フィルタリング
その１

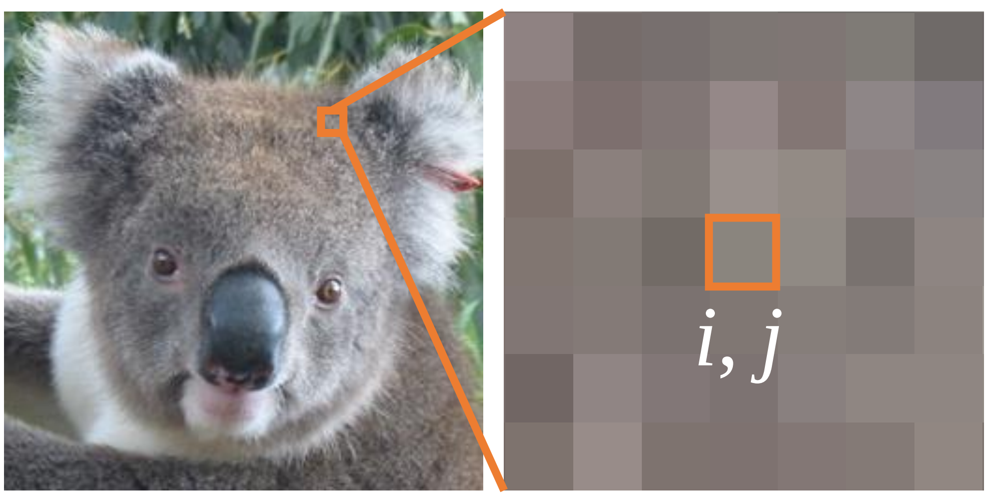
フィルタと効率的な計算

ある画素周辺での平均値の計算

Box フィルタ・移動平均

フィルタリング

- 処理対象の画素の
周辺画素を用いた計算
- 簡単な例では，平均値を求めるなど
- 特定の周波数帯域（模様）を取り出す



平均値 → 出力

$$y(i,j) = \frac{1}{N}\sum_{u,v} x(i+u, j+v)$$

u, v は中心座標からの変位

プログラム（1／3）

```
I = im2double(imread('../images/koala_small.png'));
Iycc = rgb2ycbcr( I );
Y = Iycc(:,:,1);

iy = 100;     ix = 100;   % 対象画素
r = 5;  % 窓半径

Y_local = Y( iy-r:iy+r, ix-r:ix+r );

% 処理箇所の表示
figure(1), imshow( Y );
hold on;
rectangle('Position', [ix-r,iy-r,2*r+1,2*r+1], ...
            'EdgeColor', 'red', 'LineWidth', 3 );
hold off;

% 拡大して表示
figure(2), imshow( imresize( Y_local, 4, 'nearest' ) );
% MATLAB の場合は, nearest の代わりに box
```

対象画素の周辺領域を切り出し，表示してみる

プログラム（２／３）

- １画素のみの結果を，出力画像に対して表示してみる．

- ```
  % 出力画像
  Y_out = zeros( size( Y ) );

  % 対象画素
  mu = mean( Y_local(:) );

  % 出力画像への書き込み
  Y_out( iy, ix ) = mu;


  % 出力画像の表示
  figure(3), imshow( Y_out );
  ```
  ```

プログラム（３／３）

- 全画素で表示するとどうか？

```
% 全画素での実行
[sy,sx] = size( Y );

for ix = 1+r:sx-r
  for iy = 1+r:sy-r

    Y_local = Y( iy-r:iy+r, ix-r:ix+r );
    mu = mean( Y_local(:) );
    Y_out( iy, ix ) = mu;
  end
end


% 出力画像の表示
figure(3), imshow( Y_out ); drawnow;
```

もし，１列ごとに
結果を表示したい場合は，
imshow を end 内に移動
かなりの時間がかかるので注意

補足：周波数帯域成分（１／３）

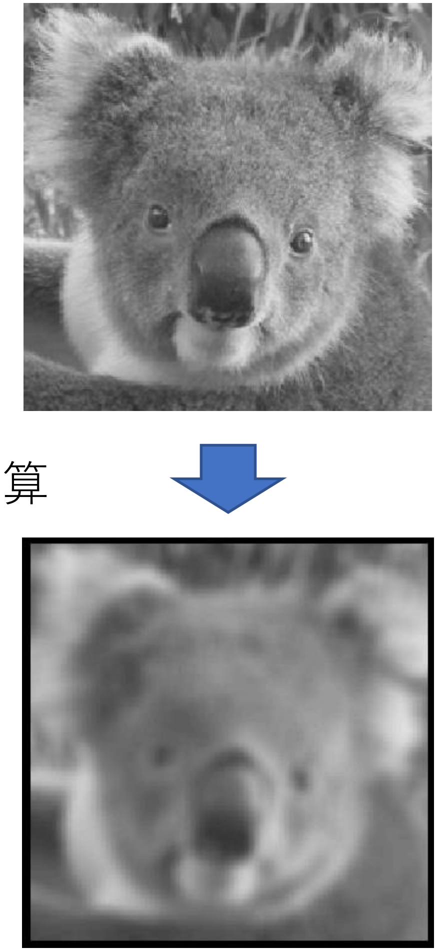
- 画像内での
「大まかな色や輝度の変化」（構造成分）と
「細かな模様の変化」（詳細/模様成分）は

- 信号処理でいう
「**低周波**数帯域成分」と
「**高周波**数帯域成分」を表している

- 平均値の計算は,
低周波数帯域成分を抽出でき,

大まかな色や輝度の変化を示す
結果画像が得られる.

周辺画素の
平均値の計算



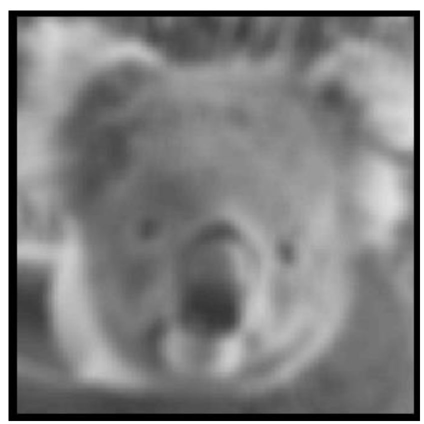
補足：周波数帯域成分（２／３）

- **低周波**数帯域成分を抽出した際に,
  失われた
  高周波数帯域成分は・・・

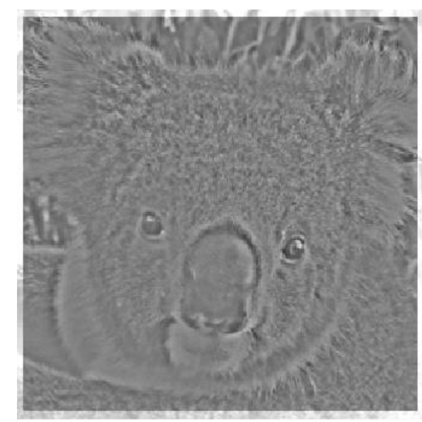
  原画像から結果画像を引くと得られる.

- **高周波**数帯域成分　は　模様　を表す.

原画像



低周波



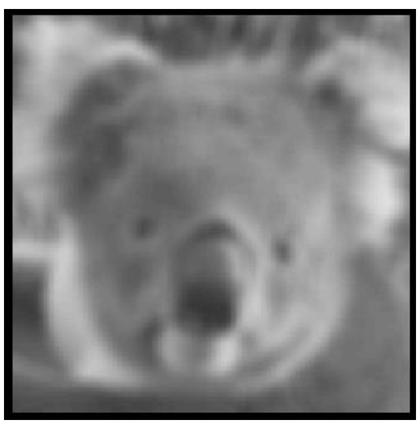
高周波

補足：周波数帯域成分（３／３）

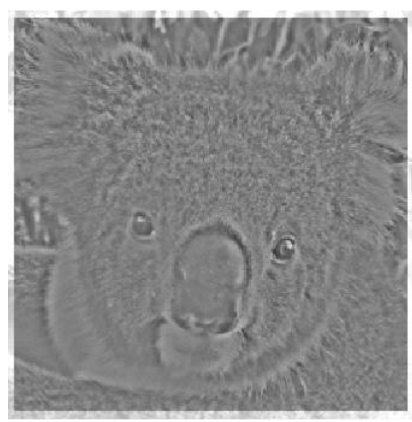
- 数式同様に，移項できる
- プログラムでは
  - ```
    Y_struct = Y_out;
    Y_detail = Y - Y_struct;

    figure(10),  imshow( Y_detail + 0.5 ); % 高周波を表示
    figure(11),  imshow( Y_struct + Y_detail ); % 再合成
    ```

Y_detail は引き算の結果得られたので，負の値を含む．
imshow は負の値を全て 0 として丸め込むため，
0 を灰色として，
負の値を黒，正の値を白として表示する．



低周波



高周波

再合成結果
    ```

フィルタカーネルを用いた表現

- 平均値の計算をフィルタ（相関演算）形式で表すと

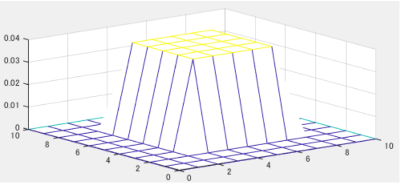
$$y(i,j) = \frac{1}{N}\sum_{u,v} x(i+u, j+v)$$

和記号の内側へ移動

$$y(i,j) = \sum_{u,v} \frac{1}{N}\, x(i+u, j+v)$$

係数を汎用的に
記号で表し直す

$$y(i,j) = \sum_{u,v} k(u,v)\; x(i+u, j+v), \qquad k(u,v) = \frac{1}{N}$$



フィルタ形状を三次元プロットすると，箱型に見えるので，ボックスフィルタと呼ばれる

すべてのフィルタ係数が同一で合計値が1となる

プログラム

```
K = ones( 2*r+1, 2*r+1 );  % 全ての要素が 1 の配列を生成
K = K / sum( K(:) ); % 合計値で割る

Y_out2 = zeros( size( Y ) );

for ix = 1+r:sx-r
  for iy = 1+r:sy-r

    Y_local = Y( iy-r:iy+r, ix-r:ix+r );
    KY = K .* Y_local;
    mu = sum( KY(:) );

    Y_out2( iy, ix ) = mu;
  end
end

figure(4),  imshow( Y_out2 );
```

自作のフィルタ演算の問題点

- MATLAB や Python などの
  中間言語（インタープリタ）の
  for 文は遅い．

- 高速化が必要な箇所を
  C++ などの言語に置き換えて
  コーディングした関数が用意されている．

- `filter2( フィルタ, 画像, オプション );`
- `imfilter( 画像, フィルタ, オプション );`

`imfilter` のほうが使い勝手は良いが，
今回はより基礎的な `filter2` を使ってみる．

プログラム

```
K = ones( 2*r+1, 2*r+1 );
K = K / sum( K(:) );

Y_out3 = filter2( K, Y, 'same' );

figure(5),  imshow( Y_out3 );
```

補足　相関演算と畳み込み演算（１／３）

- フィルタリングには，２種類の計算がある．
  - 相関演算
    correlation
    $$y(i,j)=\sum_{u,v}k(u,v)\ x(i+u,j+v)$$
  - 畳み込み演算
    convolution
    $$y(i,j)=\sum_{u,v}k(u,v)\ x(i-u,j-v)$$
  - フィルタリングとは
    画像処理では，相関演算を指す
  - 計算が同一となる条件

    フィルタを１８０度回転させても
    同じ係数配置なら，
    ２つの演算は同一
    - 例えば，ボックスフィルタでは，
      ２つの演算は同一となる．

$$y(i,j)=\sum_{u,v}\frac{1}{N}\ x(i+u,j+v)$$

$\Updownarrow$ 同一

足し算の
順序が変わる

$$y(i,j)=\sum_{u,v}\frac{1}{N}\ x(i-u,j-v)$$

補足　相関演算と畳み込み演算（２／３）

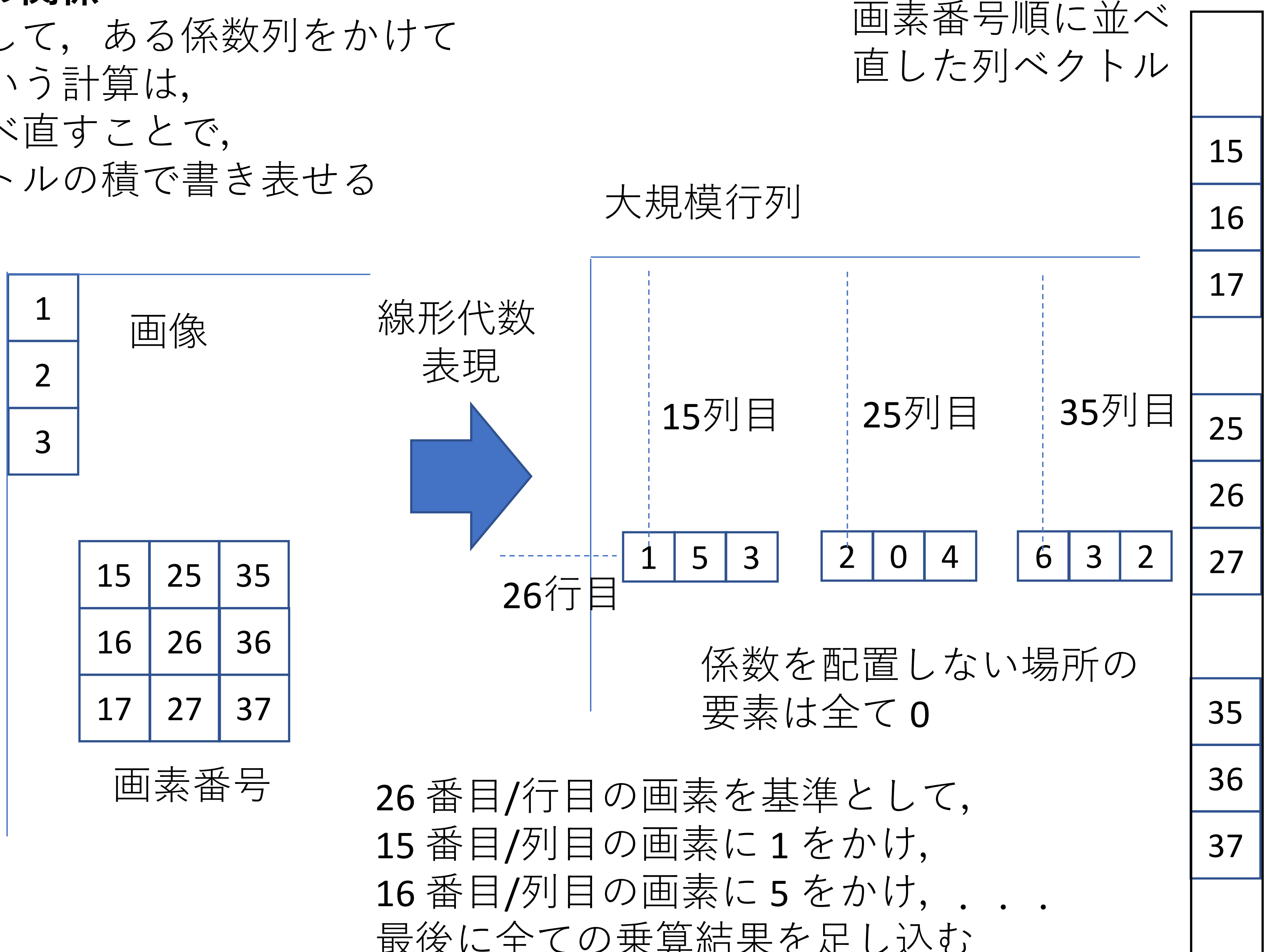
線形代数との関係
画素値に対して，ある係数列をかけて
足し込むという計算は，
数値列を並べ直すことで，
行列とベクトルの積で書き表せる

1	2	6
5	0	3
3	4	2

係数列

15	25	35
16	26	36
17	27	37

画素番号



26 番目/行目の画素を基準として，
15 番目/列目の画素に 1 をかけ，
16 番目/列目の画素に 5 をかけ，．．．
最後に全ての乗算結果を足し込む

補足　相関演算と畳み込み演算（３／３）

- ２つの演算の関係性
  - 行列的な関係：
    線形フィルタは，フィルタを行列として表せる.

    このとき，相関演算と畳み込みの片方を $\mathbf{Kx}$ と表すと，
    もう片方は転置で表される $\mathbf{K}^{\mathrm{T}}\mathbf{x}$

  - 信号処理的な関係：
    - 相関演算は，周辺の情報をある１点に集約させる
    - 畳み込み演算は，１点の情報を周辺に拡散させる

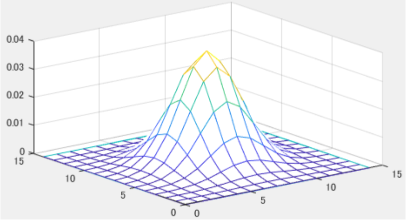
  - 逆演算 ではないが，それに近い働きをする.

    - 確認用プログラム：畳み込み演算の後，相関演算

```
K = magic( 3 )
I = zeros( 5, 5 );  I(3,3) = 1
J = conv2( I, K, 'same')
L = filter2( K, J, 'same' )
figure(1), imagesc( L );
```

ガウシアンフィルタ

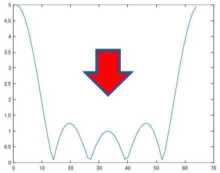
ガウシアンフィルタ



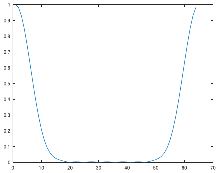
- 最も基礎的な平滑化フィルタ
  - 釣鐘型の係数値と係数配置を持つ
  - ボックスフィルタ（平均値の計算）
    では，各画素に均等なフィルタ係数（重み）
    を乗じていた

    一方，ガウシアンフィルタでは，
    処理対象の画素から離れるほど，施す重みを小さくする．
  - よく用いられる理由
    - 画像の画素値は処理対象画素から離れるに従い，
      色が変わる可能性が高い（かもしれない）．
    - ボックスフィルタでは中周波数帯域に
      漏れが生じるが，
      ガウスフィルタでは漏れがない

両端が低周波
中央が高周波



ボックス
フィルタ

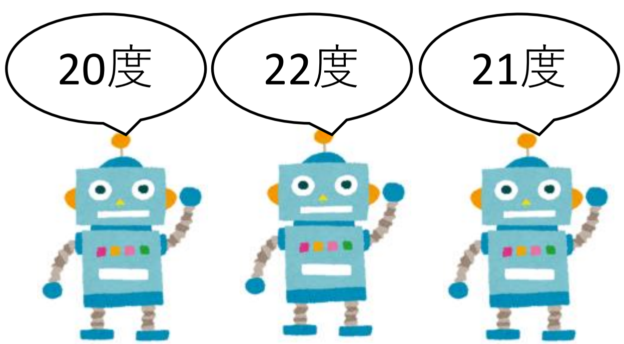


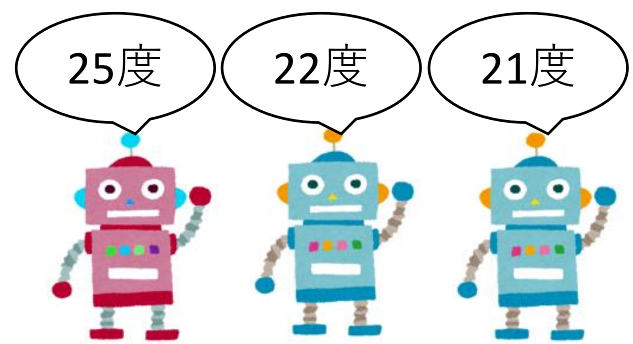
ガウシアン
フィルタ

加重平均（１／３）

- 信頼度が低い情報を切り捨てつつ
  平均を求める方法
- 問題
  青色のロボットは **100%** 正常に機能しており，
  判断内容は正しいと考えられる．
  一方，他の色に変色したロボットは，
  正常とは言えず，信用できるのは **30%** 程度である．

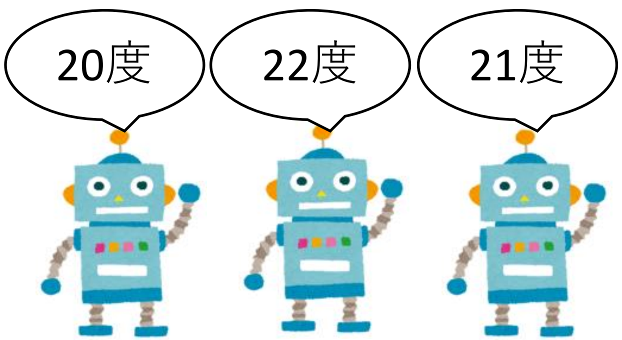
  今，ロボットに部屋の気温を答えさせた．
  部屋の気温はどの程度と思われるか，答えよ．





加重平均（２／３）

- 平均値の計算時に用いていた正規化用の定数 **1/N** を信頼度に置き換える.



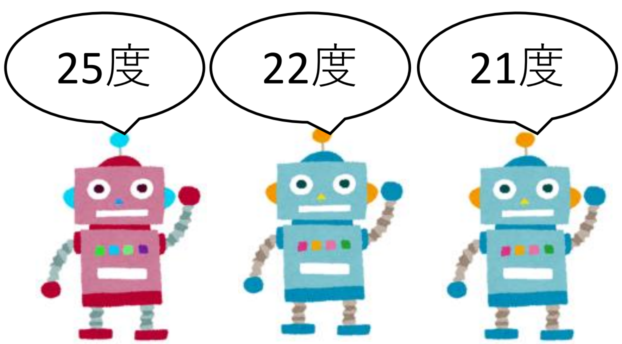
$$\frac{100 \times 25 + 100 \times 22 + 100 \times 21}{100 + 100 + 100} = 21$$

同意

$$= \frac{100 \times 25 + 100 \times 22 + 100 \times 21}{300}$$

$$= \frac{1}{3} \times 25 + \frac{1}{3} \times 22 + \frac{1}{3} \times 21$$

係数を予め正規化したバージョン



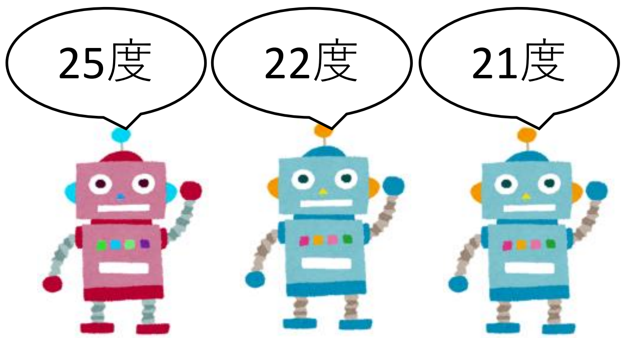
$$\frac{30 \times 25 + 100 \times 22 + 100 \times 21}{30 + 100 + 100} = 21.9565$$

同意

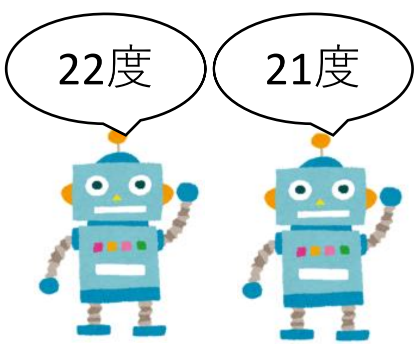
$$= 0.1304 \times 25 + 0.4348 \times 22 + 0.4348 \times 21$$

加重平均（３／３）

- **100%** 信用できない（**0%** 信用できる）場合
- 重みを 0 にすれば,
  その情報を用いていないことと同意となる.



$$\frac{0 \times 25 + 100 \times 22 + 100 \times 21}{0 + 100 + 100} = 21.5$$



$$\frac{100 \times 22 + 100 \times 21}{100 + 100} = 21.5$$

- ただし，信頼できる／できないを,
  明確に定めることは難しいため，重みを用いて,
  曖昧性を加味できるようにしている.

プログラミング（1／2）

- ガウス関数を扱う練習を兼ねて

$$\widetilde{k}(u,v) = \exp\left(-\frac{u^2+v^2}{2\sigma^2}\right) \Rightarrow k(u,v) = \frac{1}{\sum_{p,q}\widetilde{k}(p,q)}\widetilde{k}(u,v)$$

合計値が 1 となるように
正規化して用いる

- ```
ss = 2; % 標準偏差

[CX,CY] = meshgrid(-r:r, -r:r);
D2 = CX.^2 + CY.^2;
Kg = exp(- D2 / (2*ss^2));
Kg = Kg / sum(Kg(:));

figure(6), meshz(Kg);

Y_out4 = filter2(Kg, Y, 'same');
figure(7), imshow(Y_out4);
```
```

プログラミング（2／2）

- 専用のフィルタ生成関数を用いる場合

```
% フィルタサイズ
wnd = 2*round( 4*ss ) + 1;
% 標準偏差の 3倍 ～ 4倍 あると,
% 端の値が 0 に十分に近づく

% fspecial 関数を用いると,
% いくつかの代表的なフィルタを生成できる
Kg = fspecial( 'gaussian', [wnd,wnd], ss );

figure(8), meshz( Kg );
```

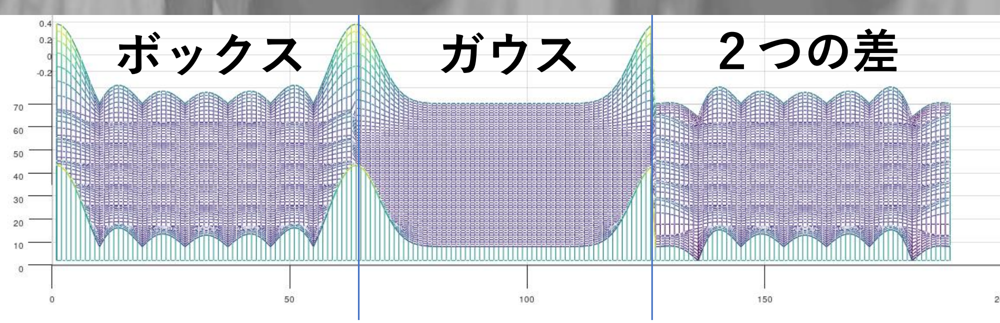
結果の比較

- 画像は，差のわかりやすい `balloon.png` に変更
- ボックスフィルタ（平均値）とガウシアンフィルタで，平滑化度合いを同程度にした場合の比較
  - ボックスフィルタの半径は `r = 3`
  - ガウシアンフィルタの標準偏差は `ss = 2.1`

目や口のあたりで違いが生じる

ボックスでは輪郭が四角形に近づいている

フィルタのスペクトラム（作用する周波数）



両端が低周波
中央が高周波

ボックスでは高周波数が漏れ出している

効率的な計算方法１

可分離型フィルタ

二次元フィルタを
水平・垂直の一次元フィルタの組に
分解する方法

効率的な計算法１（１／３）

- 二次元フィルタの計算量
  - 画像サイズ $Nh \times Nv$, フィルタサイズ $Mh \times Mv$ であるとき

    $(Nh \times Nv) \times (Mh \times Mv)$ であり，膨大.

- 可分離
  - 係数配置に対称性をもつフィルタの場合,
    垂直・水平方向の一次元フィルタの組に分解できる
    可能性がある.
  - ボックスフィルタとガウシアンフィルタは分解できる.

$$\frac{1}{9}\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix} \Rightarrow \frac{1}{3}\begin{pmatrix}1\\ 1\\ 1\end{pmatrix} \times \frac{1}{3}\begin{pmatrix}1 & 1 & 1\end{pmatrix}$$

分解したフィルタを用いて,
垂直方向（もしくは水平方向）にフィルタリングした後,
もう一方の方向にフィルタリングを行う

効率的な計算法１（２／３）

- 分離フィルタの計算量
  - ボックスフィルタやガウシアンフィルタの場合,
    画像サイズ $Nh \times Nv$, フィルタサイズ $Mh \times Mv$ であるとき

    $(Nh \times Nv) \times Mh + (Nh \times Nv) \times Mv$

    $= (Nh \times Nv) \times (Mh + Mv)$

    ノーマルな方法（Naïve な方法）では,
    $(Nh \times Nv) \times (Mh \times Mv)$ であったので,

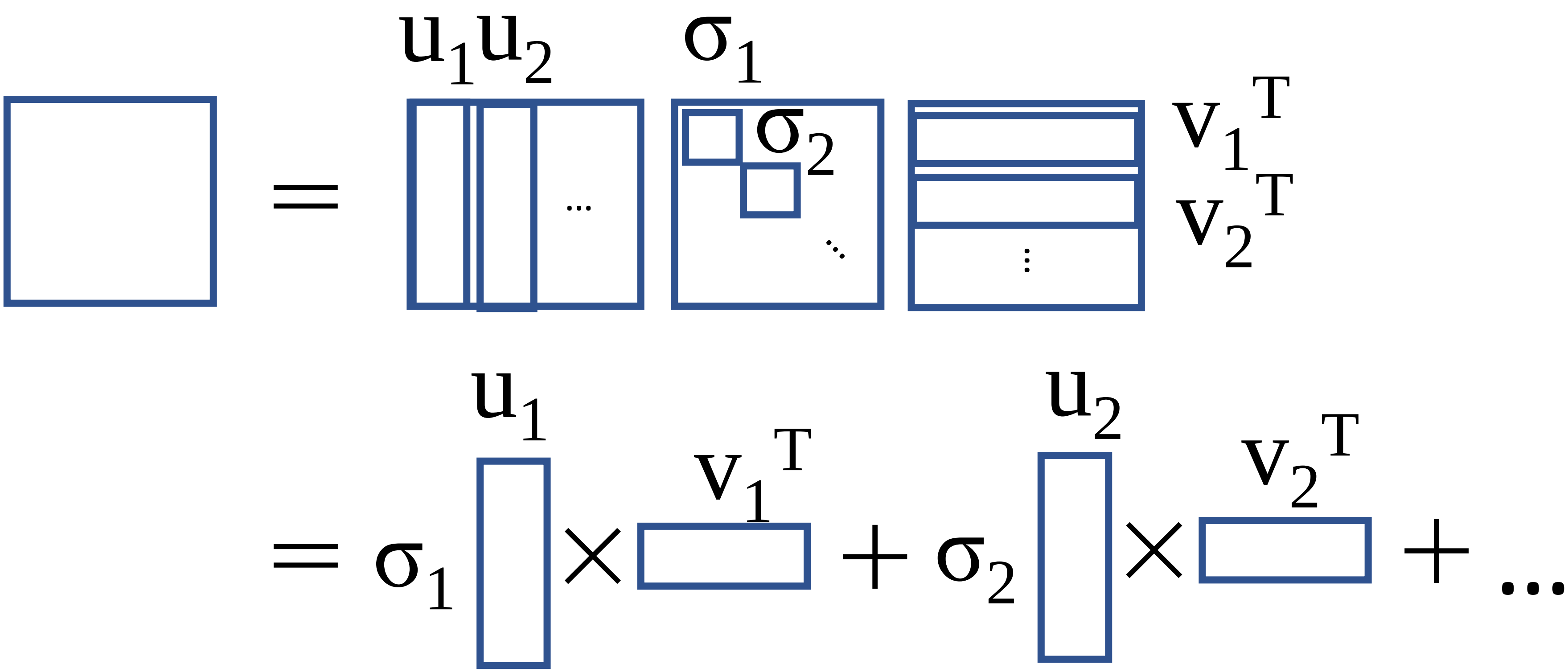
    $(Mh + Mv) << (Mh \times Mv)$ 分高速となる

  - 更に, メモリアクセスも連続となるため,
    高速に動作する.

効率的な計算法１（３／３）

- 一般的な二次元フィルタの分解方法
  - 行列の特異値分解を用いる
    - 行列分解法

$$\mathbf{K} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$
$$= \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$



プログラミング（１／２）

- 二次元ガウシアンフィルタの特異値分解
- ```
  [U,S,V] = svd( Kg, 'econ' );

  diag( S )
  % S の対角値を表示,
  % 第一特異値以外は 0 であることがわかる

  s1 = S(1,1);
  u1 = U(:,1);
  v1 = V(:,1); % ガウシアンでは u1 == v1

  u1 = sqrt(s1) * u1; %  u1 = s1 * u1;
  v1 = sqrt(s1) * v1; %  v1 = v1;  でも良い

  figure(8),  plot( u1 ); % １次元フィルタの確認
  figure(9),  meshz( u1 * v1' ); % ２次元フィルタの確認
  ```
  ```

プログラミング（2／2）

- 水平と垂直方向への一次元フィルタリング
- tic % 実行時間の計測

```
Yh  = filter2( u1 ,  Y, 'same' );
Yhv = filter2( v1', Yh, 'same' );
toc

% 順序を入れ替えても良い
% Yv  = filter2( v1',  Y, 'same' );
% Yvh = filter2( u1 , Yv, 'same' );

figure(10),  imshow( Yhv );

% なお，filter2 も imfilter も関数内部で
% 分離フィルタを生成しているため，二次元フィルタを入力しても，
% 今回の方法と速度に違いはない．
% conv2 は分離フィルタを用いないため，2D フィルタでは遅くなる
tic
conv2( Y, Kg, 'same' );
toc % Kg が点対称のときは conv2 == filter2
```

効率的な計算方法２

フーリエ変換を介しての計算

効率的な計算法２（１／２）

- 線形フィルタ：相関演算と畳み込み演算は
フーリエ変換を介して効率よく計算できる.
  - 畳み込み演算は
  フーリエ変換領域で，画素ごとの乗算　として表せる.
  - 相関演算は
  虚数部分の符号を反転してからの乗算　として表せる.

$$y(i,j) = \sum_{u,v} k(u,v)\ x(i+u, j+v)$$

$$F(p,q) = K(p,q)X(p,q)$$

$$y(i,j) = \sum_{u,v} k(u,v)\ x(i-u, j-v)$$

$$F(p,q) = \overline{K(p,q)}X(p,q)$$

$$\overline{a+jb} = a - jb$$

画像空間領域

フーリエ変換領域

効率的な計算法２（２／２）

- 計算量
  - 画像サイズに依存する．
  - 画像サイズ $Nh \times Nv$ であるとき，
    フィルタサイズを同サイズに拡張，周りに 0 を埋め込む
  - 画像とフィルタを，それぞれ，フーリエ変換する．
    二次元のフーリエ変換の計算量は，一つの画像につき

    $$Nh \times O(Nv \log(Nv)) + Nv \times O(Nh \log(Nh))$$

  - フーリエ変換できてしまえば，全画素での乗算で
    $Nh \times Nv$
  - 最後に逆フーリエ変換を行う．計算量は順変換と同じ
  - **非可分離のフィルタやサイズが極めて大きなフィルタを用いる場合は，フーリエ変換を介したほうが良い．**

プログラミング

- 高速フーリエ変換を自分で書くのは難しいため，用意された関数を用いる．
  - 一般には FFTW と呼ばれるライブラリを用いる．

-
```
tic
[sy,sx,sc] = size( Y );

FY = fft2( Y );  % 画像のフーリエ変換
FK = conj(psf2otf( Kg, [sy,sx] )); % フィルタのフーリエ変換

FY_out = FY .* FK;  % 画素ごとの乗算

Y_out5 = ifft2( FY_out );  % 逆フーリエ変換

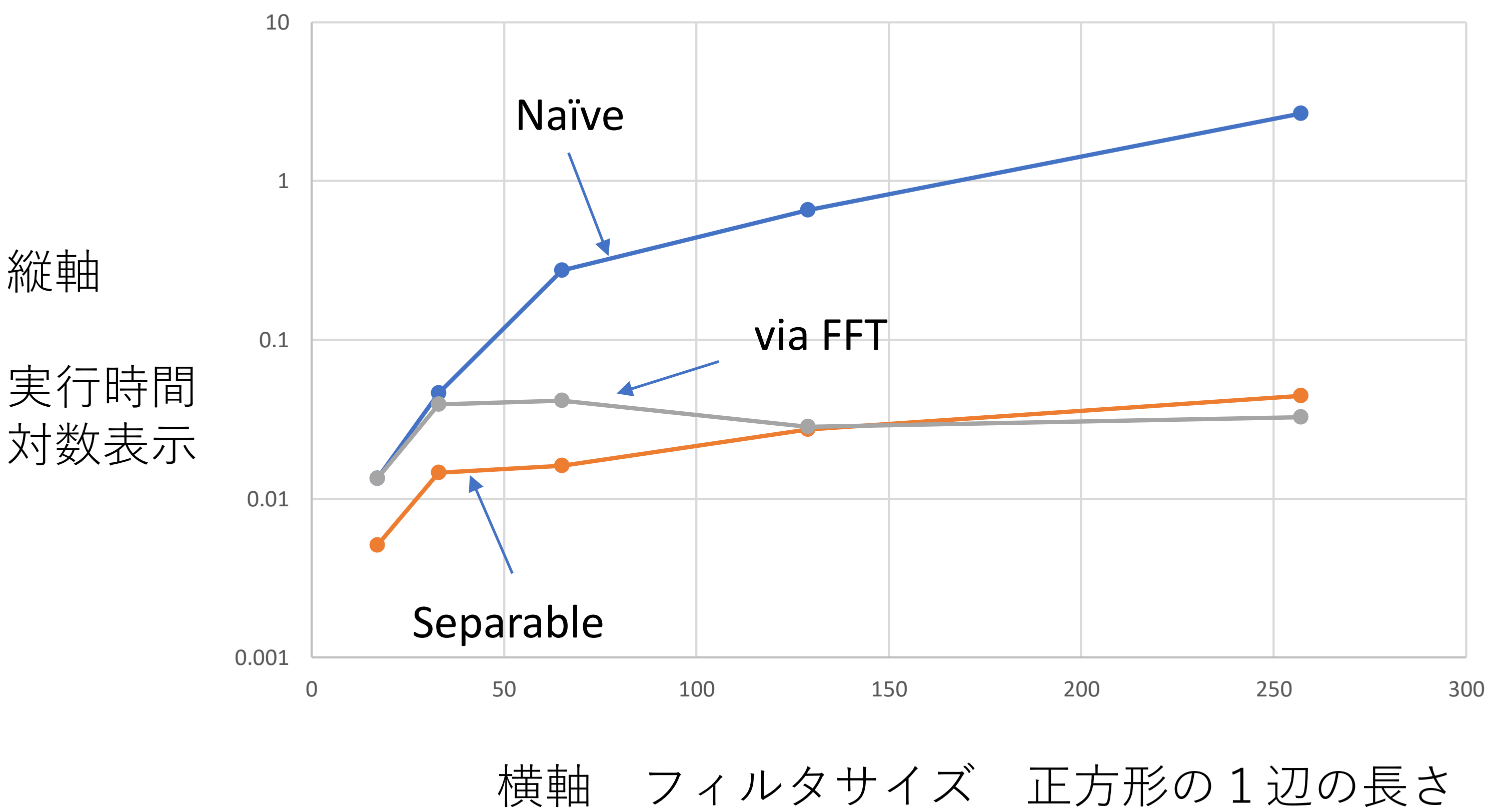
Y_out5 = real( Y_out5 );  % 虚数部分を除去する
toc

figure(11),  imshow( Y_out5 );
```

実行時間の比較

実行時間

- 画像サイズ　480v ✕ 640h
  - 可分離なフィルタの場合，可分離の実行時間が短い
  - フーリエ変換を用いる場合，速度はほぼ一定（定数時間）



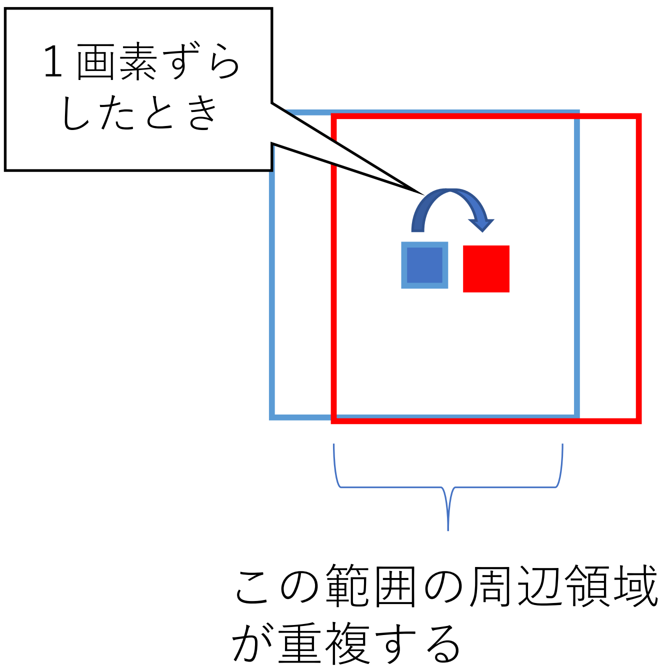
おまけ
効率的な計算方法３

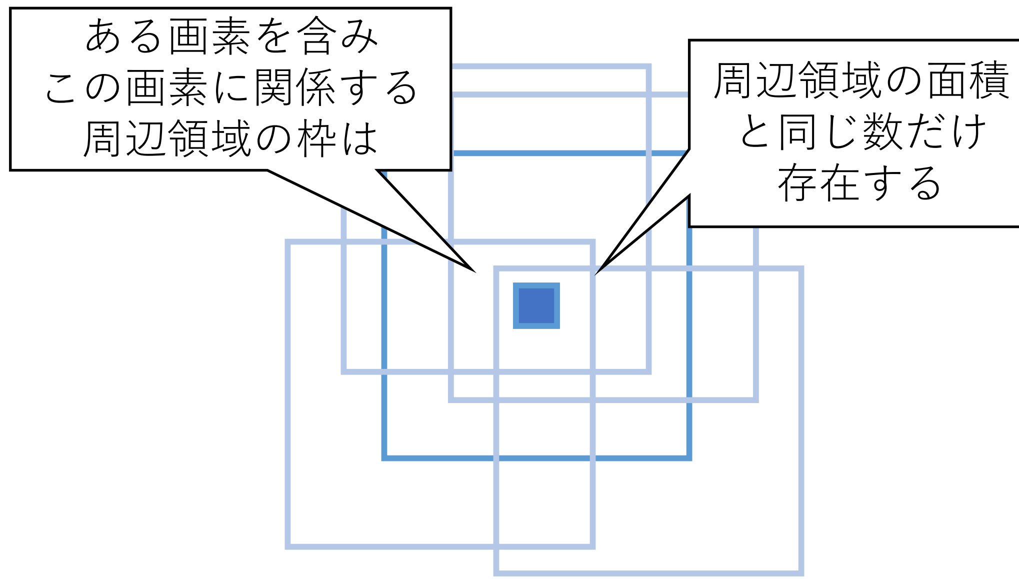
各画素で周辺画素値の合計値・平均値を求めるための
積分計算を用いた高速な計算方法

時間があるなら
高校生の知識を活用できるので
是非見ていただきたい

周辺画素を用いて計算する際どの程度の計算の無駄が生じるか

- １画素ずつずらしながら計算すると
  - その都度，周辺情報を取得，メモリアクセスに時間がかかる
  - 重複計算量は，膨大





平均値・合計値の計算であれば
この重複計算を省くテクが存在する

- そして，その方法は高校生で習った

定積分 の計算

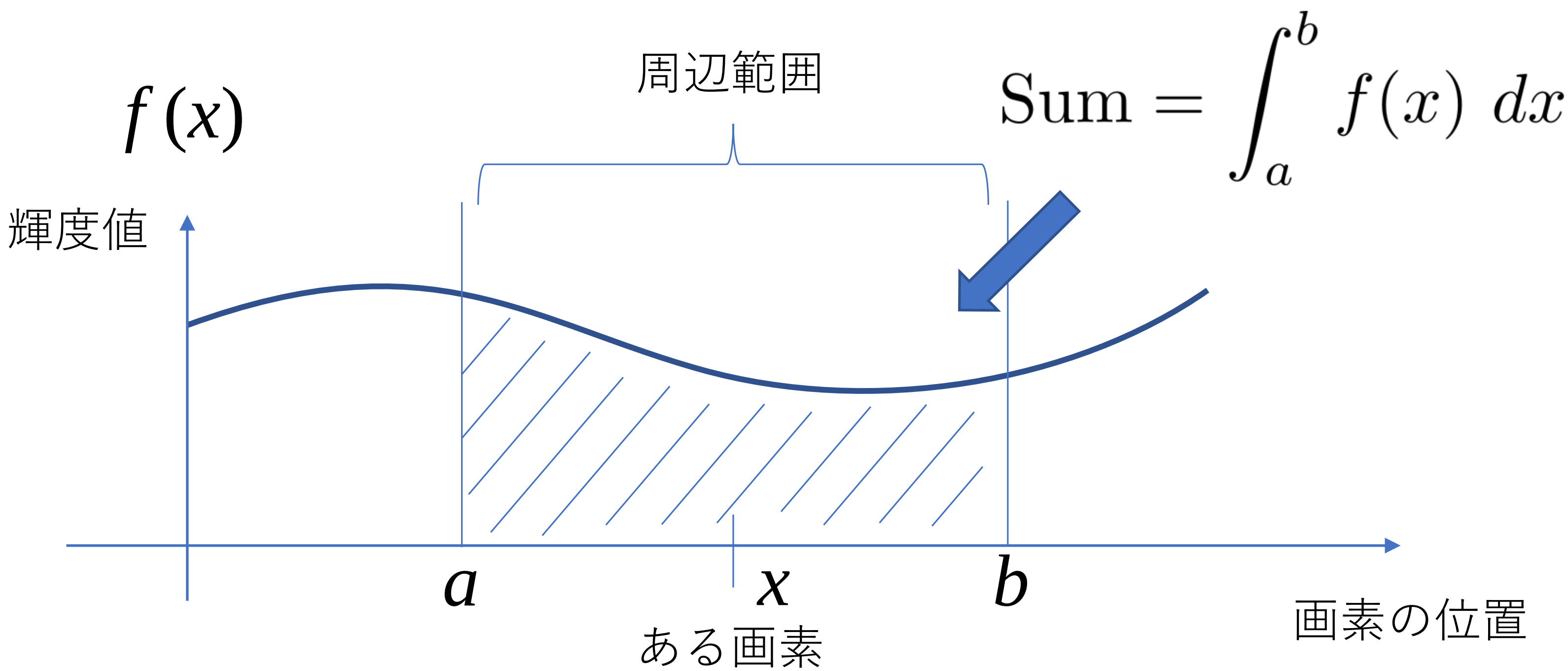
$$S = \int_a^b f(x)\ dx = F(b) - F(a)$$

ここで $$F(x) := \int f(x)\ dx$$

は積分後の関数
積分定数はあってもなくても構わない
$(F(b) + C) - (F(a) + C) = F(b) - F(a)$ と打ち消されるため

定積分はある区間の合計値を求める計算

- 画像の画素値が以下のように変化し，
- ある画素 x の周辺領域の範囲を a と b とすると
- この区間での定積分で，画素値の合計値が求まる



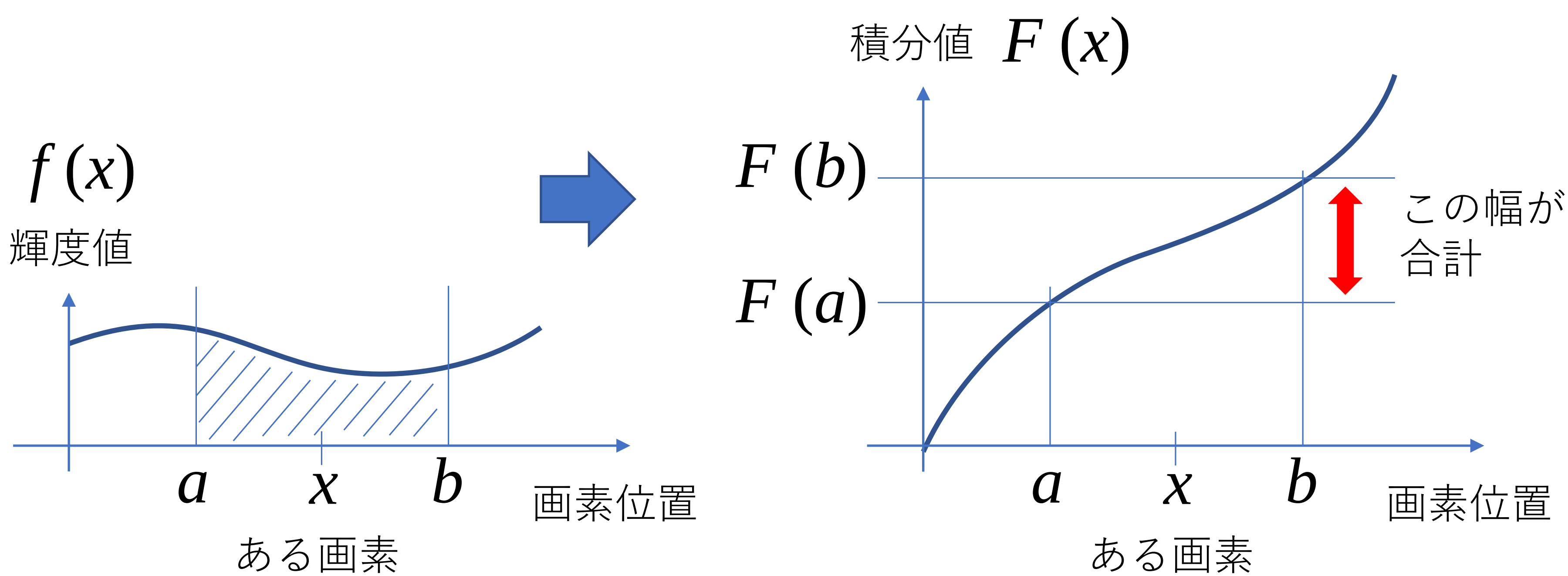
積分後の関数において考えてみると

- もし，積分後の関数や出力値を**予め**用意できるなら

$$F(x) := \int f(x)\, dx$$

引き算で済む

$$\mathrm{Sum} = F(b) - F(a)$$



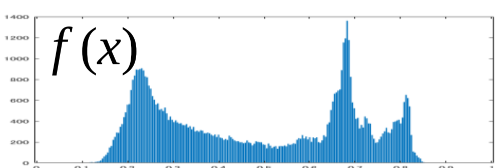
積分方法はすでに習っている

- ヒストグラム平坦化の際に行ったように

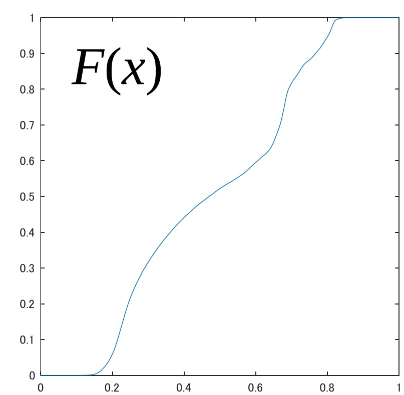
  ディジタル画像処理のような
  画素ごとの画素値，離散信号を扱う場合

積分は，累積和で計算できる

$$F(x) = \sum_{z=1}^{x} f(z)$$



頻度値の関数



トーンマップ関数

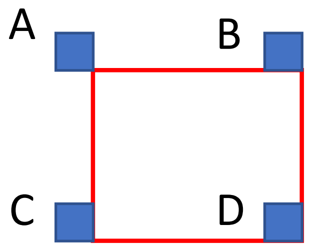
アルゴリズム　画像用（２次元版）

- 垂直方向へ，上から下へと，画素値の累積和を計算する．
- この計算結果を更に，
  水平方向へ，左から右へと，画素値の累積和を計算する．
  - この結果得られた画像は，
    積分画像 (Integral images) や
    範囲総和テーブルSummed Area Table (SAT) と呼ばれる

- 積分前　の画像での，合計値を計算したい範囲について，
  積分後　の画像での，範囲の４隅の画素値を得る．

  それぞれ，図のように A, B, C, D とする．

- 合計値は A + D – B – C で計算できる．
- 合計値を領域面積で割り，平均値を得る．



赤枠内の合計を
計算する場合

プログラム（１／２）

- ```
  % SAT を計算
  SAT = cumsum( Y, 1 );   % 垂直方向（１番目の次元方向）
  SAT = cumsum( SAT, 2 ); % 水平方向（２番目の次元方向）

  % 試しに，愚直な方法で
  iy = 100; ix = 100; r = 5;
  area = (2*r+1)^2; % 周辺領域の画素数

  Y_local = Y( iy-r:iy+r, ix-r:ix+r );
  mu_naive = sum( Y_local(:) ) / area;

  % 積分画像
  a = SAT( iy-r-1, ix-r-1 );   b = SAT( iy-r-1, ix+r );
  c = SAT( iy+r,   ix-r-1 );   d = SAT( iy+r,   ix+r );
  mu_SAT = (a + d - b - c) / area;

  % 比較
  mu_naive
  mu_SAT
  ```
  ```

プログラム（２／２）

```
% 全画素で計算
Y_out_SAT = zeros( size(Y) );

for ix = 1+r+1:sx-r % ix-r-1 の際に生じる ix = 0 を防止
  for iy = 1+r+1:sy-r

    % 積分画像
    a = SAT( iy-r-1, ix-r-1);
    b = SAT( iy-r-1, ix+r  );
    c = SAT( iy+r,   ix-r-1);
    d = SAT( iy+r,   ix+r  );

    % 合計計算
    Y_out_SAT( iy, ix ) = a + d - b - c;
  end
end

% 各画素値について，領域面積で割る
Y_out_SAT = Y_out_SAT / area;

% 結果の比較，左から naïve, SAT, 誤差
figure(1), imshow( [Y_out, Y_out_SAT] );
figure(2), meshz( Y_out-Y_out_SAT ); % 計算誤差を表示
```

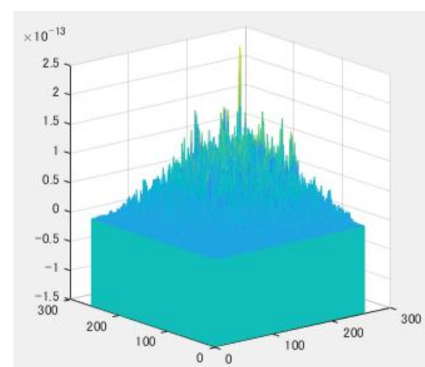
「Naïve な方法」とは
単純で直感的な方法

実験結果の比較

- 累積和を用いるため，計算誤差が蓄積されるが,
  見た目の変化は知覚できない程度.
- 計算量は，範囲総和テーブル(SAT)さえ作ってしまえば,
  窓サイズに依存せず，窓サイズに対して $O(1)$
- SAT の計算量は画素数に比例
- 画素周辺での合計値も $a + d - b - c$ のみであり，画素数に比例

Naïve な方法

SAT 使用



1e-13 のオーダーの誤差