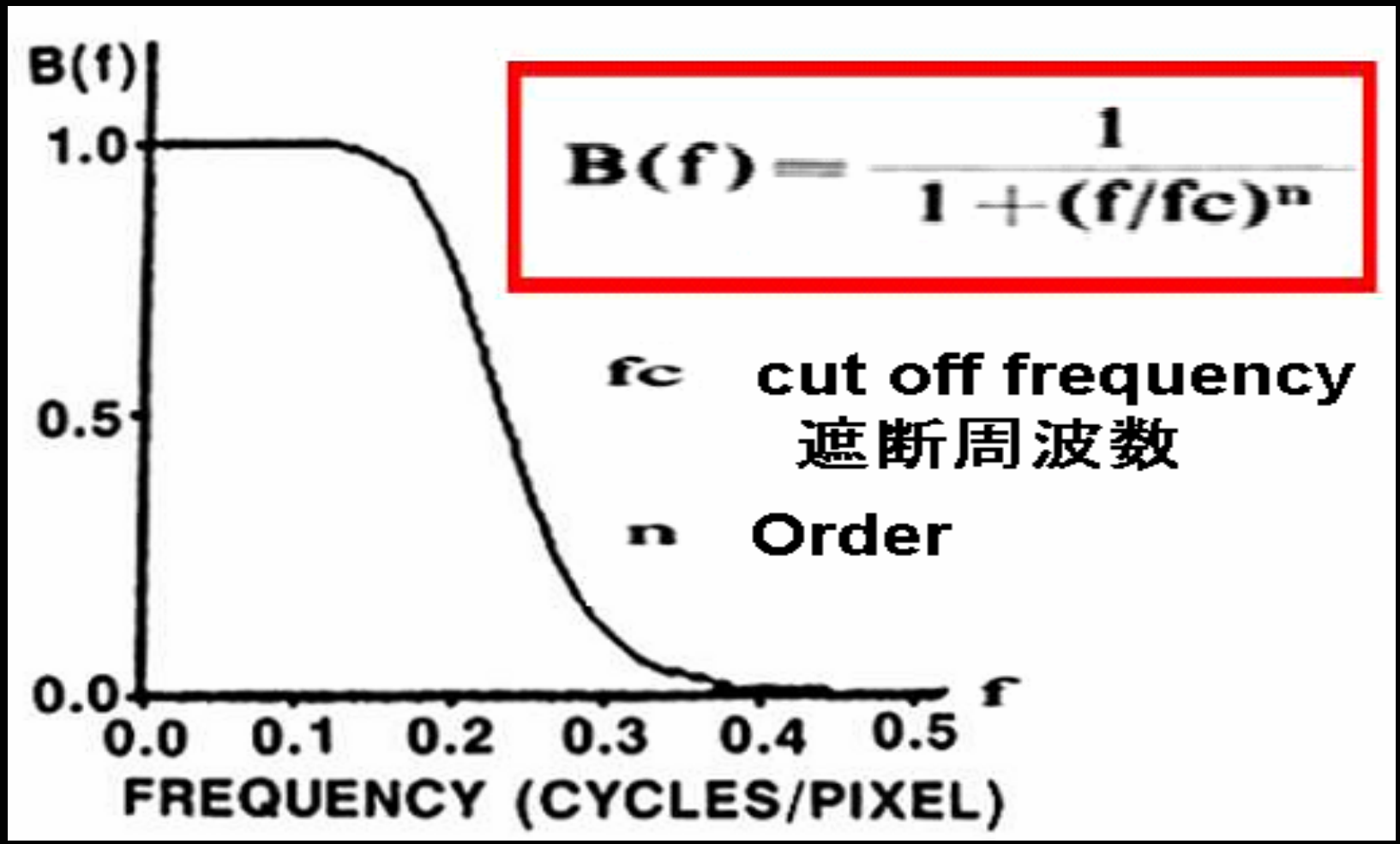
バターワース
フィルタ



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。



```
//---------------- Butterworth filter ----------------
SET_CUTOFF:
printf("¥n¥n Cutoff Freq.(0～0.5) = "); scanf("%lf",&cutoff);
for( i=0; i<=200; i++ ){
  Bu[i] = 1.0 / ( 1.0 + pow( ((double)i / (cutoff*256.0)), 9.0));
}
```

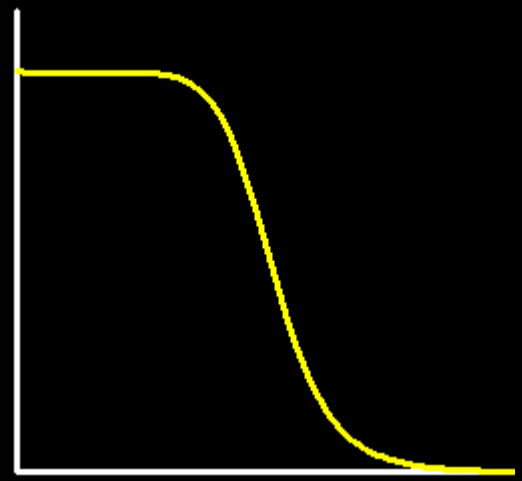
Order を 9.0 に設定した Butterworth filter を配列 Bu[ ] に入れている。

周波数の単位は cycle／pixel になる。

遮断周波数の最大値は、ナイキスト周波数 0.5

関数 pow(X , Y) は X^Y を算出する関数。

```
//------------ Disp 1D-Butterworth filter ----------

    SetColor(BLACK); Rect(10, 300, 266,500, 1);

    PenWidth(3);  SetColor(WHITE);

    Line(10, 500, 10, 270);  Line(10, 500, 266, 500);

    SetColor(YELLOW);

    for( i=0; i<=128; i++ ){

      ip = 10 + i*2 ;  ip2 = 10 + (i+1)*2 ;

      jp  = 500 - (int)( Bu[i]  * 200.0 );
      jp2 = 500 - (int)( Bu[i+1]* 200.0 );

      Line( ip, jp, ip2, jp2 );
    }
    PenWidth(1);
```



Rect(X1, Y1, X2, Y2 , 1) ;

座標(X1, Y1)と(X2, Y2)を対角とする四角を描く。

5番目の変数が1なら中を塗り潰す。0なら輪郭のみ。

このプログラムでは、黒い四角形を塗り潰して、

再描画の際にグラフィック画面を部分的にクリアしている。

PenWidth(3) ;

描画する点や線の太さをピクセル単位で設定できる。

カッコ内を3にすると、太さ3ピクセルの描画を行う。

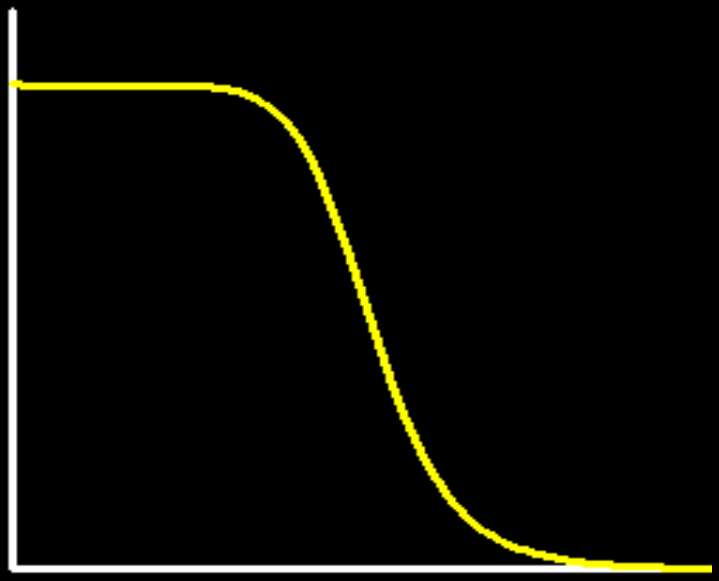
目的とする描画が終了したら PenWidth(1) ;

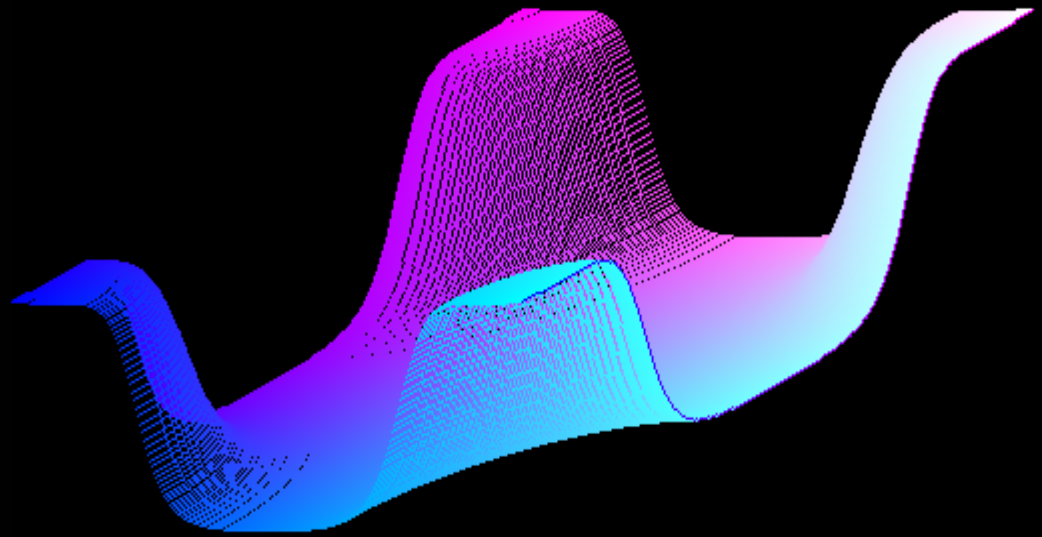
と書いて1ピクセルに戻すのが無難。

```
//------------------ 2D-Butterworth filter --------------------

for( j=1; j<=128; j++){ for( i=1; i<=128; i++ ){

  r = sqrt( (double)( i*i + j*j ) ) ; B2[ i ][ j ] = Bu[ (int)r ];
}}

for( j=1; j<=128; j++){ for( i=129; i<=256; i++){

  r = sqrt( (double)( (257-i)*(257-i) + j*j )); B2[ i ][ j ] = Bu[ (int)r ];
}}

for(j=129;j<=256;j++){ for(i=1;i<=128;i++){

  r = sqrt((double)(i*i + (257-j)*(257-j))); B2[i][j] = Bu[ (int)r ];
}}

for(j=129;j<=256;j++){ for(i=129;i<=256;i++){

 r=sqrt((double)((257-i)*(257-i)+(257-j)*(257-j)));B2[i][j] = Bu[ (int)r ];
}}
```

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。





```
//---------- 2D-Butterworth filtering ----------

for( j=1; j<=256; j++ ){ for( i=1; i<=256; i++ ){

        Bx[ i+256*(j-1) ] = x[ i+256*(j-1) ] * B2[ i ][ j ];

        By[ i+256*(j-1) ] = y[ i+256*(j-1) ] * B2[ i ][ j ];
}}
```

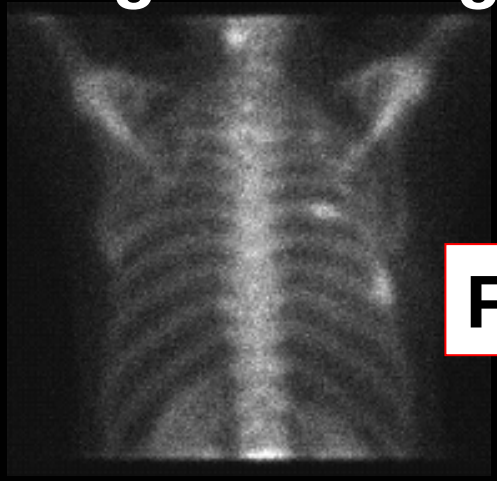
２次元周波数分布のsin成分 x[ ]、cos成分 y[ ] に２次元 Butterworth filter B2[ ][ ] をかけている。

x[ ] と y[ ] は、２次元データが１次元配列に入っているので、上記プログラムのようにして、フィルタ処理された sin、cos成分を １次元配列 Bx[ ]、By[ ] に入れている。

```
//-------------------- IFFT ------------------------
printf("¥n¥n Perform IFFT (OK: enter) "); scanf("%c",&yn);

  FFT2D( Bx, By, 256, -1 );

for( j=0; j<=255; j++ ) { for( i=1; i<=256; i++ ) {
  imgIFFT[ i ][ j+1 ] = Bx[ i+256*j ];
} }
```

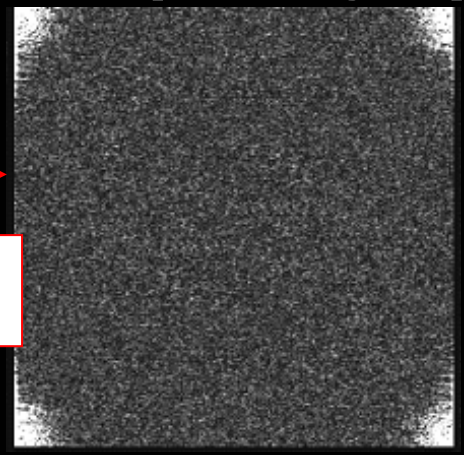
Bx[ ]、By[ ] を FFT2D関数で ２次元逆フーリエ変換すると、Bx[ ] に周波数空間処理が行われた実空間画像データが入っている。

Original Image

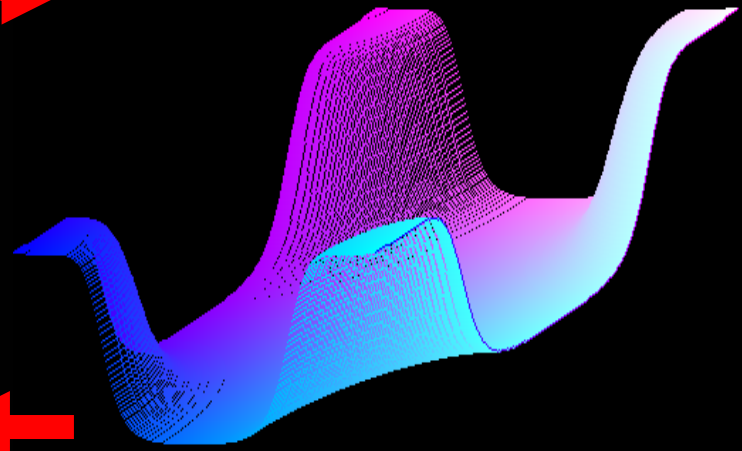


FFT

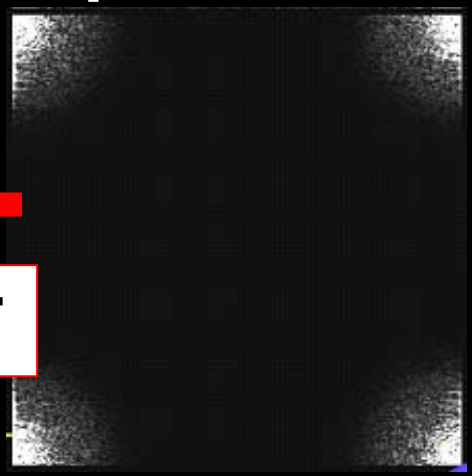
Frequency spectrum



Frequency space filtering

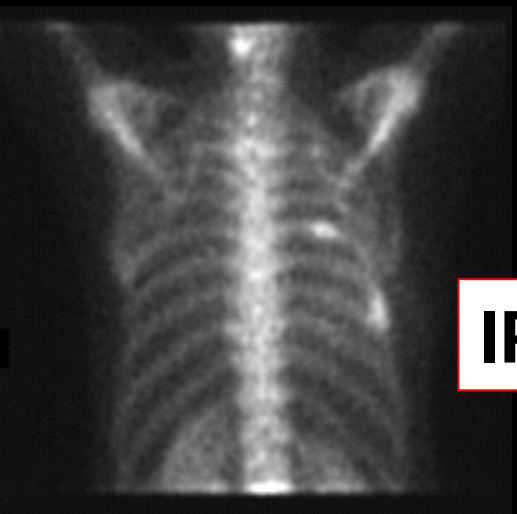


Filtered Frequency spectrum



IFFT

Filtered Image



課題4　smoothing2.cを改良して、下記の5種類の空間フィルタリングを行うプログラムを作成して画像bone6に行った処理画像を送ってください。

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