平滑化処理プログラム smoothing2.c において、

main関数の中で smoothing (img , img2); と書き、
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 という配列名として宣言している。

このとき、smoothing関数側は、配列x、y の１行の長さは
260 だと知ることが出来るが、列が何列あるか知ることは
できない。
しかし、この関数は x、y の列数を知る必要がない。
その理由を記述して下さい。

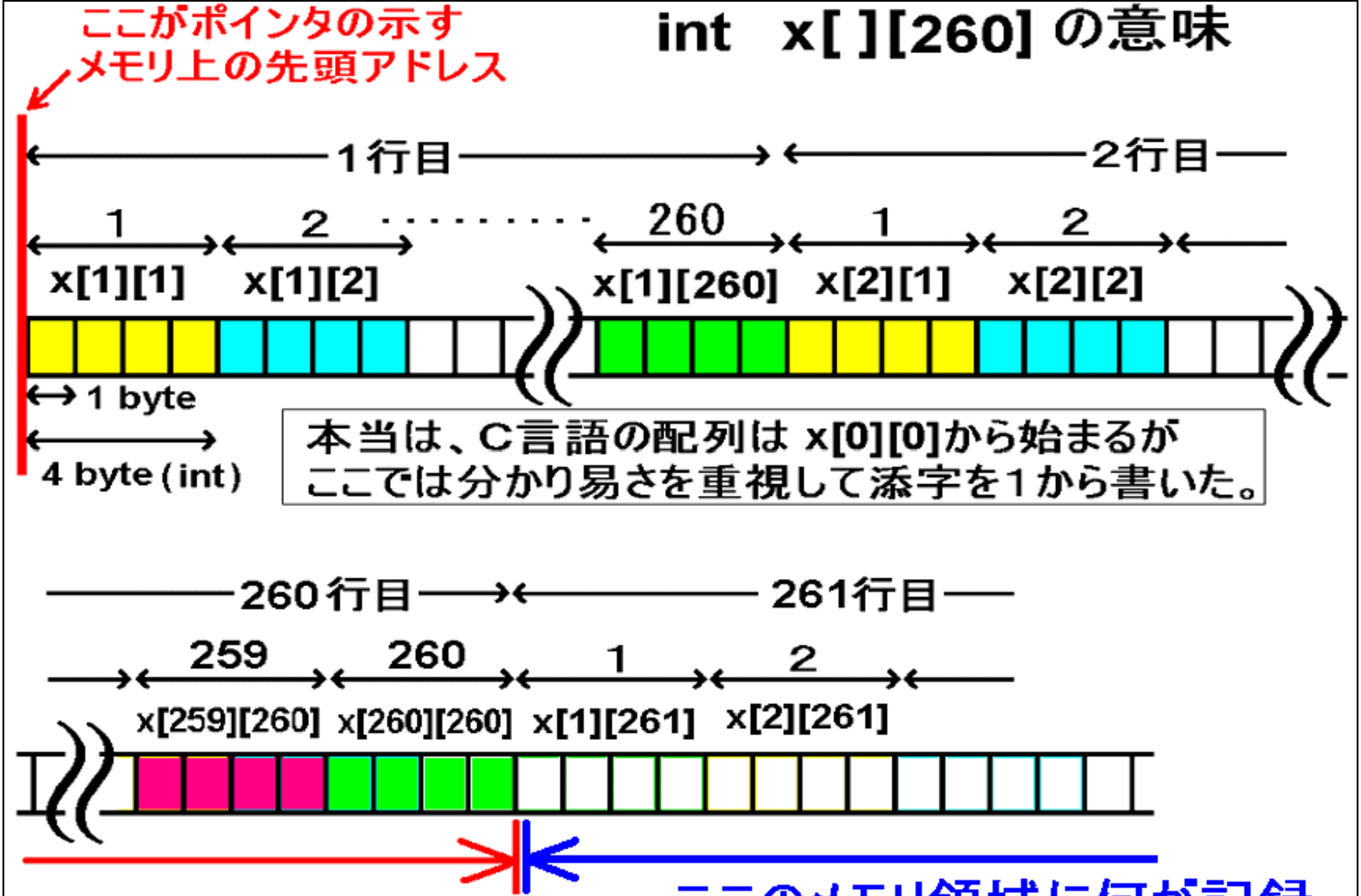
正解と考えられる解答例：

ポインタの示すアドレスから1ピクセル4バイトずつ
260ピクセルのデータを順番に切り出していくだけなので
関数は列数を知らなくてもよい。

与えられたデータをだだ忠実に並べていくだけなので
関数自体は、いつ終わるのかは判らなくても良い。

配列 の1行の長さを決定することができれば、
それを基にして次の列を作ることができるので、
列が何列あるか知らなくても良い。
1行の長ささえわかれば、その長さでどんどん折り返す
ことができ、値がある分だけ繰り返し続ければ良いので、
何列あるかというのは重要ではない。

int x[ ][260] の意味


ここがポインタの示す
メモリ上の先頭アドレス
1行目
2行目
1
2
260
1
2
x[1][1]
x[1][2]
x[1][260]
x[2][1]
x[2][2]
1 byte
4 byte (int)
本当は、C言語の配列は x[0][0]から始まるが
ここでは分かり易さを重視して添字を1から書いた。
260行目
261行目
259
260
1
2
x[259][260]
x[260][260]
x[1][261]
x[2][261]
メイン関数側で 画像データimgを
入れたメモリ領域は、ここまで。
ここのメモリ領域に何が記録
されているかは全く不明。

メイン関数側で、配列 img に入れた画像データは
260行、260列、1画素4バイトなので、
img の内容を記録したメモリ上の先頭アドレス
（ポインタ）から 4 x 260 x 260 バイト目までのデータ
を 配列 x に託している（エイリアス）。

ところが、関数 smoothing は、どこまでが img の
データなのかは、全く知らない。

配列 x、y は、関数 smoothig の中では、読み出せと
命令された列の配列データは何列でも読む。

例えば 261列目の配列データを使えとプログラム文に
書けば、素直に先頭アドレスから順番に数えて 261列目
に相当するメモリ内の数字を読み出す。

```
void smoothing ( int x[ ][ 260 ], int y[ ][ 260 ] )
{
    for ( j=2;  j <= 261; j++ ) { for ( i=2; i<=255; i++ ) {
        y[ i ][ j ] = x[ i ][ j ] ;
    } }
 }
```

例えば、このようなプログラムを書いてしまっても
文法ミスは無いので、C言語コンパイラはこの文を
エラーとは指摘しないで実行する。
滅茶苦茶、意味不明な計算結果が出るだけである。

つまり、**配列の列数の管理は、**
プログラマに全責任を託されている。
意味不明な列数を配列に入れないように注意する。

しかし、C言語の配列の宣言法に慣れていないと、
このプログラムエラー（バグ）を犯しやすい。

img [256][256] と 配列を宣言すると、
img[0][0] から img[255][255] までの要素ができる。

配列の添字に 256 を記述すると、エラーが起きたり、
不明なデータが計算に使われてしまう。

解決策は、

画像などのデータがぴったり収まるような配列宣言
を避け、すこし余裕のある配列を宣言しよう。

画素データの最初の要素を img[0][0] から使えば
問題が生じない。しかし、これは慣れないと扱い難い。

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

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]    ;
      }}
}
```

3.

−1	−1	−1
0	0	0
1	1	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

```
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

```
//        filter.c

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

maxcount = get_int_max(img); disp_img( img, maxcount, 0, 0);

//------------------------- filtering  ------------------

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

filter1( img, img2); disp_img( img2, maxcount, 260*0, 260*1 );
filter2( img, img2); disp_img( img2, maxcount, 260*1, 260*1 );
filter3( img, img2); disp_img( img2, maxcount, 260*2, 260*1 );
filter4( img, img2); disp_img( img2, maxcount, 260*0, 260*2 );
filter5( img, img2); disp_img( img2, maxcount, 260*1, 260*2 );
```

画像 x の 画素最大値 を出力する関数

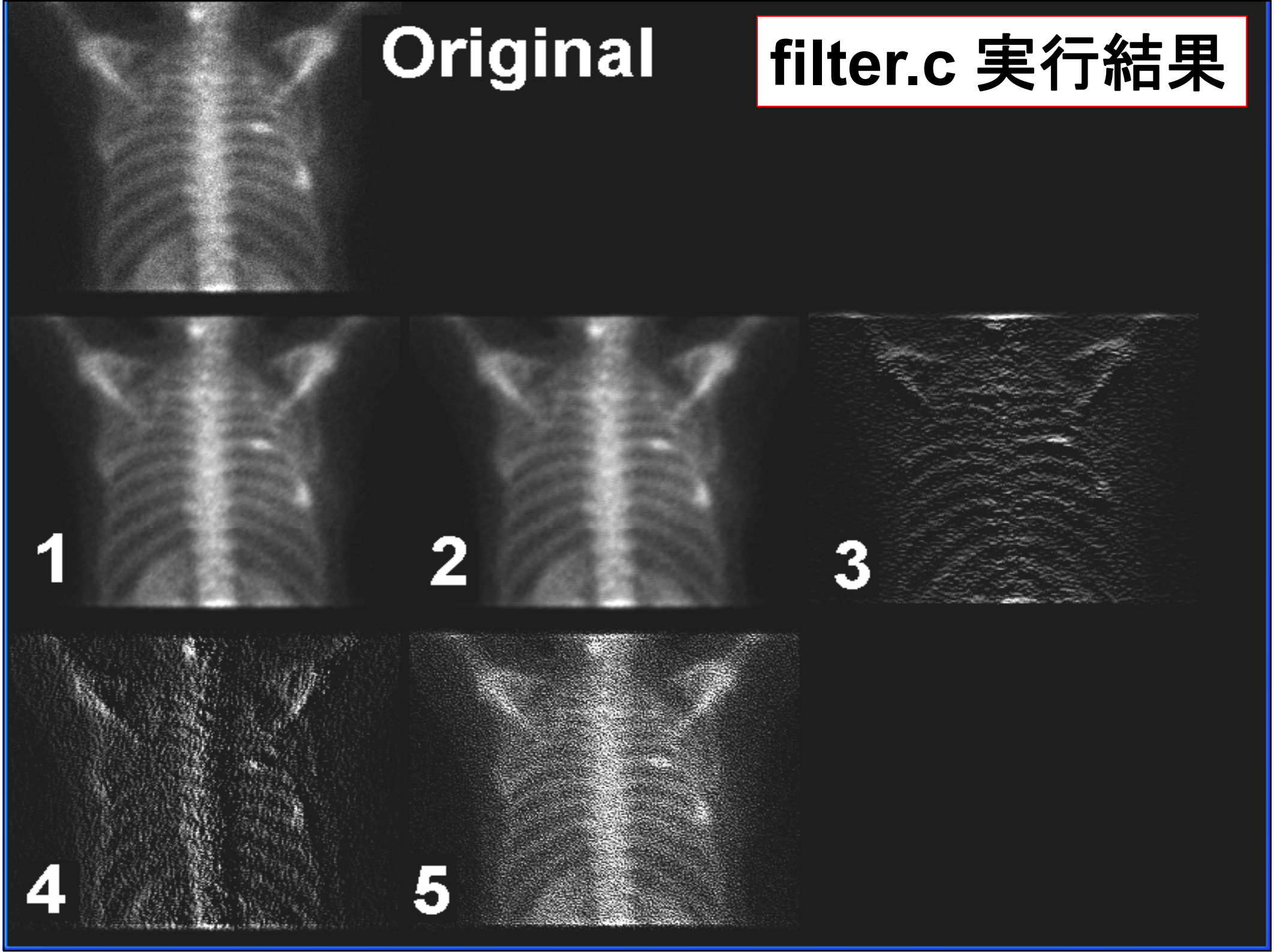
return関数　:　関数の出力値を設定する

get_int_max関数は、整数の出力値をもつので、int で定義。

```
int get_int_max( int x[ ][260] )
{
        int     i, j, max = 0 ;
        for(j=1;j<=256;j++){ for(i=1;i<=256;i++){

            if(x[i][j] > max) max = x[i][j];
        }}
        printf("¥n¥n max count = %d ", max);
        return( max );
}
```

画像 x を 最大値 max、座標（xp、yp）に描画する関数

```
void disp_img( int x[ ][260], int max, int xp, int yp )
{
        int i, j, count ;

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

            count = x[i][j];  count *= 255 / max;

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

                SetColor( RGB(count, count, count) );
                SetPixel( i+xp, j+yp );
    }}
}
```


Original
filter.c 実行結果
1
2
3
4
5

SPECT（ Single Photon Emission CT ）

PET（ Positron Emission CT ）の原理

断層画像を得る方法

フィルタ重畳逆投影法

FBP （ Filtered Back Projection ）

逐次近似再構成法 Iterative Reconstruction

MLEM （ Maximun Likelihood Expectation Maximization ）

OSEM （ Ordered Subsets Expectation Maximization ）

Convolution　重畳積分

FBP

Filtered Back Projection

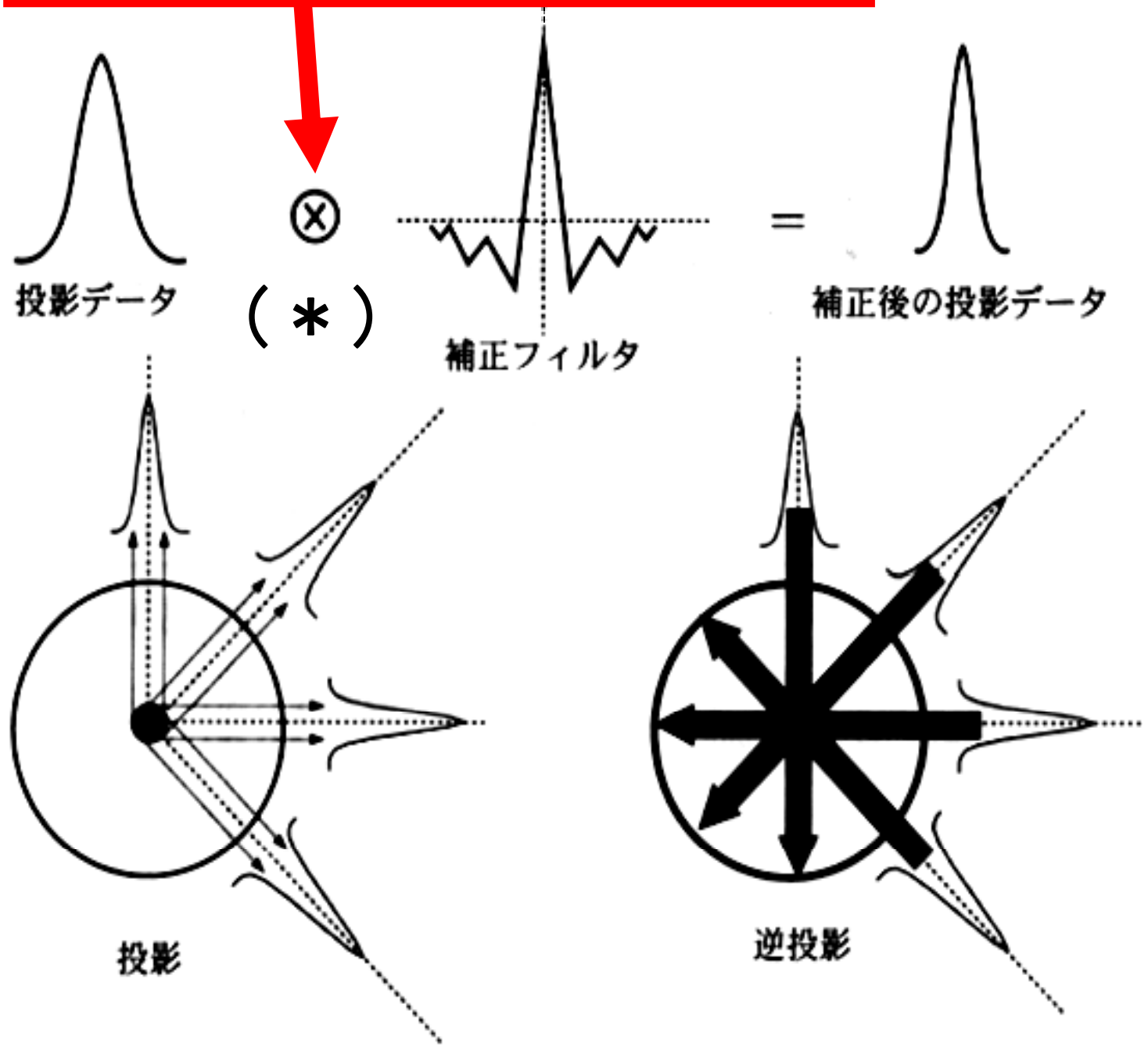
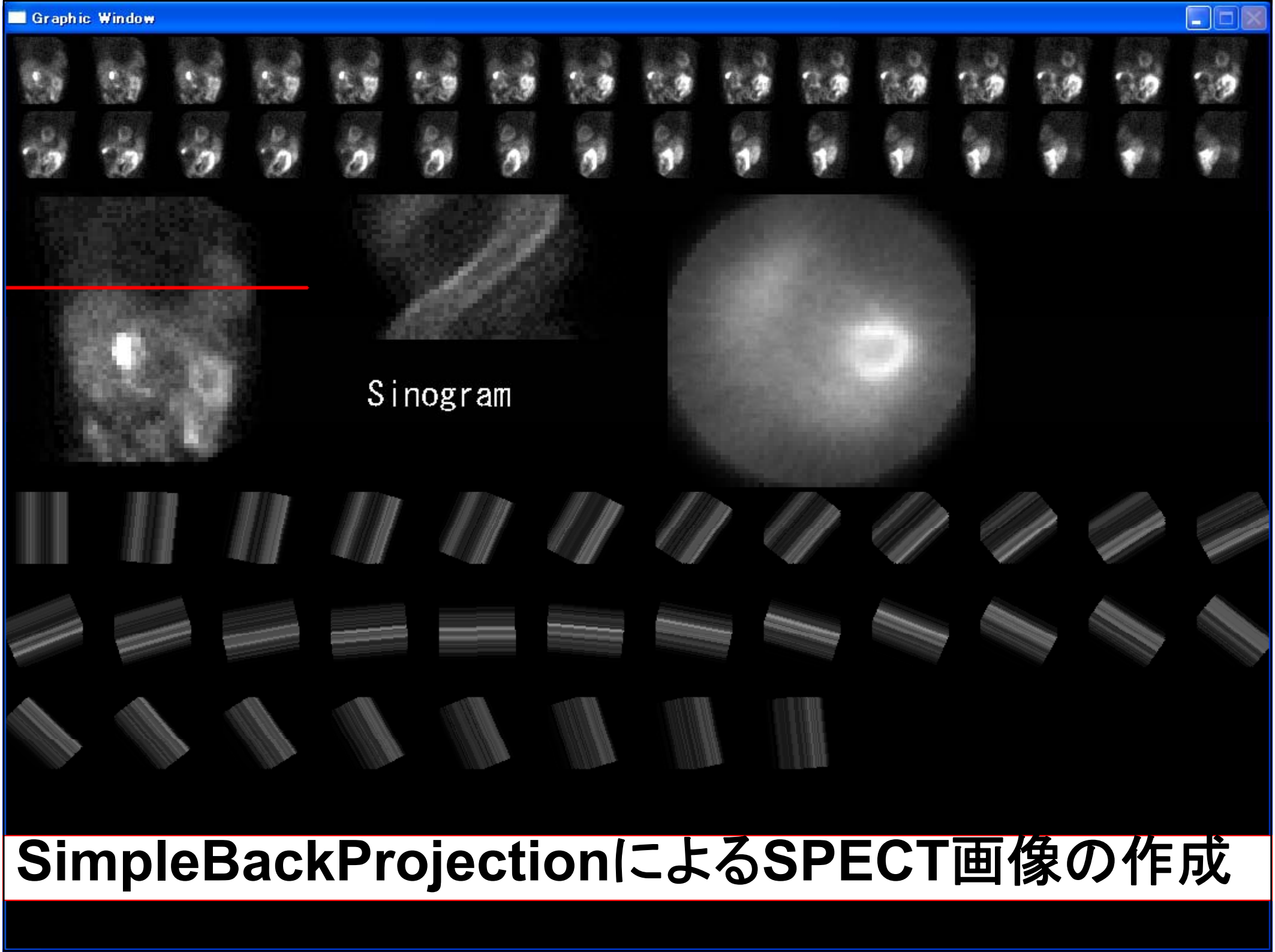
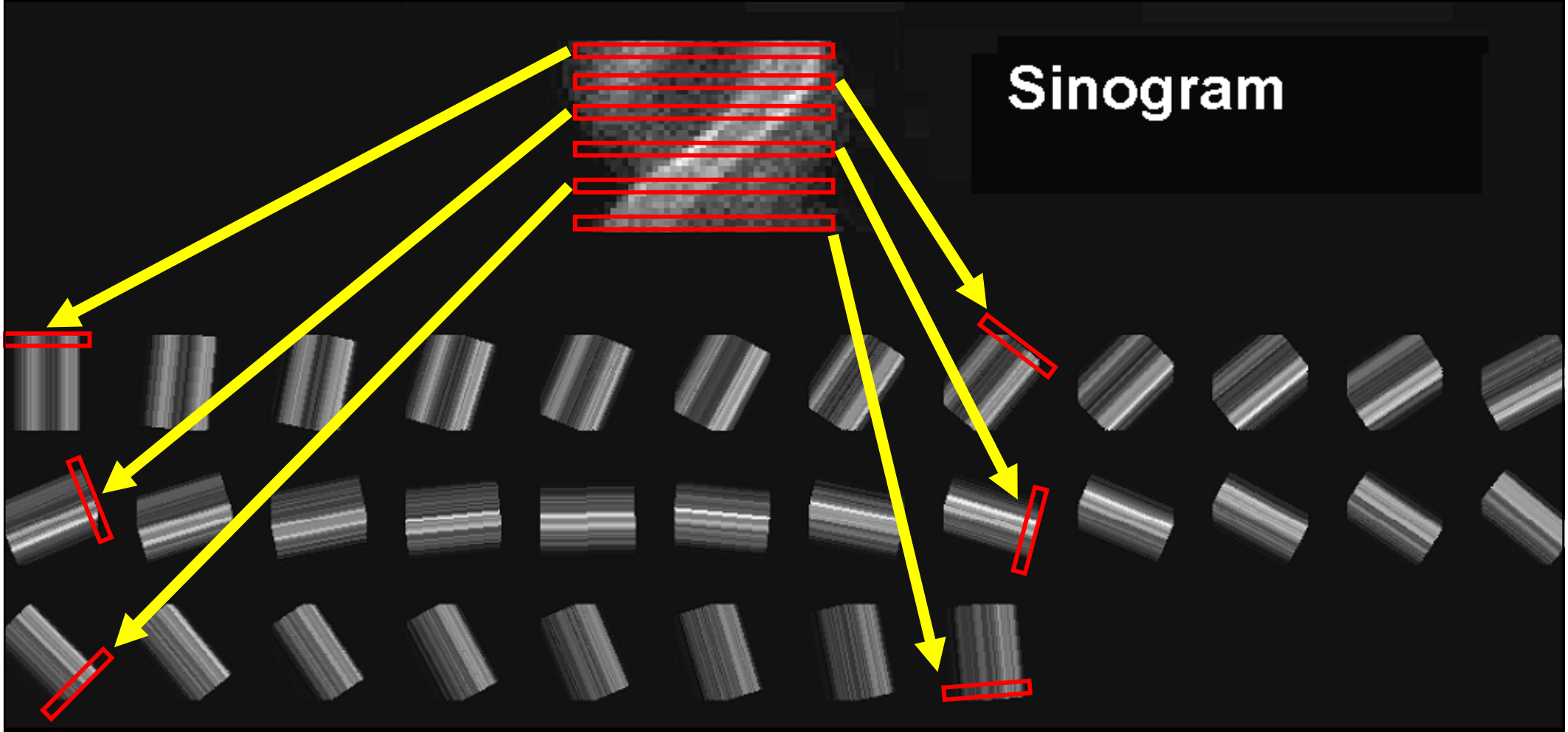


図 4・31　投影データ収集とフィルタ補正逆投影

雑音（高周波成分）を除去するフィルタで高周波成分は除去できるが，同時に周辺部やエッジにボケを生ずる．



SimpleBackProjectionによるSPECT画像の作成



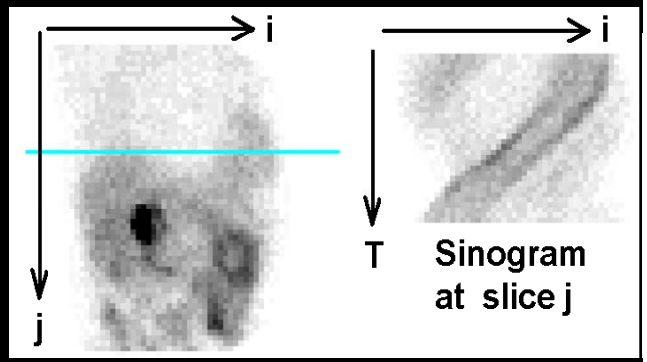
サイノグラムの各スライスの１次元配列は、
３２方向の各々の角度から収集されたデータ。

このデータから、収集された各々の角度に傾いた
２次元透視画像（ Ｐθ ）を作成する。

単純重ね合わせ再構成法　Simple Back Projection

収集された各々の角度に傾いた2次元透視画像（ Ｐθ ）を
全部単純に重ねると再構成画像ができる。
（回転中心近傍の値が盛り上がった不正確な画像。）

スライス j におけるサイノグラムを求める。
サイノグラムの各スライスの１次元配列は、
３２方向の各々の角度から収集されたデータ。

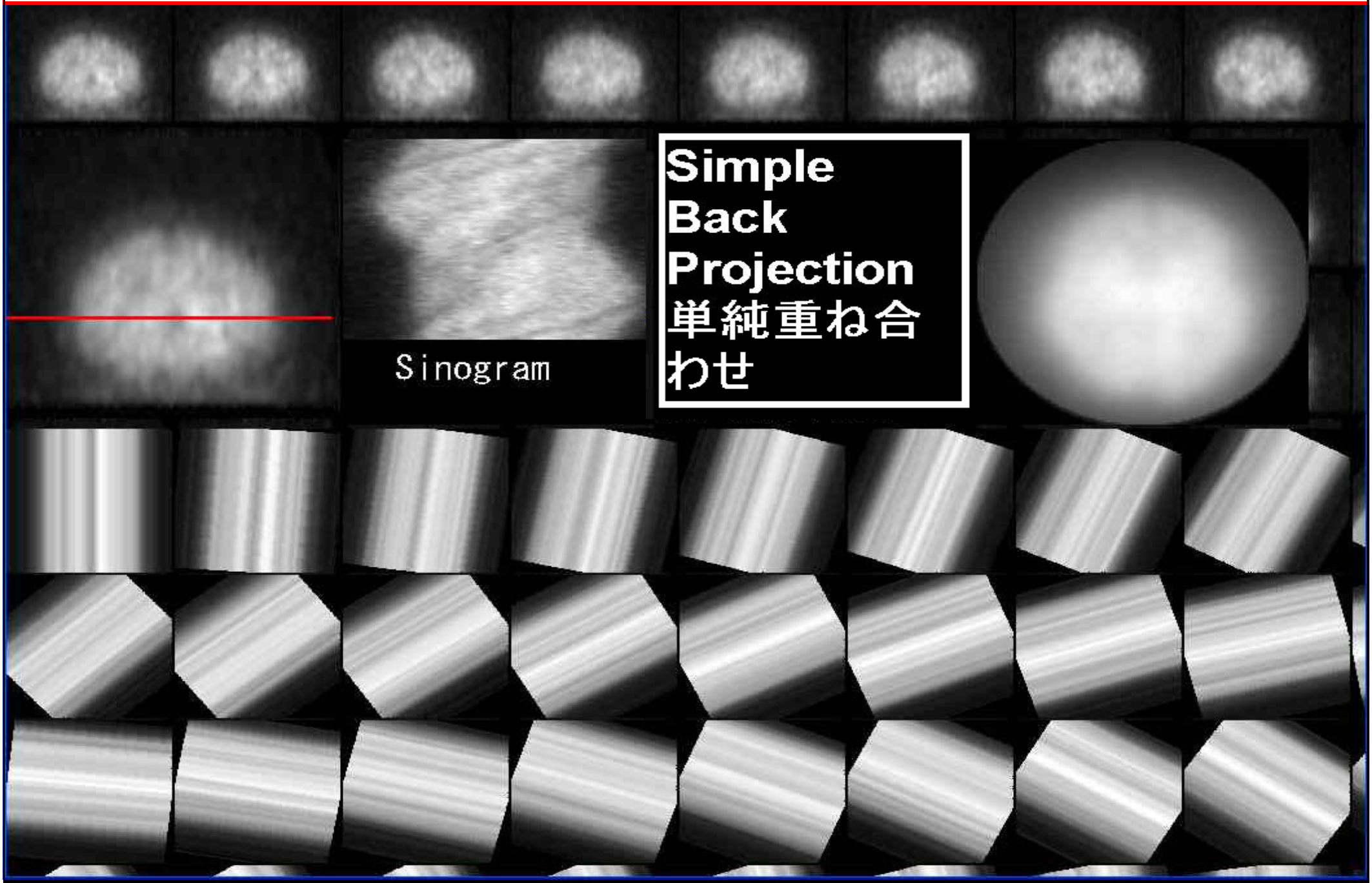


サイノグラムの各スライスの１次元配列から、
収集された各々の角度に傾いた２次元透視画像Ｐθを作成する。
Ｐθを単純に重ね合わせた画像をＩとすると

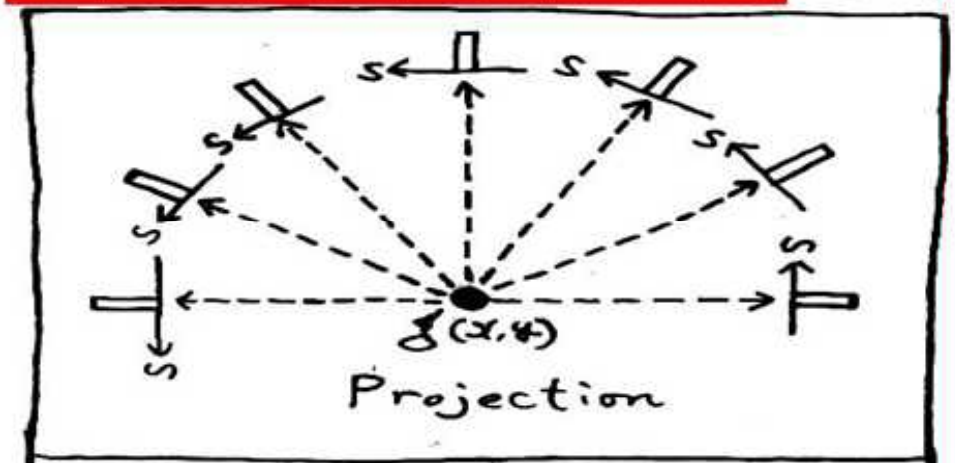
$$I = \int P\theta \, d\theta \quad \text{(Simple back projection)}$$

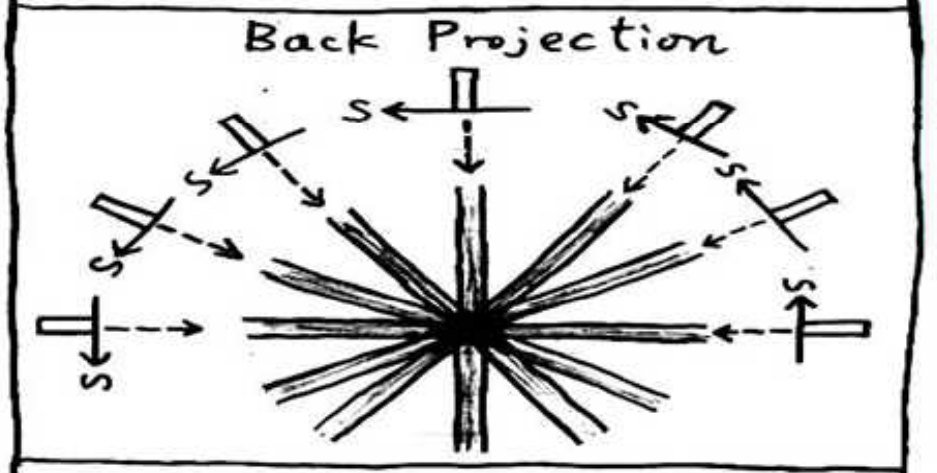
Ｉ　は、回転中心部ほど重ね合せ回数が多くなり、
中心から距離が遠いほどカウントの低い像になる。

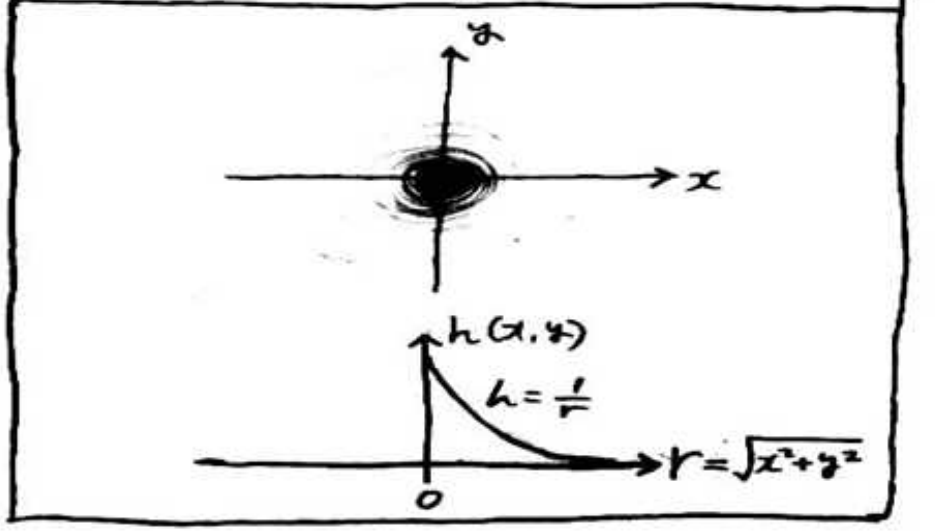
123I-IMP Brain SPECT　単純重ね合わせ像は回転中心近傍のカウントが持ち上がる。画像が不鮮明。



⑤ Back Projection







xy平面上の1点 f(x,y) の
透視データ $P_\theta(s)$ (Projection)
を $\theta = 0 \sim \pi$ の範囲で求める.

次に $P_\theta(s)$ を重ね合わせて
画像の再構成を行なう
(Back Projection)

得られる画像 h(s,θ) は

$$h(s,\theta) = \sum P_\theta(s)\,\Delta\theta = \int_0^\pi P_\theta(s)\,d\theta$$

h(s,θ) は中心部ほど重ね
合わせ回数が多くなり、中心から
遠ざかるほど少なくなって、
元の点の像には戻らず、
点の存在した位置からの
距離に反比例した
濃度分布の像をつくる.

重ね合わせの
点像分布関数 h(x,y) は

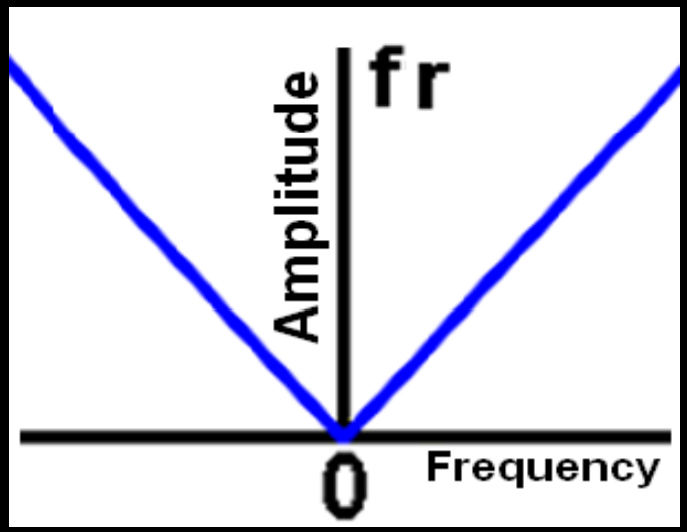
$$h(x,y) = \frac{1}{r} = \frac{1}{\sqrt{x^2+y^2}}$$

つまり、回転中心からの距離ｒに反比例した濃度に補正する
フィルタ　１／ｒ　を正確な断層像ｇに畳み込んだ像が Ｉ である。
式で表現すると　Ｉ＝ｇ＊（１／ｒ）となる。
Ｉ、ｇ、１／ｒ のフーリエ変換を Ｌ、Ｇ、Ｆ（１／ｒ）と
表現すると、畳み込みの定理より
Ｌ＝Ｇ・Ｆ（１／ｒ）となる。

２次元フーリエ変換の公式の極座標表現を用いると、
（ｆ ｒは周波数空間上の原点からの距離 ）
Ｆ（１／ｒ）＝
∫∫（１／ｒ）exp（－ｊ(2πｒ ｆｒ)） ｒｄｒｄθ ＝１／ｆ ｒ

これより　Ｌ ＝ Ｇ ／ｆ ｒ　なので　Ｇ＝Ｌ・ｆ ｒ

Ｇ ＝Ｌ・ｆｒ　の 意味は、 ２次元周波数空間上で、
単純重ね合わせ画像をフーリエ変換した２次元データ Ｌ に、
フィルタ関数 ｆｒ（ ｆｒは周波数空間上の原点からの距離）
をかけると、正しい再構成画像をフーリエ変換したデータ Ｇ
になる。



Ｇ＝Ｌ・ｆｒ　に、畳み込みの定理 を
用いると、以下のような実空間での計算に
変換できる。

この式を 逆フーリエ変換すると、

ｇ＝ｌ＊ｈ　（ ｈ は フィルタｆｒ の 逆フーリエ変換 ）

この式に、$I = \int P\theta \, d\theta$ を代入すると、

$g = \int P\theta \, d\theta * h$

$g = \int (P\theta * h) \, d\theta$ （hはθと独立した値なので交換可）

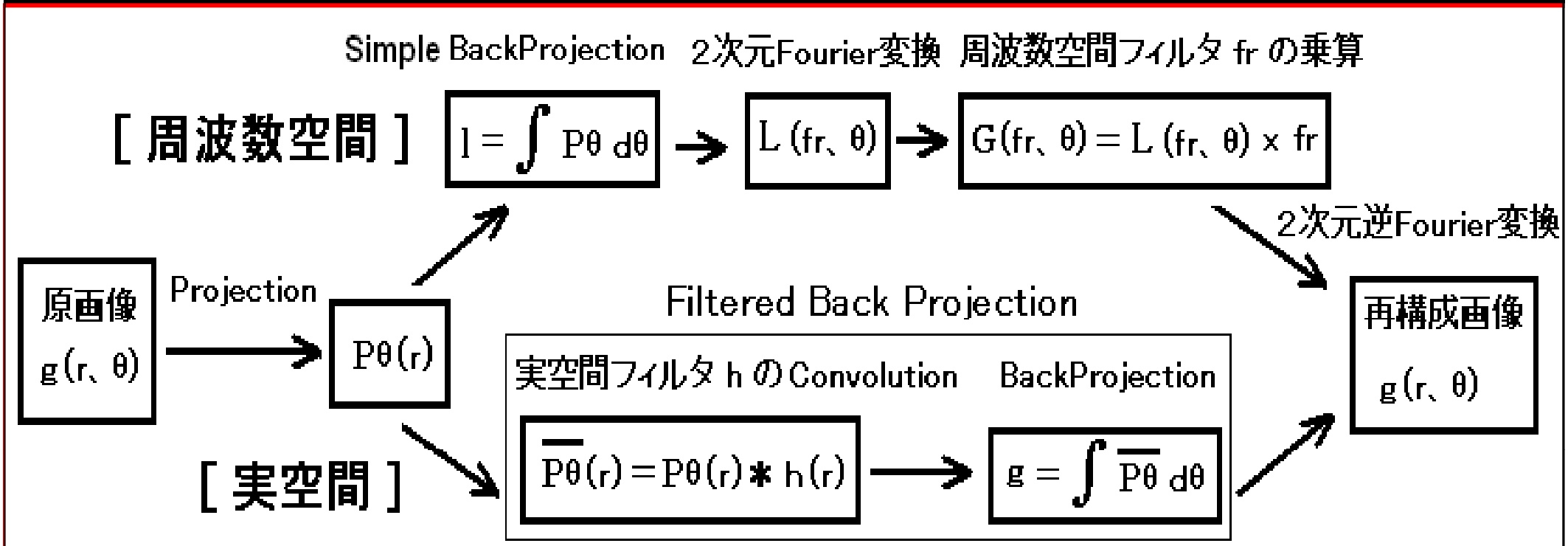
$$g = \int \overline{P\theta} \, d\theta \quad \left(\overline{P\theta} = P\theta * h \right)$$ **FBPの式**

Pθ に 実空間フィルタ h (= frの逆フーリエ変換) を畳み込めば、
重ね合せると正確な断層像 g になる2次元透視画像 $\overline{P\theta}$ を
算出できる。これを **Filtered Back Projection（FBP）**という。

周波数空間での実際の計算においては、フィルタ H(=fr)は
常に正の値であり(絶対値)、
さらにサンプリング定理より、ナイキスト周波数以上の成分を
削除する必要があるので、

フィルタ逆重畳画像再構成法
Filtered Back Projection　(FBP)

サイノグラムの2次元透視画像 **Pθ** に 、
実空間フィルタ **h** (= frの逆フーリエ変換) を畳み込めば、
重ね合せると正確な断層像 g になる2次元透視画像 $\overline{\mathbf{P\theta}}$ を
算出できる。



周波数空間での再構成フィルタ Hは、

H ＝｜ｆｒ｜（ｆｒがナイキスト周波数未満の場合）

H ＝　０　　（ｆｒがナイキスト周波数以上の場合）

となる。　これをＲａｍｐフィルタという。

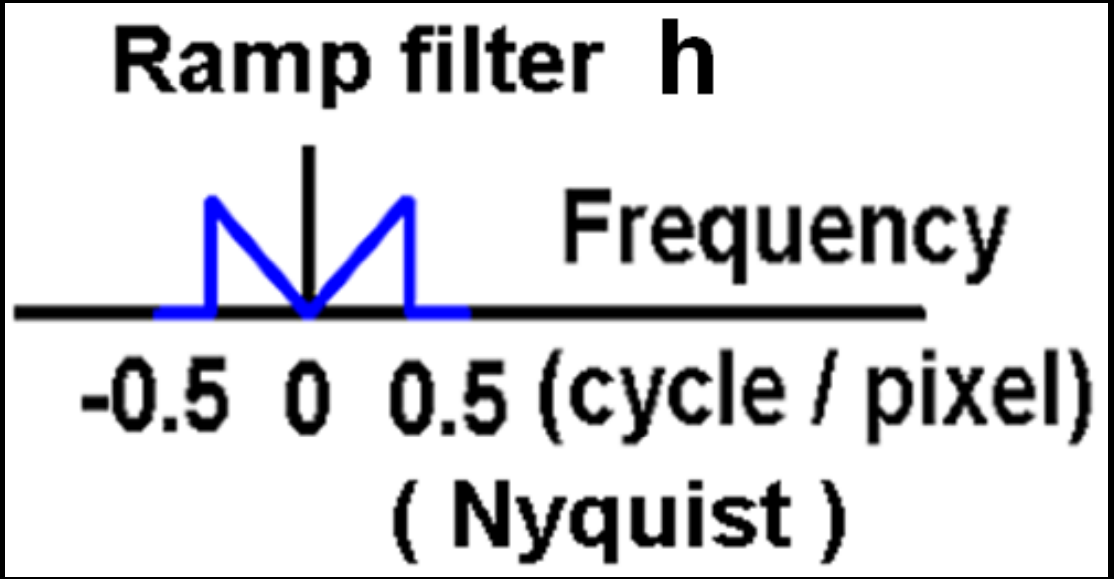
Ｒａｍｐフィルタを逆フーリエ変換して

実空間Ｒａｍｐフィルタｈにしてから、

実空間でPθにｈを畳み込む。

$$\overline{P\theta} = P\theta * h$$

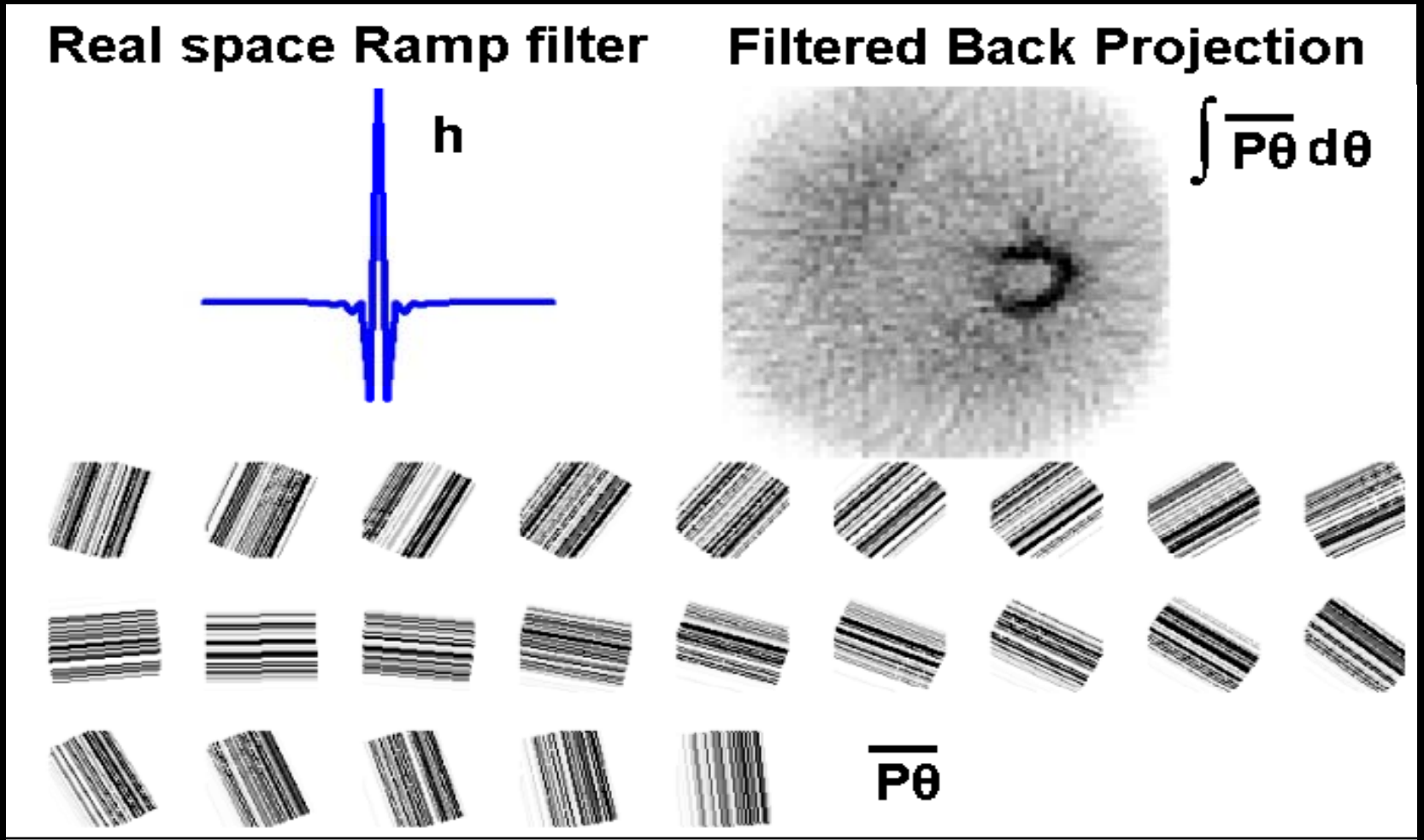
（＊は畳み込み演算）



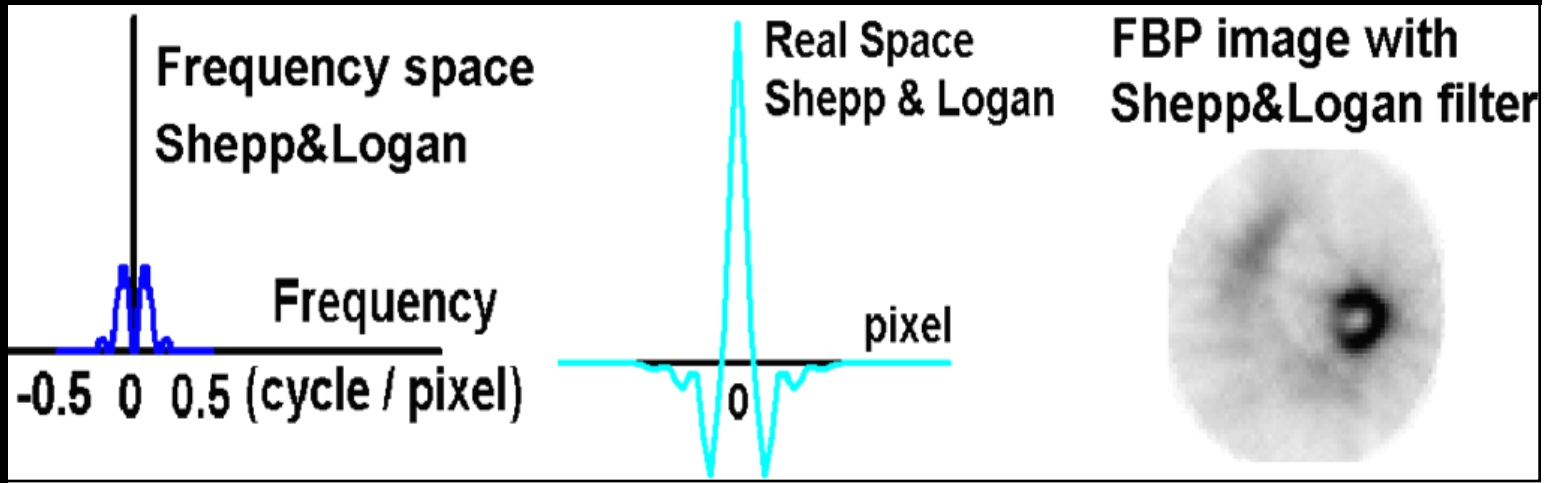
hを畳み込んだ2次元透視画像 $\overline{P\theta}$ を単純に重ね合わせた画像 g は、

$g = \int \overline{P\theta}\, d\theta$　　（ Filtered back projection ）

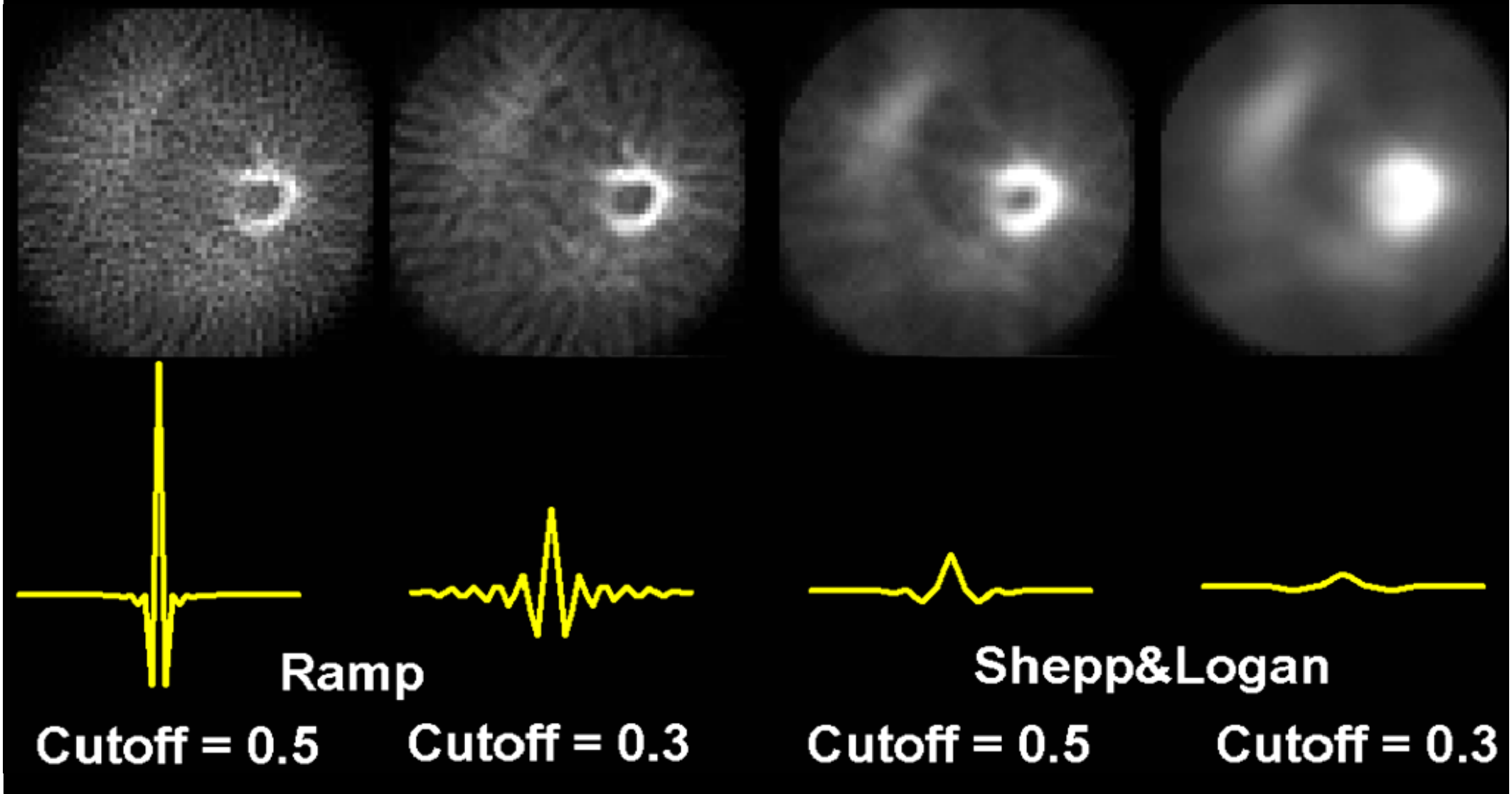
Filtered back projection は、
回転中心部ほど重ね合わせ回数が多くなる誤差が
フィルタhによって補正され、正しい再構成画像となる。



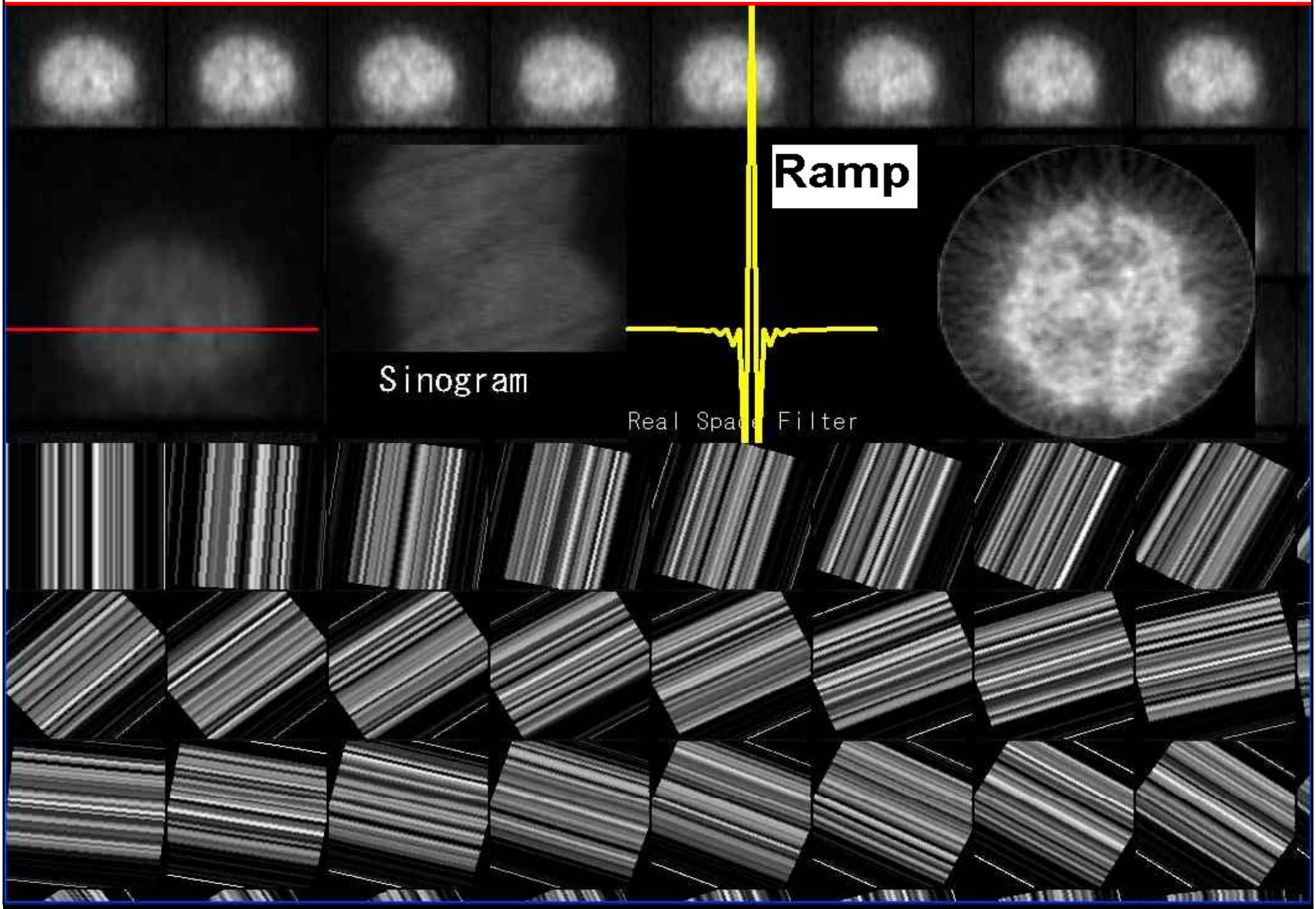
Rampフィルタは理論的には正確な再構成フィルタだが、
実際の臨床データに適用すると ナイキスト周波数以上の高周波成分を不連続に遮断するために生じる高周波ノイズや 放射状アーチファクトが目立つ場合が多いため、
臨床では高周波成分を抑制する工夫を施した再構成フィルタが用いられている。
SPECTで用いられる一般的な再構成フィルタとして、
Shepp＆Loganフィルタがある。
周波数空間上で、ナイキスト周波数近傍の高周波成分を連続的に減衰させるように設計されており
再構成画像に生じる**高周波ノイズや放射状アーチファクトを抑制**する効果をもつ。



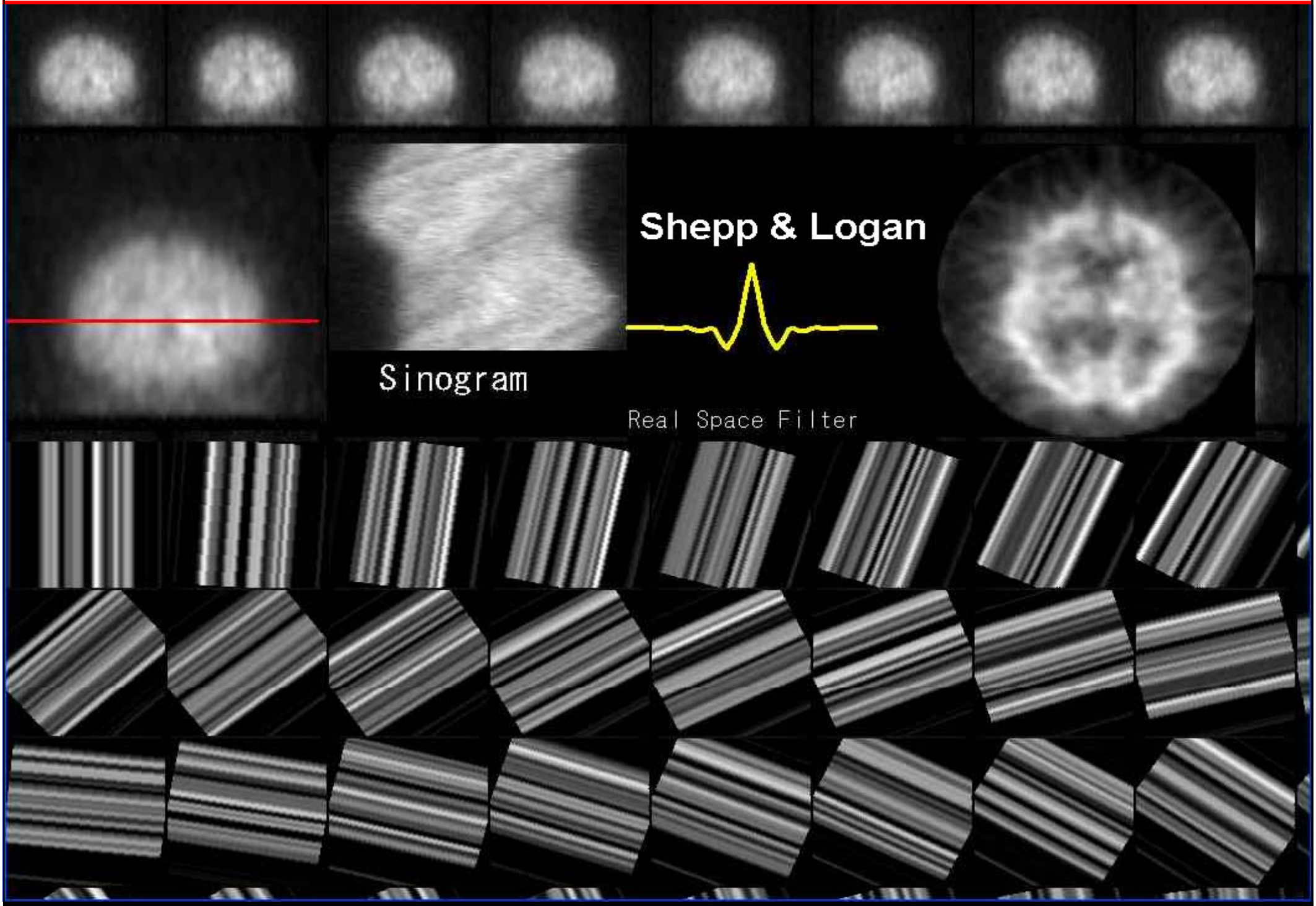
実空間フィルタを畳込んだ2次元透視画像（$\overline{P\theta}$）を
重ね合わせるとフィルタ逆投影再構成像ができる。
フィルタの形状で、再構成画像の高周波成分が変る。

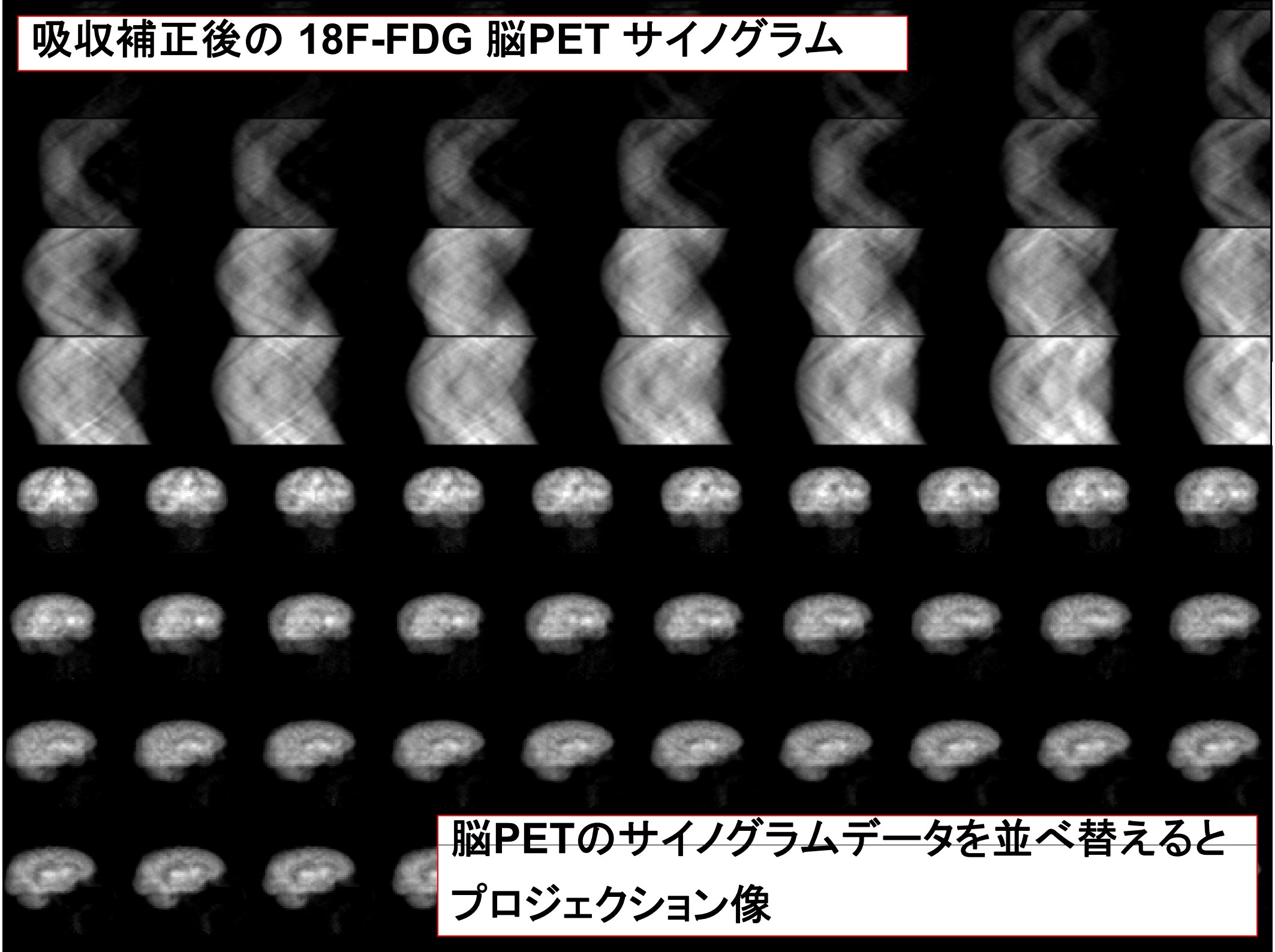


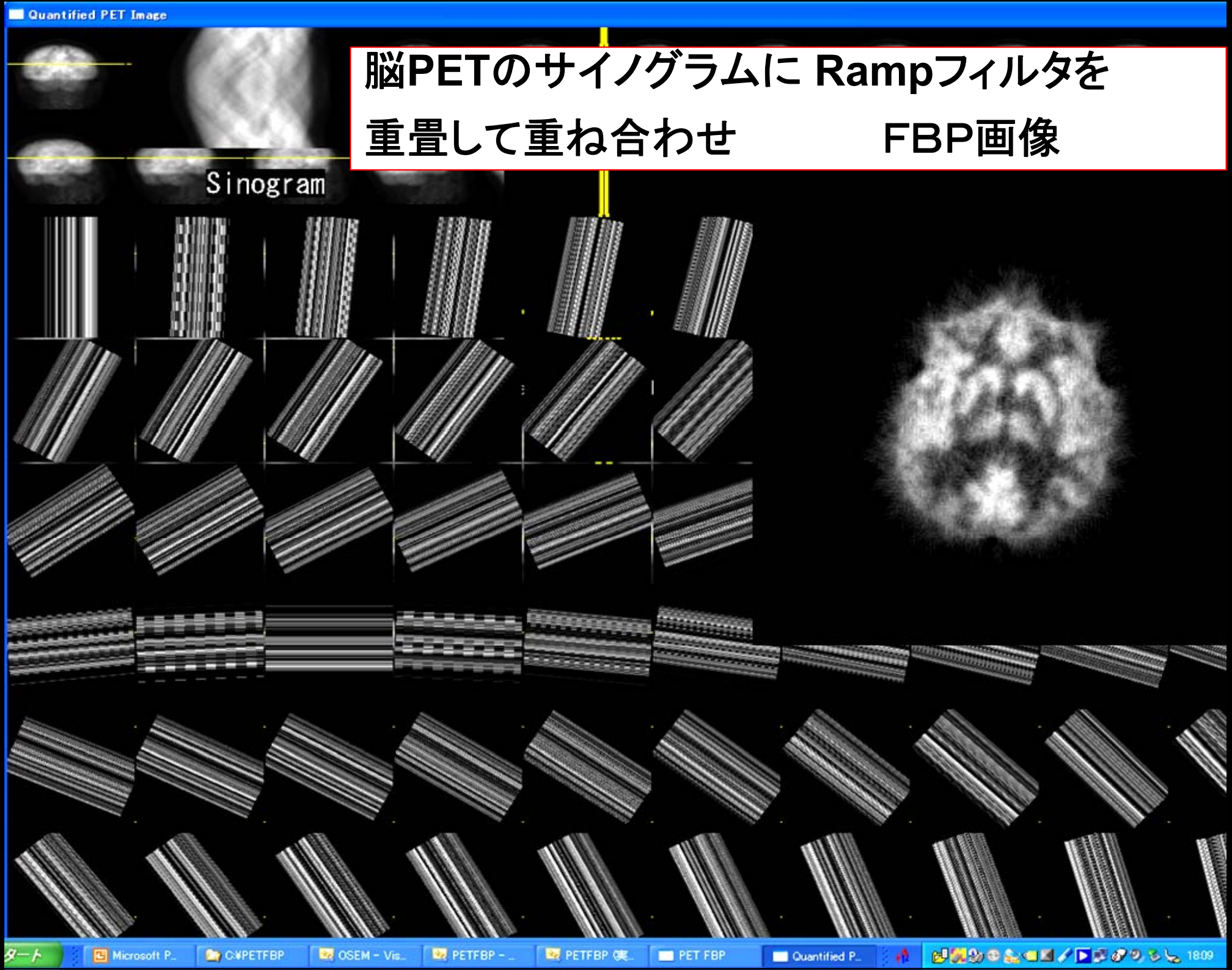
123I-IMP Brain SPECT FBP with Ramp filter



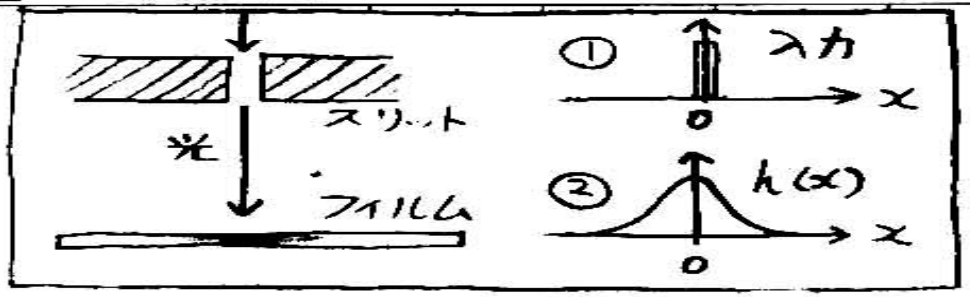
123I-IMP Brain SPECT　FBP with Shepp&Logan filter




吸収補正後の 18F-FDG 脳PET サイノグラム
脳PETのサイノグラムデータを並べ替えると
プロジェクション像


Quantified PET Image
Sinogram
脳PETのサイノグラムに Rampフィルタを
重畳して重ね合わせ
FBP画像
Microsoft P...
C:¥PETFBP
OSEM - Vis...
PETFBP - ...
PETFBP (実...
PET FBP
Quantified P...
18:09

畳込みの定理　Convolution

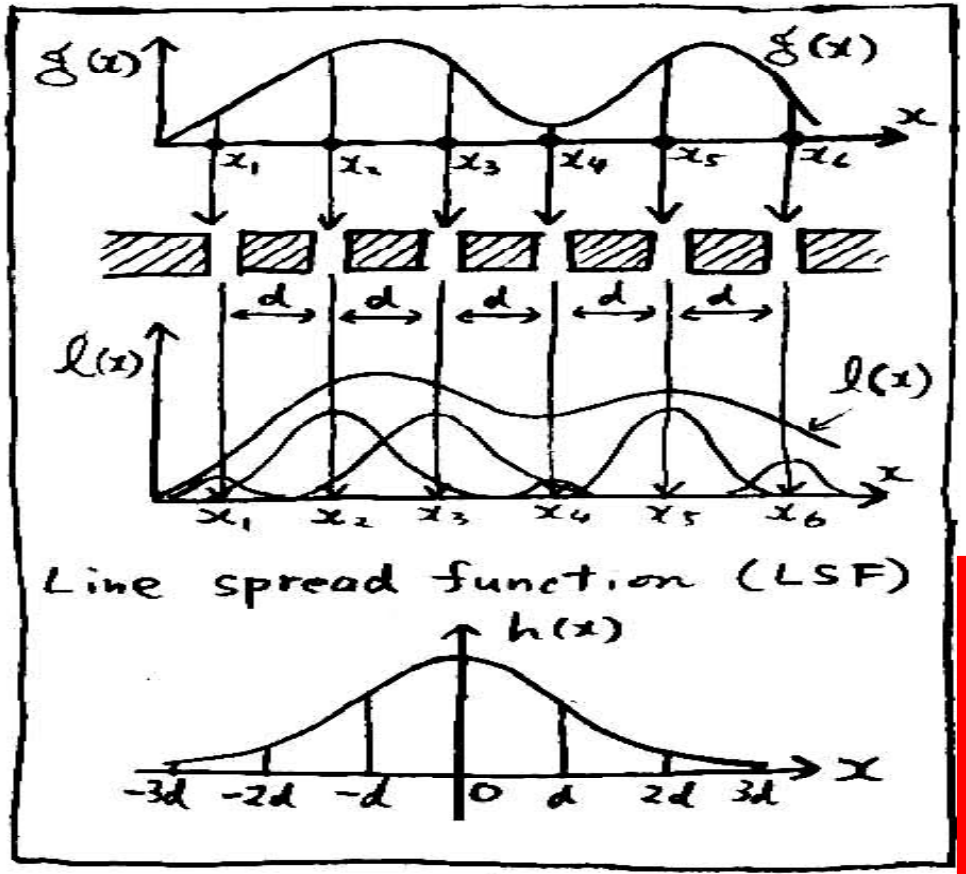


スリットを通したビーム①がフィルム上で②のようなh(x)の濃度分布の像をつくるとする。

x軸に沿って明るさがf(x)である線状の被写体を，間隔dのスリットを通してフィルムに写す。

フィルムのx₄の位置での濃度は

$$l(x_4) = f(x_4)h(0) + f(x_5)h(-d) + f(x_6)h(-2d) + f(x_3)h(d) + f(x_2)h(2d) + f(x_1)h(3d)$$



任意の座標x_iにおける$l(x_i)$は

$$l(x_i) = f(x_i)h(0) + f(x_{i+1})h(x_i - x_{i+1}) + f(x_{i+2})h(x_i - x_{i+2}) + \cdots + f(x_{i-1})h(x_i - x_{i-1}) + f(x_{i-2})h(x_i - x_{i-2}) + \cdots$$

$$= \sum_{n=-\infty}^{\infty} f(x_n)h(x_i - x_n)$$

スリットを取りはずすと（スリット間隔を無限に狭くすると）
任意の座標 x における フィルムの濃度 $l(x)$ は

$$l(x)=\int_{-\infty}^{\infty} g(n)\,h(x-n)\,dn$$

convolution 積分
（たたみ込み積分）

これを $l(x)=g(x)*h(x)$ と表わす。

$h(x)$ を convolution 関数という。

$g(x)$ が $h(x)$ のために $l(x)$ に ボケてしまったことを意味する。

たたみ込みの定理

$h(x-n)$ の Fourier 変換を $H(f)$ とすると

$$h(x-n)=\int H(f)\,e^{j(2\pi f(x-n))}\,df \quad (\text{逆 Fourier 変換})$$

これを $l(x)=\int g(n)\,h(x-n)\,dn$ に代入すると

$$l(x)=\int g(n)\left[\int H(f)\,e^{j(2\pi fx)}\,e^{-j(2\pi fn)}\,df\right]dn$$

$$=\int\left[\int g(n)\,e^{-j(2\pi fn)}\,dn\right]H(f)\,e^{j(2\pi fx)}\,df$$

$$=\int G(f)\,H(f)\,e^{j(2\pi fx)}\,df$$

$l(x)$ の Fourier 変換を $L(f)$ とすると

$l(x)=\int L(f)\,e^{j(2\pi fx)}\,df$, よって $L(f)=G(f)\,H(f)$

畳み込みの定理

データ g をフーリエ変換して、

その周波数空間成分 G に

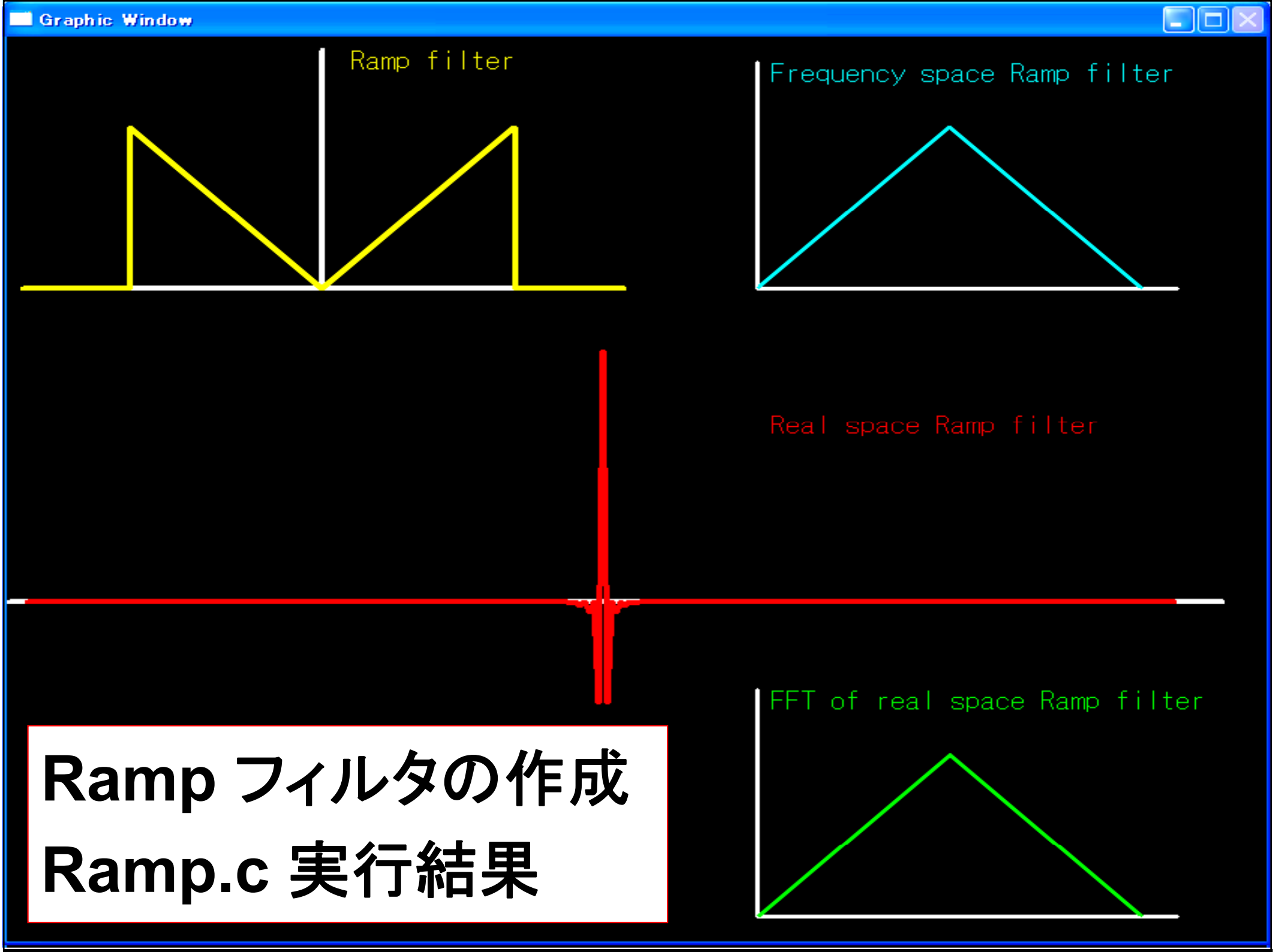
周波数空間 Rampフィルタ H をかけて

逆フーリエ変換すると、

実空間で、実空間 Rampフィルタ h を

g に畳み込みしたデータと同じになる。

（ G x H と g * h は等価演算 ）



Ramp フィルタの作成

Ramp.c 実行結果

測定方法によって定まる Nyquist 周波数を f_0 とすると

実際の計算式は $F(f_r) = |f_r| \cdot G$ ではなく,

$$F(f_r) = H(f_r) \cdot G \quad , \quad H(f_r) = \begin{cases} |f_r| & (|f_r| < |f_0|) \\ 0 & (|f_r| \geq |f_0|) \end{cases}$$

この $H(f_r)$ を逆 Fourier 変換したものを $h(r)$ とすると,

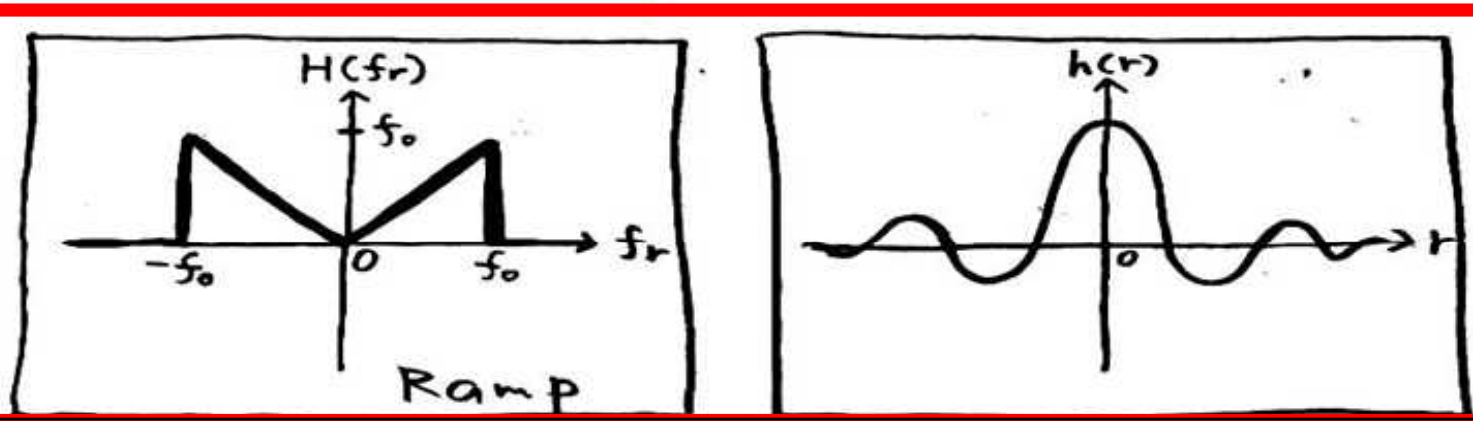
$$h(r) = \int_{-f_0}^{f_0} |f_r| e^{j(2\pi f_r r)} df_r$$

$$= 2\int_0^{f_0} |f_r| \cos(2\pi f_r r)\, df_r$$

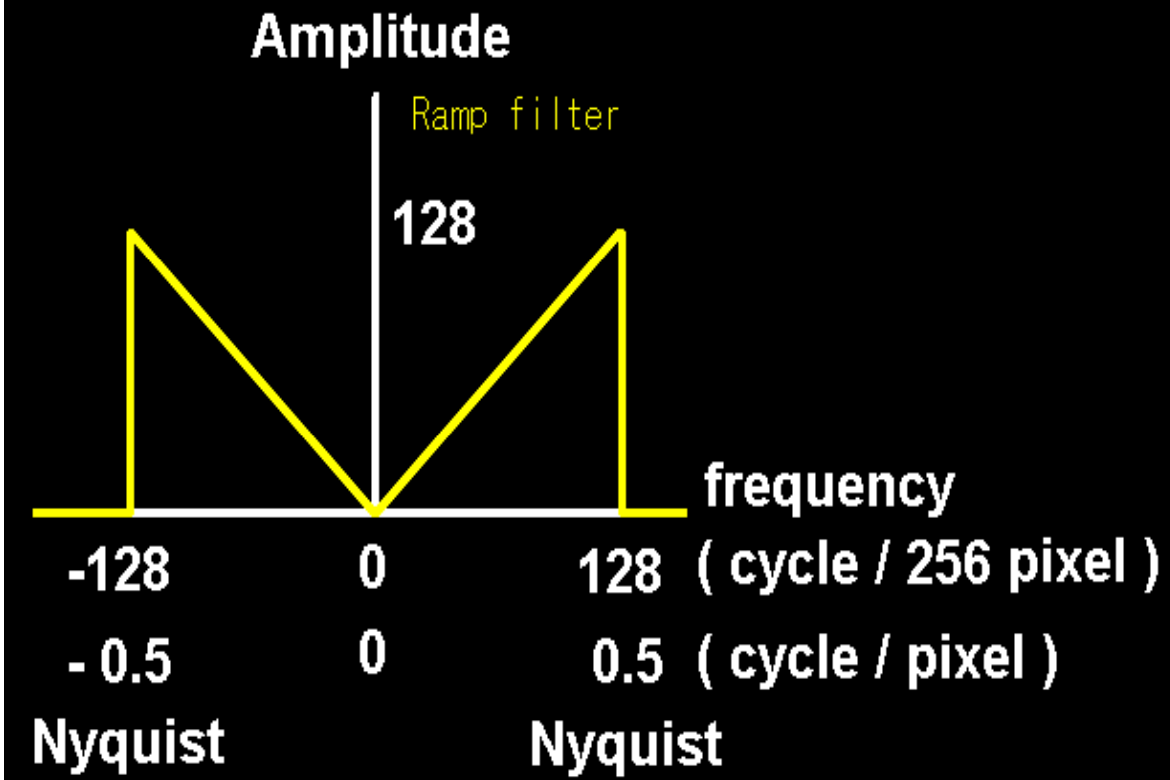
$$= 2f_0^2 \frac{\sin(2\pi f_0 r)}{2\pi f_0 r} - \left(f_0 \frac{\sin(\pi f_0 r)}{\pi f_0 r}\right)^2$$

$$h(r) = 2f_0^2 \operatorname{sinc}(2f_0 r) - f_0^2 \operatorname{sinc}^2(f_0 r)$$

Ramp filter $\left(\operatorname{sinc} x = \frac{\sin(\pi x)}{\pi x}\right)$



```
//------ Generate frequency space Ramp filter ---------
f0 = 128;   // Nyquist
for ( i=-200; i<=200; i++){
        if ( abs( i )  <=  f0 ) R[ i+200 ] = ( double) abs( i ) ;
        else                    R[ i+200 ] = 0.0 ;
}
```



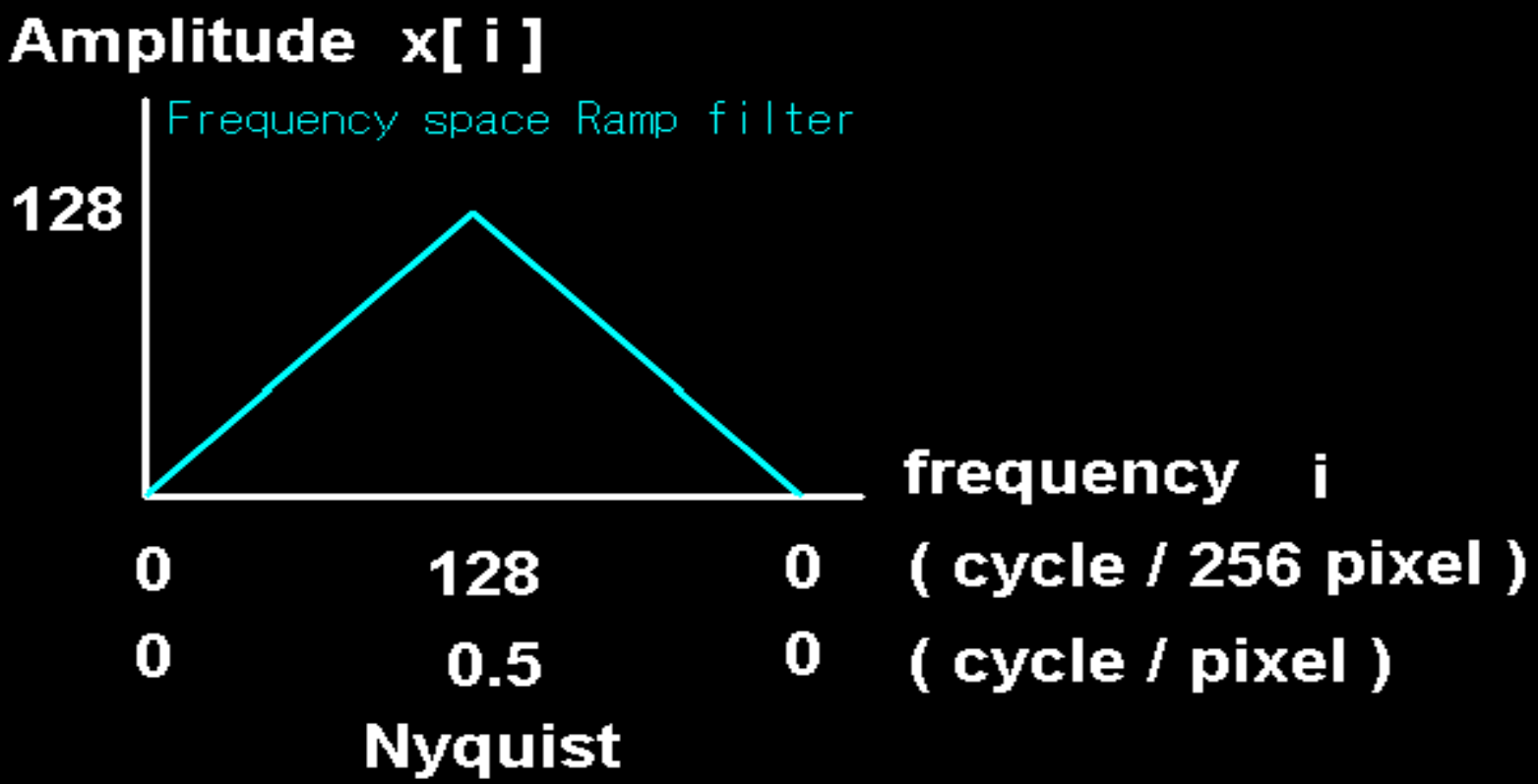
if (A) B ; else C ;

Aが正しければBを実行、
そうでなければCを実行。

abs ( ) ;
絶対値を計算する関数

```
//------------ Transform Ramp filter -------------
        for( i=1; i<=128; i++ ){ x[i]       = R[ 200+i ] ; }
        for( i=1; i<=128; i++ ){ x[i+128] = R[ 329-i ] ; }
```

1次元 FFT を実行する1次元データは、
両端に直流成分(周波数 0)、中央が高周波(ナイキスト)になるように並べ替える。



```
//---------- Save Frequency space Ramp filter ----------
printf("¥n¥n Save Frequency space Ramp filter "); scanf("%c",&yn);
        GetFileName( f, 1);          fp = fopen( f, "wt");
        for( i=1; i<=256; i++){ fprintf ( fp, " %lf ¥n",  x[ i ] ) ; }
        fclose( fp );
```

周波数空間の Ramp filter を ファイルに保存する。

GetFileName(f, 1); で、保存するファイル名を指定する

ダイアログが開く。

fp=fopen(f, "wt"); で、テキスト形式ファイルとして書き込む。

fprintf (fp, " %lf ¥n", x[i]) ;   file - printf

fprintf () 関数は、ファイルに文字や数字を書き込む関数。

グラフ描画関数　Graph ( ) の作成

座標（x1, y1）から（x2, y2）の範囲に、配列 x[ ] のグラフを最大値 max、color 色 で描画。

```
void Graph ( double x[ ], int x1, int y1, int x2, int y2, double max,
                COLORREF color, char *s )
{
   int    i, ip, jp, ip2, jp2 ;
   PenWidth(3);
   SetColor(WHITE); Line(x1, y1, x2, y1); Line(x1, y1, x1, y2);
   SetColor(color); FontSize(20); DrawString(x1+10, y2, s );
     for(i=1; i<=255; i++){
        ip = x1 + i ;  ip2 = x1 + (i+1) ;
        jp  = y1 - (int)(x[i]*100./max);  jp2 = y1 - (int)(x[i+1]*100./max);
        Line( ip, jp, ip2, jp2 );
     }
}
```

グラフ描画関数　Graph ( ) の 最後の引数 s は、グラフに添付する文字列の、メモリ上の先頭アドレス(ポインタ)。
したがって この引数の変数型指定は　char * となる。

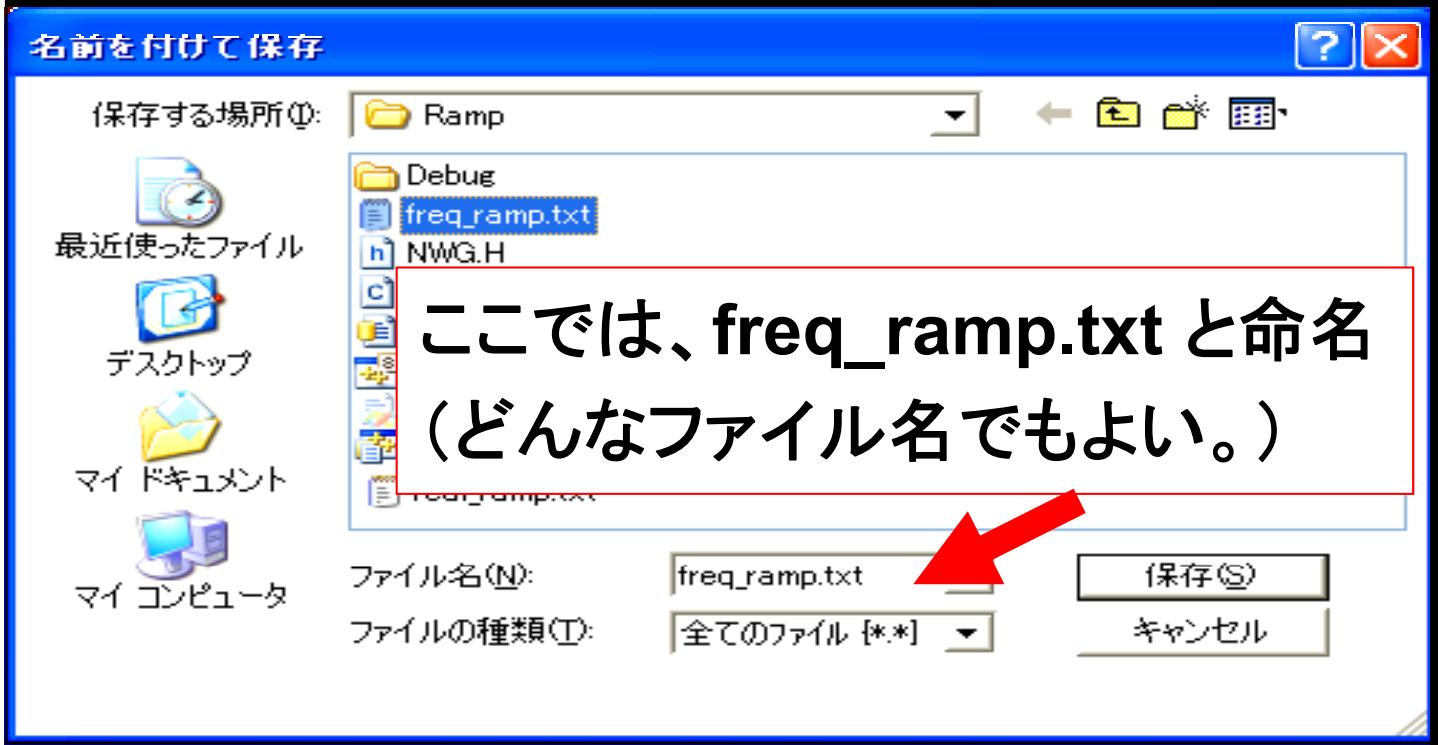
DrawString(x, y, s)
グラフィック画面の座標 (x, y) に、FontoSize関数で指定した文字サイズで、文字列 s を書く。

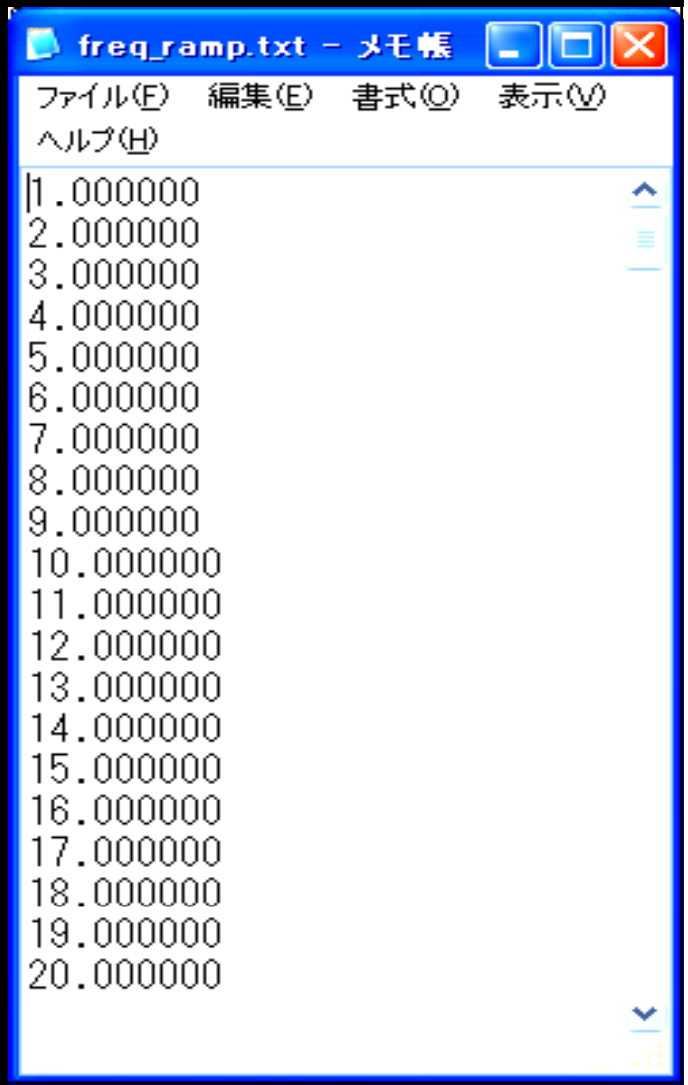
色を指定する変数 color の型は、COLORREF型 になる。

```
fprintf ( fp, “ %lf ¥n”,  x[ i ] ) ;
```

fp は、保存するファイルのポインタ。
“ %lf ¥n”, x[i]　は、printf関数と同じ様式で、
実数変数 x[i] の内容を、実数形式（%lf）でファイルに書く。

作成されたファイルを メモ帳などで開くと
テキスト形式で x[i] が保存されることを
確認できる。（¥nで各数字が改行される。）





```
//------- Transform Ramp filter into Real space ---------
printf("¥n¥nTransform Ramp filter into Real space");scanf("%c",&yn);
        for( i=1; i<=256; i++ ){ y[ i ] = 0.0 ; }
        FFT1D( x , y , 256 , -1 ) ;
```

1次元 高速 逆フーリエ変換　IFFT

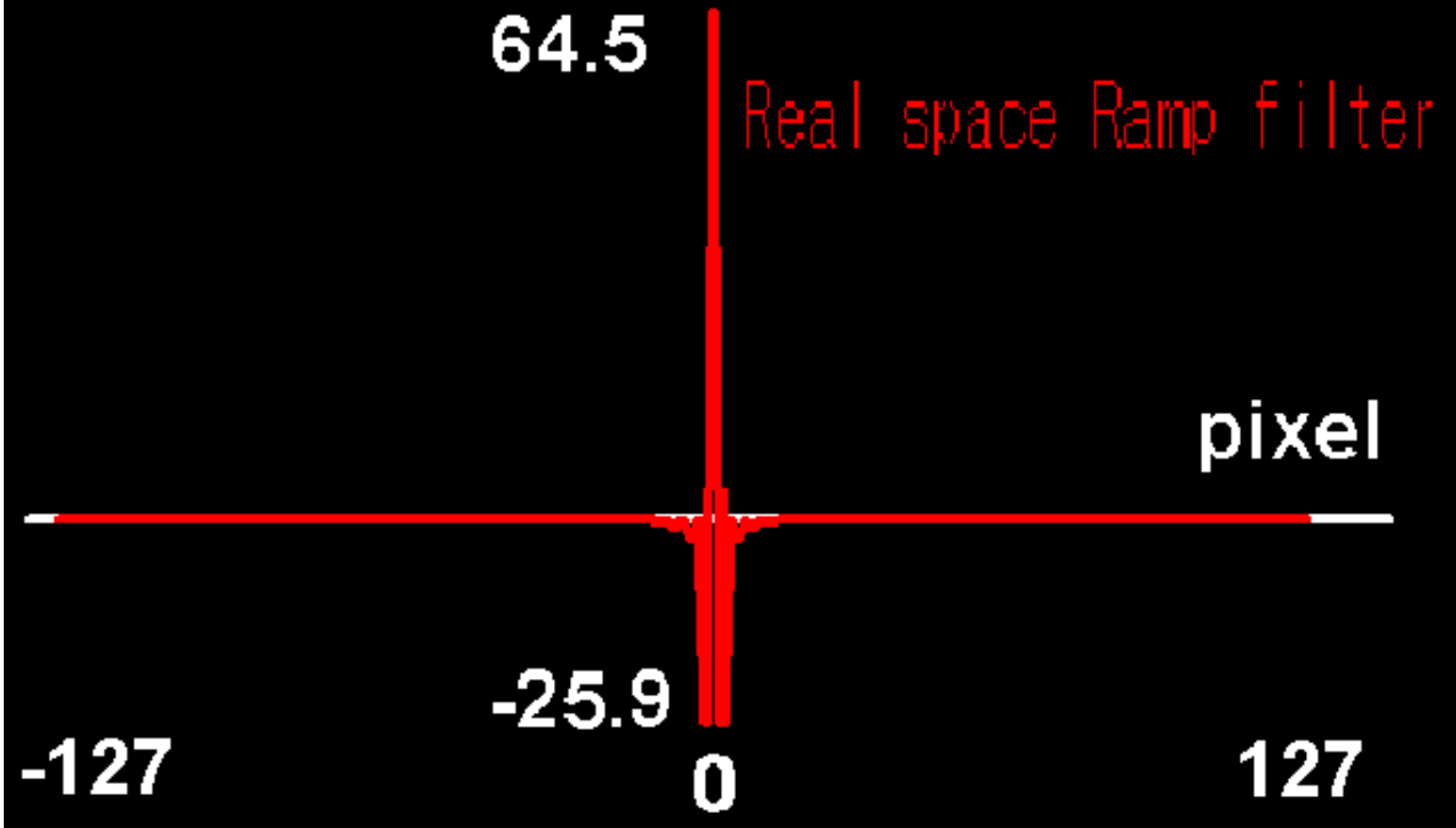
1次元の周波数成分（x に実数（sin）成分、y に虚数（cos）成分 = 0）を逆フーリエ変換して、配列 x に実空間実数成分、配列 y に実空間虚数成分を出力。

3個目の引数は、配列のデータ数。

4個目の引数が -1の場合は IFFT を実行。

実空間 Rampフィルタ の保存

```
printf("¥n¥n Save Real space Ramp filter "); scanf("%c",&yn);
GetFileName( f, 1 );        fp = fopen( f, "wt");
for(i=1; i<=128; i++){ fprintf( fp, "%.16lf ¥n", x[ i ] ) ; }
fclose(fp);
```



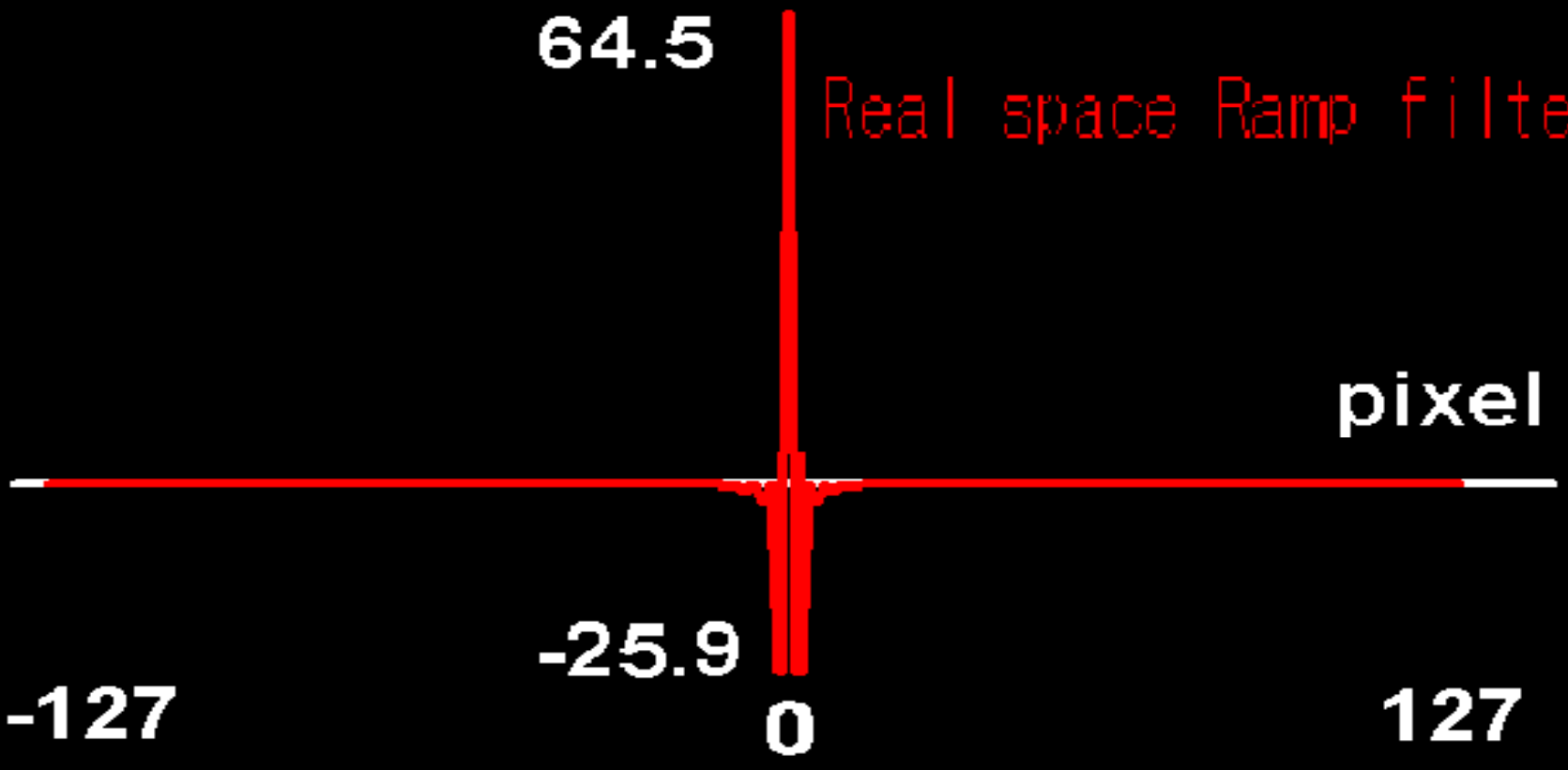
real_ramp.txt - メモ帳

ファイル(F) 編集(E) 書式(O) 表示(V)

```
64.5000000000000000
-25.9356188849910230
0.0000000000000000
-2.8794209655336709
0.0000000000000000
-1.0349257348742205
0.0000000000000000
-0.5267492473322766
0.0000000000000000
-0.3176239916379738
0.0000000000000000
-0.2117660719483017
0.0000000000000000
-0.1508830976358833
0.0000000000000000
-0.1126856963003952
```

```
//-------- Integration of Real space Ramp filter -------
for( sum=0.0 , i=1; i<=256; i++ ){ sum += RR[ i ]; }
printf("¥n¥n Integration of Real space Ramp ¥n = %.4lf ",sum ) ;
```

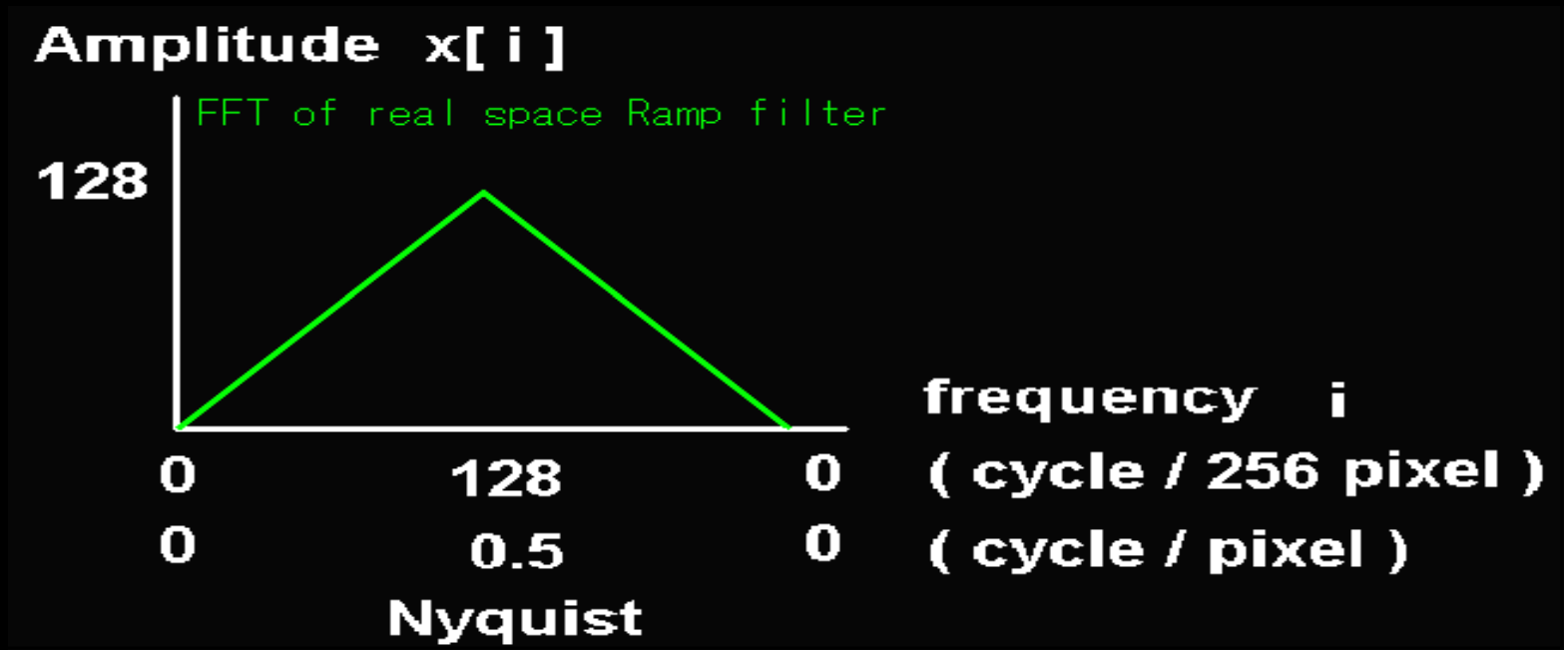
実空間フィルタの積分値は 1 になる必要がある。
（処理後にカウント総和が変化しないことが必要。）

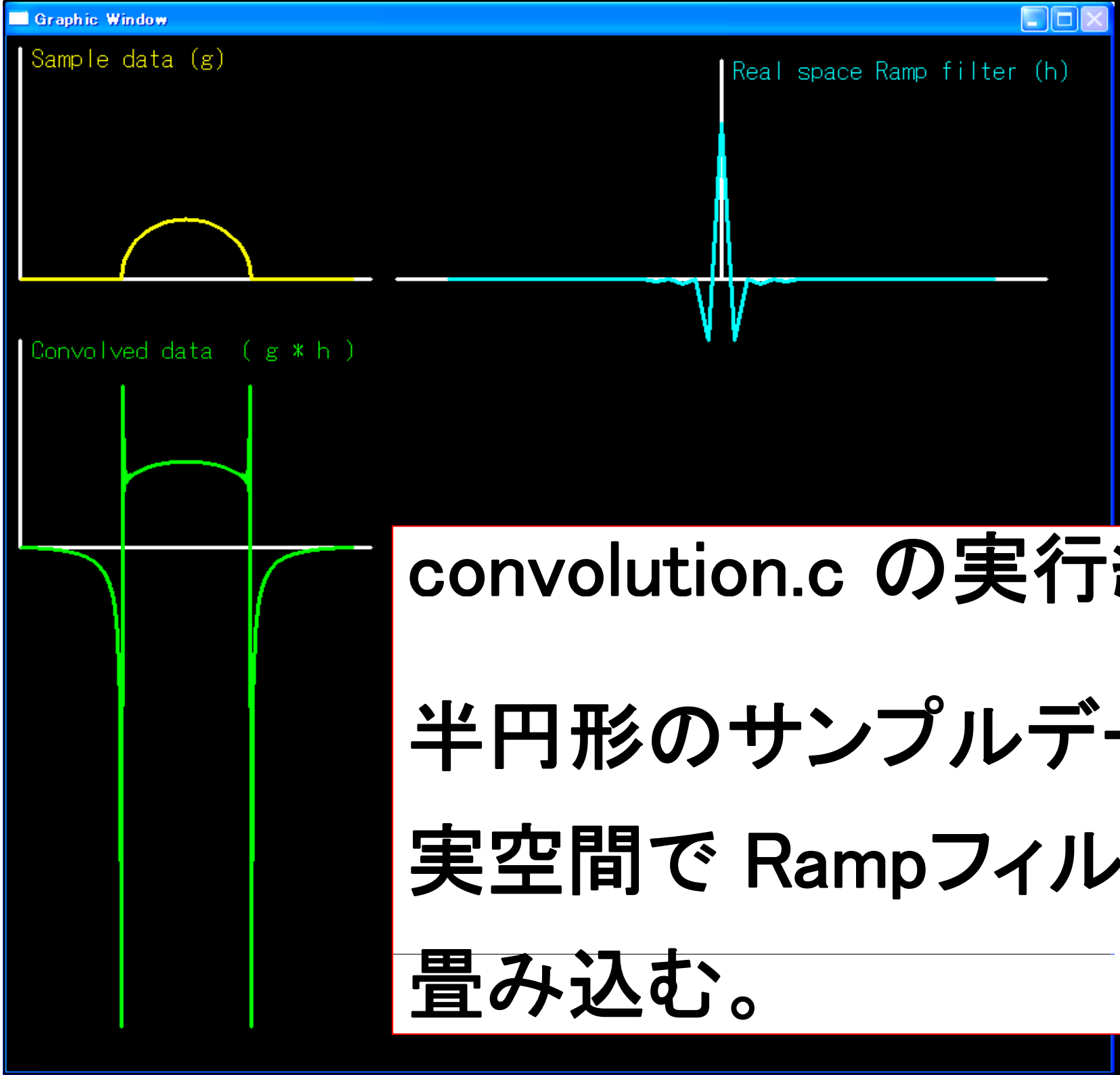


```
printf(″¥n¥n Transform Ramp filter into frequency space ″);
scanf(″%c″,&yn);
   for(i=1; i<=256; i++){ y[ i ]  = 0.0 ;  }
   FFT1D( x, y, 256, 1) ;
```

１次元高速フーリエ変換　FFT

配列 x に実空間Rampフィルタ（ y には 0）を入力し、FFT を行うと、元のRampフィルタになることを確認。





convolution.c の実行結果

半円形のサンプルデータ g に実空間で Rampフィルタ を畳み込む。

```
//------Generate sample data g[ ] :  Half Circle --------
for ( i= 1 ; i<= 77; i++){ g[i] = 0.0; }
for ( i=78 ; i<=178; i++){
  g[i] = sqrt(abs(50.0*50.0-((double)i-128)*((double)i-128)));
}
for ( i= 179; i<= 256; i++){ g[i] = 0.0; }
```

サンプル実空間データ g として、
半径 50 ピクセルの半円 g[i] を作成。

```
//---------- Load Real space Ramp filter h[ ] ---------
printf("¥n¥n Load Real space Ramp filter "); scanf("%c",&yn);
 GetFileName( f, 0 ) ;    fp = fopen( f, "rt") ;
 for(i=0; i<=127; i++){ fscanf( fp, "%lf ¥n", h + i ); }
 fclose(fp) ;
```

実数1次元配列として宣言した h[ ] に、Ramp.c で作成した実空間 Rampフィルタを読み込む。

fscanf の 3番目の引数は、読み込んだファイルデータを書き込むアドレス(ポインタ)を記述するので、h+i と書くと、配列 h[0] から h[127] に データが書き込まれる。
(単純に fscanf(fp, "%lf ¥n", &h[i]) ;　と書いても良い。)

畳み込み Convolution　I = g * h

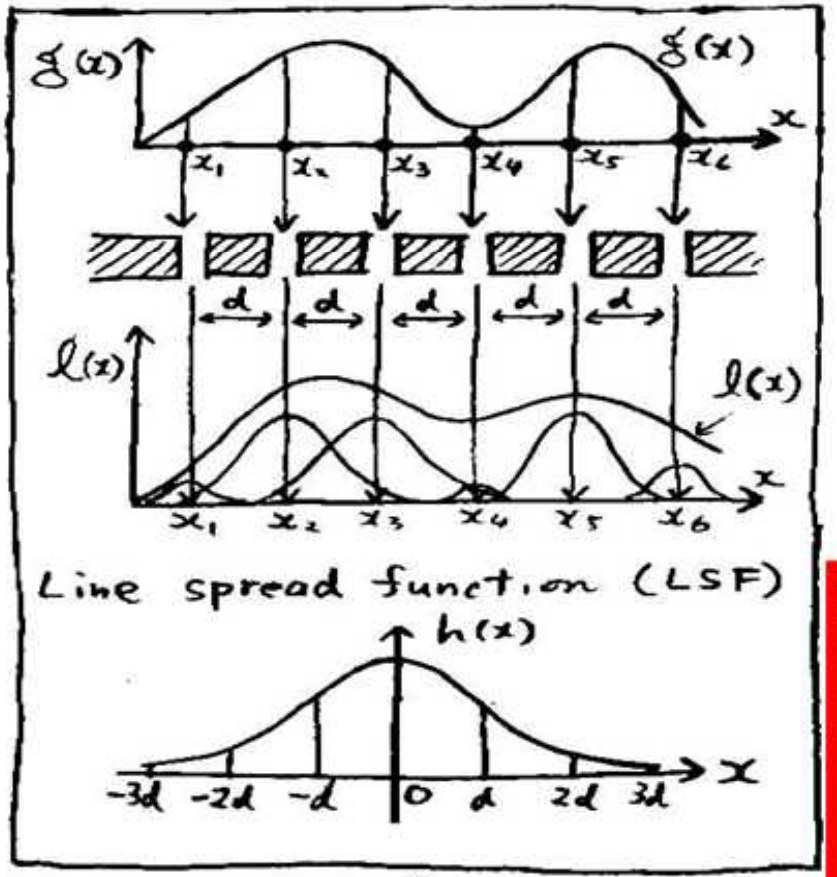
サンプルデータ g[ ] に 実空間Rampフィルタ h[ ] を畳み込んで、配列 I [ ] に書き込む。

```
//---------- Convolution h[ ] into g[ ] ----------------

printf("¥n¥n Convolution Real space Ramp filter "); scanf("%c",&yn);

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

    gh = 0.0 ;

    for(j=0; j<=127; j++){ if( i+j <= 256) gh += g[ i+j ] * h[ j ]; }

    for(j=1; j<=127; j++){ if( i-j >= 1  ) gh += g[ i-j ] * h[ j ]; }

    I[i] = gh ;
}
```

④ Convolution

スリットを通したビーム①がフィルム上で②のようなh(x)の濃度分布の像をつくるとする。

x軸に沿って明るさがg(x)である線状の被写体を，間隔dのスリットを通してフィルムに写す。



フィルムのx4の位置での濃度は

$$l(x_4) = g(x_4)h(0) + g(x_5)h(-d) + g(x_6)h(-2d) + g(x_3)h(d) + g(x_2)h(2d) + g(x_1)h(3d)$$

任意の座標x_iにおける$l(x_i)$は

$$l(x_i) = g(x_i)h(0) + g(x_{i+1})h(x_i - x_{i+1}) + g(x_{i+2})h(x_i - x_{i+2}) + \cdots + g(x_{i-1})h(x_i - x_{i-1}) + g(x_{i-2})h(x_i - x_{i-2}) + \cdots$$

$$= \sum_{n=-\infty}^{\infty} g(x_n)h(x_i - x_n)$$

$$l(x) = g(x) * h(x) \rightleftarrows L(f) = G(f)\,H(f)$$

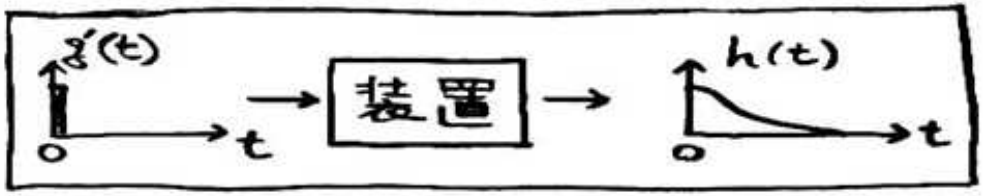
• Deconvolution

・既知の $l(x)$, $h(x)$ を使って $g(x)$ を求める式は

$$g(x) = \int \frac{L(f)}{H(f)} e^{j(2\pi f x)}\, df$$ (ボケの補正)

・既知の $g(x)$, $l(x)$ を使って $h(x)$ を求める式は

$$h(x) = \int \frac{L(f)}{G(f)} e^{j(2\pi f x)}\, df$$ (装置関数)
(インパルス応答)



(t: 時間)

ある装置にパルスを入力した時の出力 $h(t)$ を装置関数（インパルス応答）という。

装置関数 $h(t)$ の装置に，ある信号 $g(t)$ を入力すると出力 $l(t)$ は

$$l(t) = g(t) * h(t) \quad , \quad L(f) = G(f)\,H(f)$$

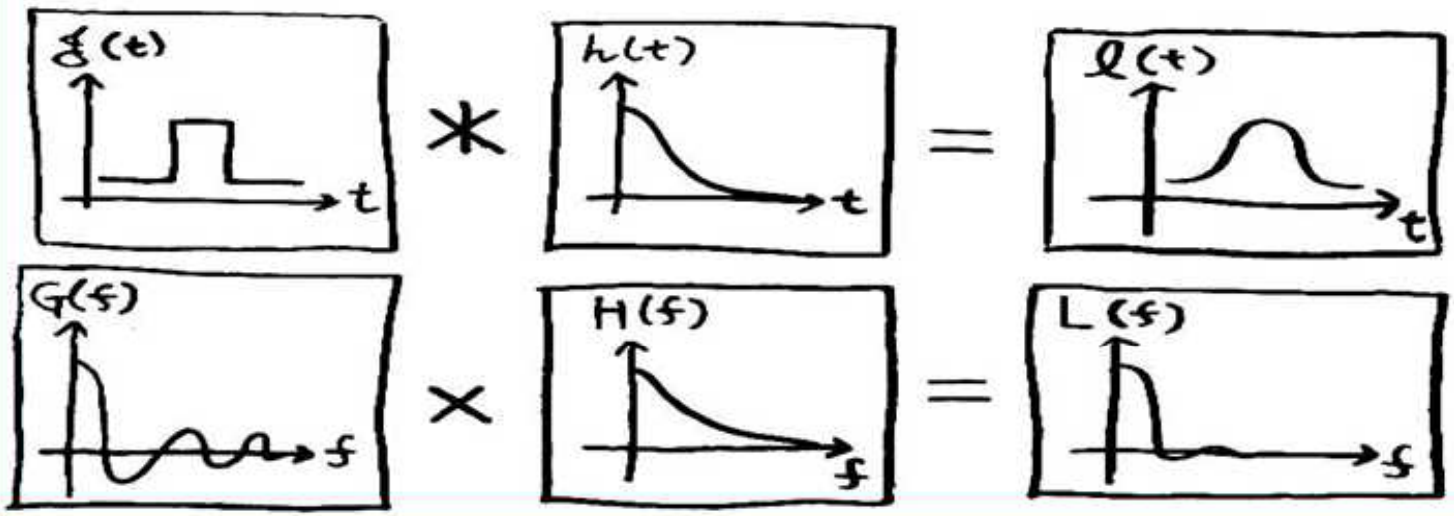
よって 上式で 入力 $g(t)$ と出力 $l(t)$ から 装置関数 $h(t)$ を求めることができる。

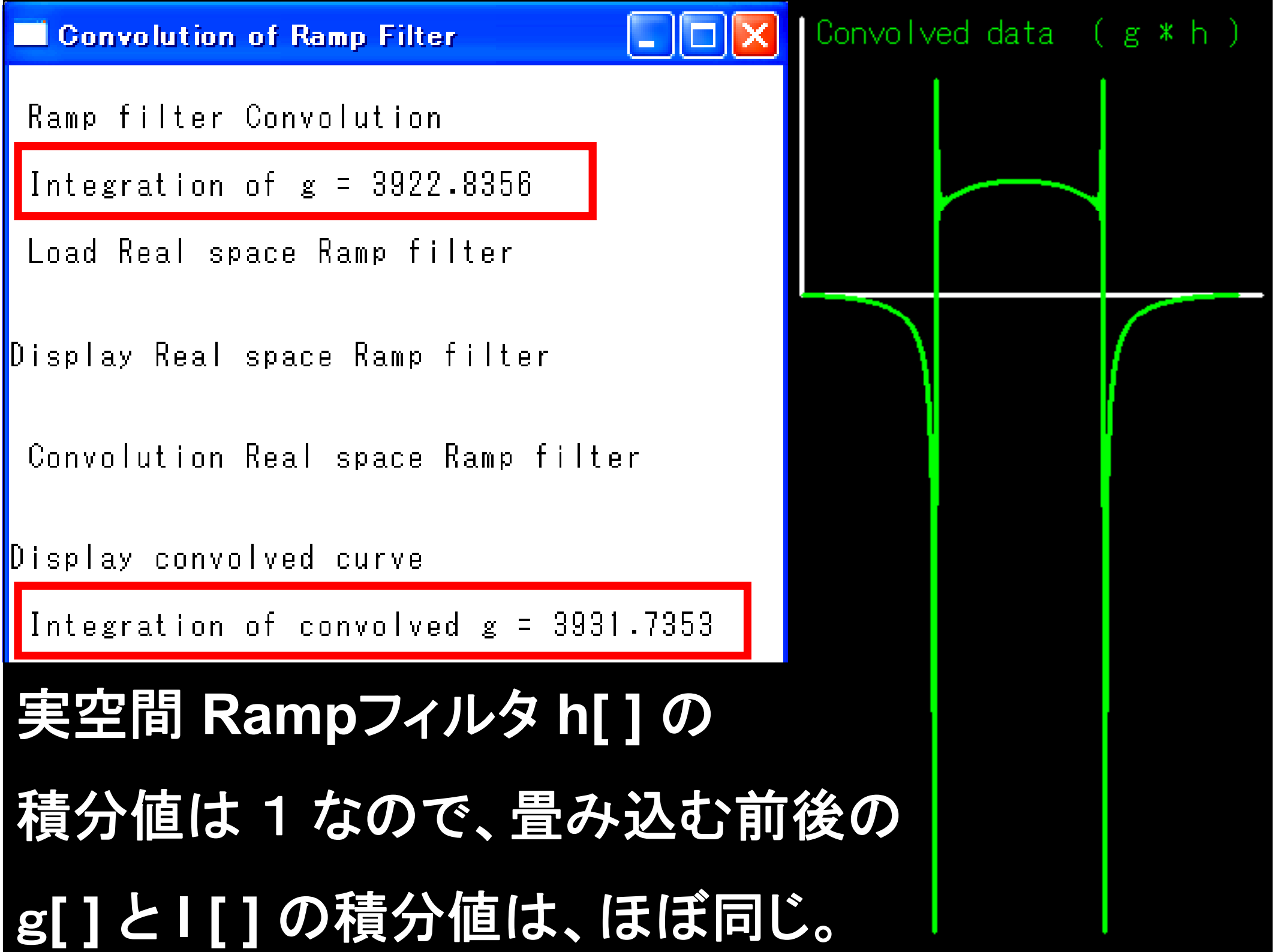
位置(x)や時間(t)の関数fが 装置の特性hによって
ボケてgになること は、

曲線fの細かい変動(高周波成分)がhによって
減衰して 大ざっぱな曲線gになることを示す。

つまり 周波数空間において、G(f)はH(f)によって
高い周波数の成分だけ減衰させられて
高周波成分の少なくなった L(f)になる。

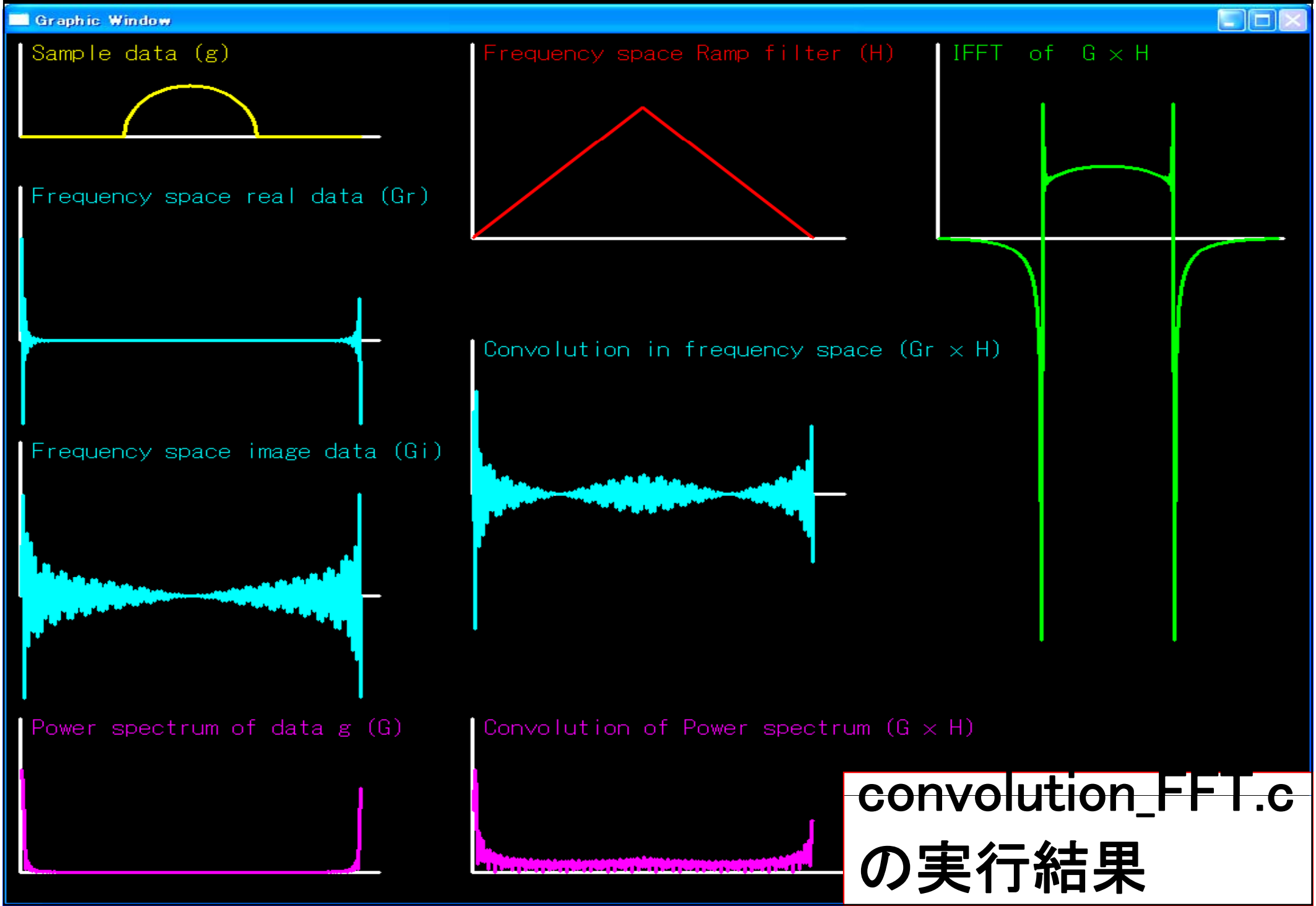
convolutionが周波数空間において乗算として
示されることは、fが大きい値のときにH(f)は小さい
値をとるfilterで、そのためG(f)にH(f)を掛けた
L(f)は高周波成分が小さくなって、そのために
gは大ざっぱな曲線にボケると解釈できる。





実空間 Rampフィルタ h[ ] の
積分値は 1 なので、畳み込む前後の
g[ ] と I [ ] の積分値は、ほぼ同じ。

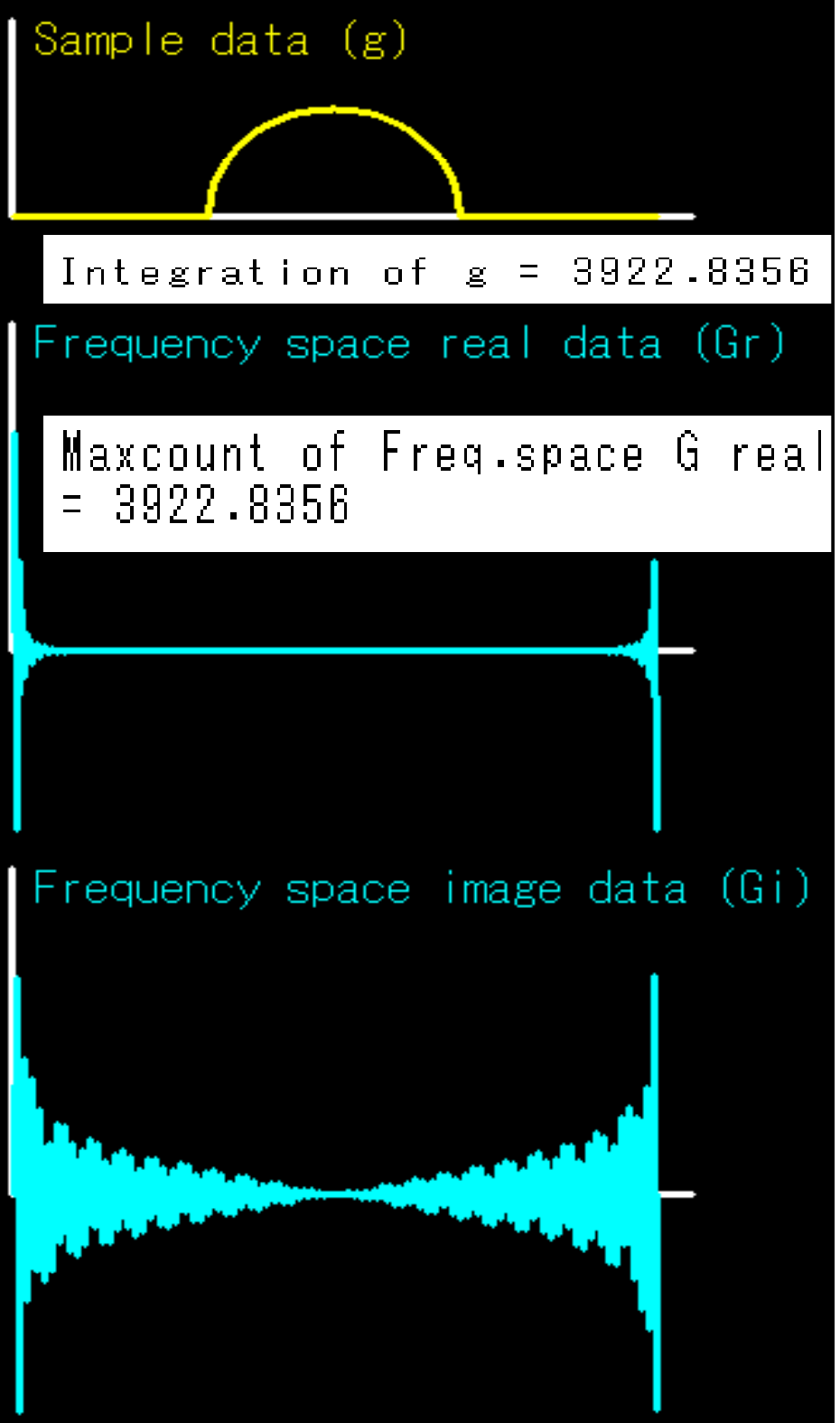
周波数空間で Rampフィルタ を かける



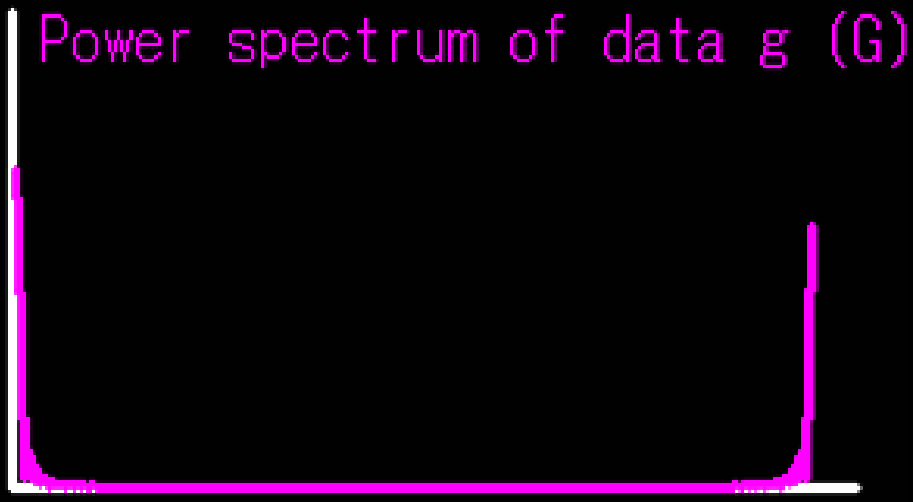
convolution_FFT.c
の実行結果

実空間 サンプルデータ g を
フーリエ変換して、
周波数空間の 実数成分 Gr、
虚数成分 Gi を表示。

実空間 サンプルデータ g の
積分値 が、周波数空間の
実数成分 Gr の最初の成分
（原点、直流成分）と同じ値に
なることを確認する。



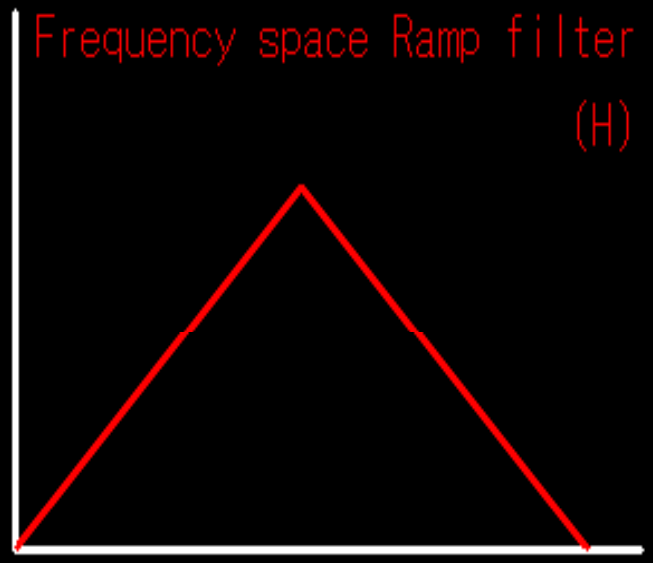
周波数空間の 実数成分 Gr、
虚数成分 Gi の二乗和の
平方根（パワースペクトル G ）
を表示。



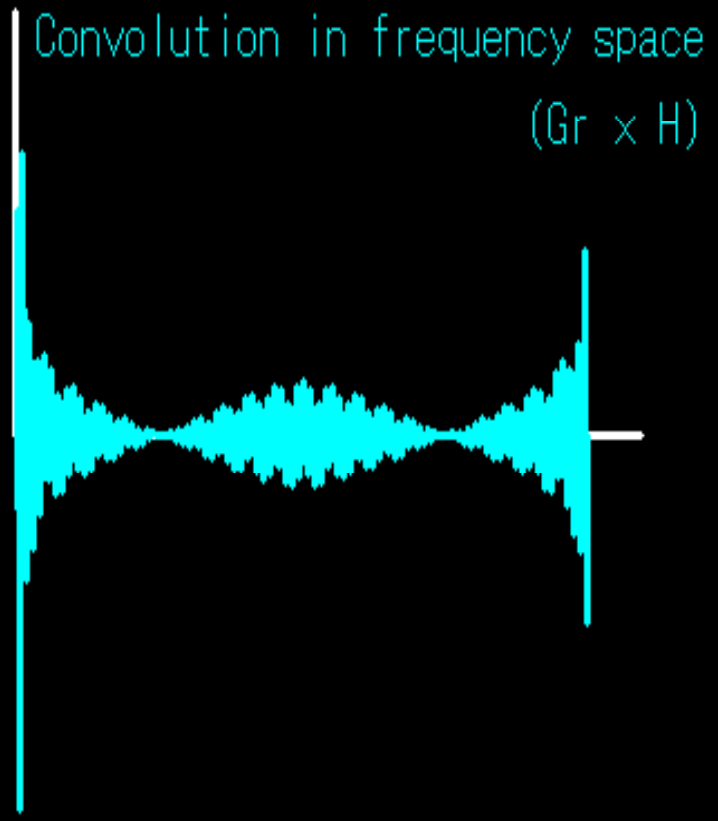
Maxcount of Power spectrum of G
= 3922.8356

実空間 サンプルデータ g の
積分値 が、周波数空間の
パワースペクトル G の最初の
成分（原点、直流成分）と同じ
値になることを確認する。

```
//------- Convolution H[ ] into G[ ] --------

for (i= 1; i<= 256; i++ ){ GHr[i] = Gr[i] * H[i] ; }

for (i= 1; i<= 256; i++ ){ GHi[i] = Gi[i] * H[i] ; }
```




```
//- Transform convolved GH into real space  -

        FFT1D( GHr, GHi, 256, -1 ) ;
```



データ g の周波数成分 Gr、Gi に、

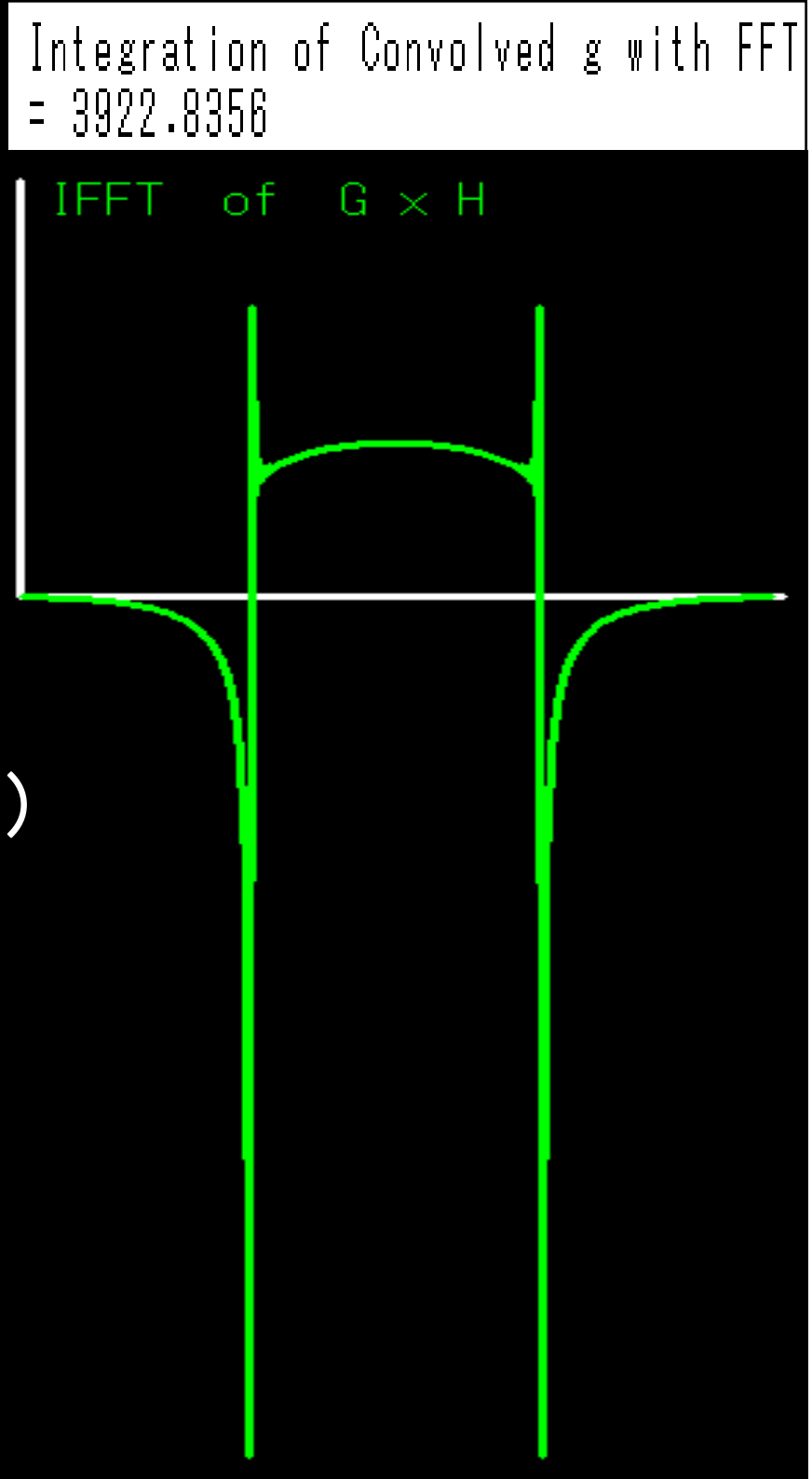
周波数空間 Rampフィルタ H をかけて、

逆フーリエ変換する。

データ g の周波数成分 Gr、Gi に、周波数空間 Rampフィルタ H をかけて、逆フーリエ変換すると、実空間で、実空間 Rampフィルタ h を g に畳み込みしたデータと同じになる。（ **G x H と g * h は等価演算** ）

積分値が 元データ g と同じことを確認する。

（カウント総和は保たれる。）



課題5　プログラム Ramp.c を改良して
実空間 Shepp&Loganフィルタを計算するプログラム
SheppLogan.c を作って下さい。

周波数空間Shepp&Logan の式

$$H(f_r) = \left| \frac{f_0}{\pi} \sin\left(\frac{2\pi f_r}{f_0}\right) \right| \mathrm{sinc}^2\left(\frac{f_r}{f_0}\right)$$

$$\left(\mathrm{sinc}\ x = \frac{\sin(\pi x)}{\pi x} \right)$$

絶対値を計算する関数は　abs( )　　y = | x |　→　　y = abs(x);

階乗を計算する関数は　pow(　,　)　　$y = x^2$ → y = pow(x,2.0);

周波数空間Shepp&Logan の式の f0 は高周波遮断周波数。
f0 をプログラムで設定できるように
printf("¥n¥n Cut off freq. (0～0.5) = "); scanf("%lf",&CutOff);
を入れてください。

ここで CutOff の単位は cycles/pixel 。 Shepp&Loganの式に
使われる f0 の単位は cycles / 画像の1辺長 = cycles / 256 pixel
なので f0 = 256.0 * CutOff ; と変換してから式に代入してください。

作成したプログラムとフィルタのグラフ画像を
メールに添付して提出してください。