

デジタルメディア処理2

担当: 井尻 敬

デジタルメディア処理 2、2017（前期）

4/13 デジタル画像とは : イントロダクション

4/20 フィルタ処理1 : 画素ごとの濃淡変換、線形フィルタ、非線形フィルタ

4/27 フィルタ処理2 : フーリエ変換、ローパスフィルタ、ハイパスフィルタ

5/11 画像の幾何変換 1 : アファイン変換

5/18 画像の幾何変換 2 : 画像の補間、イメージモザイク

5/25 画像領域分割 : 領域拡張法、動的輪郭モデル、グラフカット法、

6/01 前半のまとめ (約30分)と中間試験 (約70分)

6/08 特徴検出1 : テンプレートマッチング、コーナー・エッジ検出

6/15 特徴検出2 : DoG特徴量、SIFT特徴量、ハフ変換

6/22 画像認識1 : パターン認識概論、サポートベクタマシン

6/29 画像認識2 : ニューラルネットワーク、深層学習

7/06 画像符号化1 : 圧縮率、エントロピー、ランレングス符号化、MH符号化

7/13 画像符号化2 : DCT変換、ウェーブレット変換など

7/20 後半のまとめ (約30分)と期末試験 (約70分)

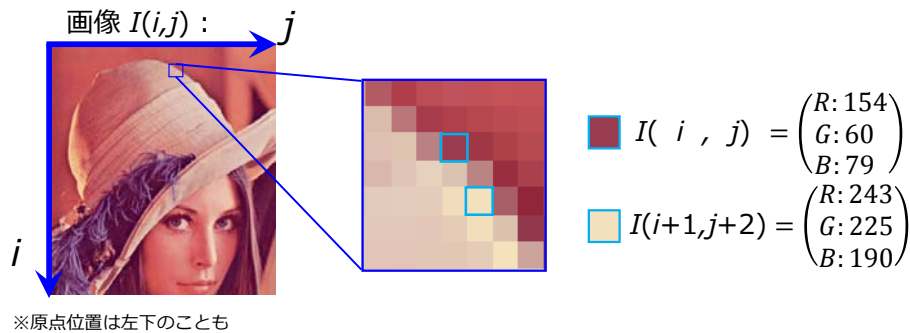
Contents : フィルタ処理

- デジタル画像とは (復習)
- トーンカーブ
 - 反転, 二値化, ポスタリゼーション, ソラリゼーション, ガンマ変換, カラー画像
- 空間フィルタ (線形)
 - 平滑化フィルタ, ソーベルフィルタ, ガウシアンフィルタ, ラプラシアンフィルタ
- 空間フィルタ (非線形)
 - メディアンフィルタ, バイラテラルフィルタ

デジタル画像のフィルタリング

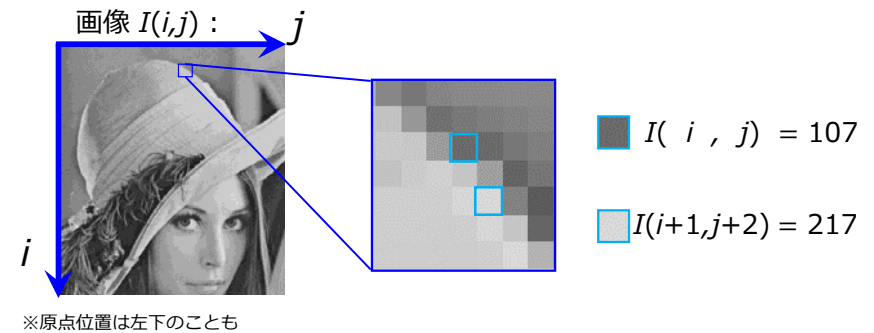
デジタル画像：カラー画像

- 離散値を持つ画素が格子状に並んだデータ
- 画素：pixel = picture + element
- 例 24bit bitmap：各pixelが(R,G,B)毎に整数値[0,255]を持つ



デジタル画像：グレースケール画像

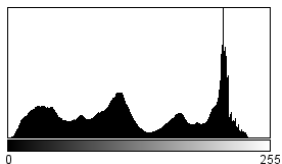
- 離散値を持つ画素が格子状に並んだデータ
- 画素：pixel = picture + element
- 例 8bit bitmap：各pixelが整数値[0,255]を持つ



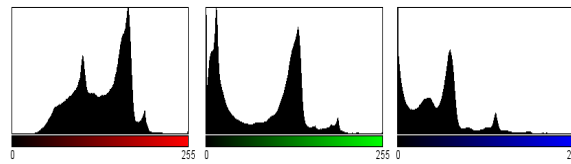
『頻度表（ヒストグラム）』とは

各階調の画素数を数えた表のこと
回転や平行移動に依存しない特徴量 → 画像処理に頻出

グレースケール画像

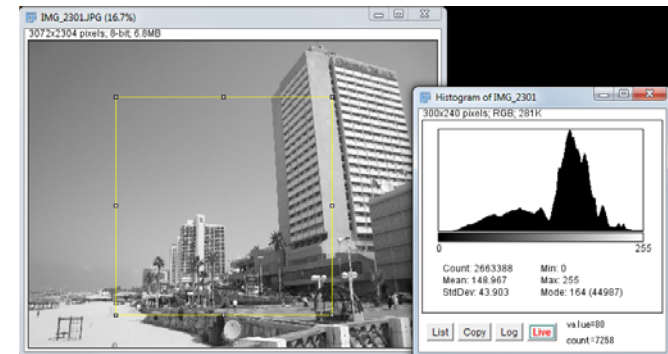


RGBカラー画像



ImageJでヒストグラムを確認してみる

1. ImageJ 起動
2. 画像読み込み
3. Menu > analyze > histogram
4. LiveをOnにすると矩形選択した領域のヒストグラムを確認可能



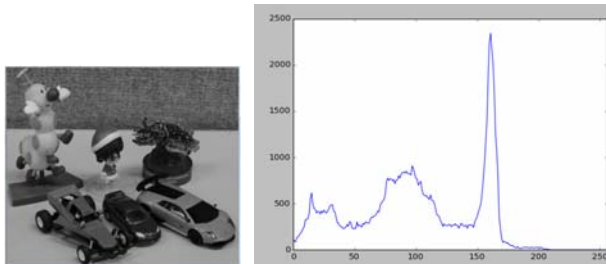
```
import numpy as np
import pylab as plt
import cv2
import itertools
#画像読み込み & グレースケール化
img = cv2.imread("imgs/sample.png")
img_gry = cv2.cvtColor( img,
cv2.COLOR_BGR2GRAY )

#histogram生成
hist = np.zeros(256)
for y in range(img_gry.shape[0]):
    for x in range(img_gry.shape[1]):
        hist[ img_gry[y, x] ] += 1

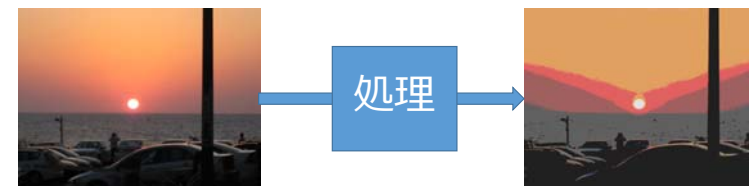
#windowを生成して画像を表示
cv2.imshow("Image", img_gry )

#histをmatplotlibで表示
plt.plot(hist)
plt.xlim([0, 256])
plt.show()
```

ヒストグラムの計算： histogram.py



デジタル画像のフィルタリング



入力画像に対し何らかの計算処理を施し…

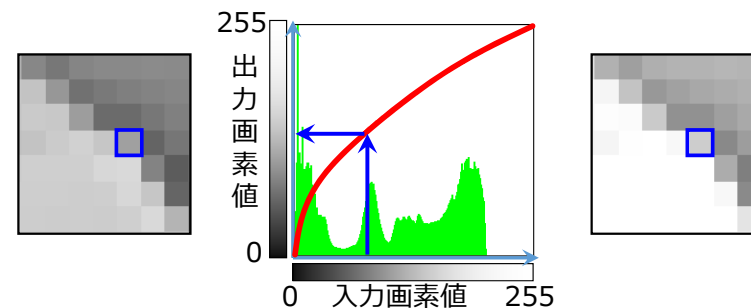
- 特定の周波数を持つ信号を強調する・捨てる（ノイズ除去）
- アーティスティックな効果を得る
- 画像処理（ステレオ視・領域分割・識別器）に必要な特徴ベクトルを得る

トーンカーブ

トーンカーブ

CToneCurve.exe (C++)
Image>Adjust>Window/Level (ImageJ)

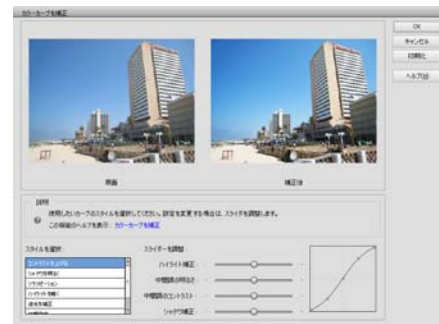
- 入力画像は8bit グレースケールとする
- 各画素の値を異なる値に変換する**階調変換関数**を考える
- 階調変換関数をグラフで表現したものを**トーンカーブ**と呼ぶ



トーンカーブは写真編集の基本ツール



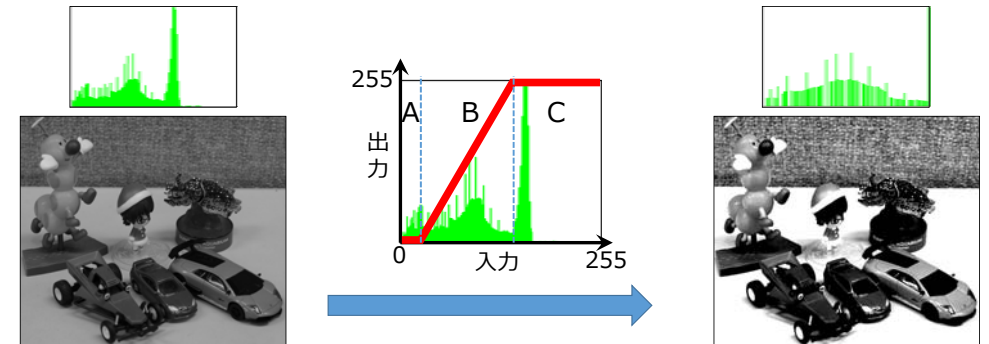
GIMP



PhotoShop Elements カラーカーブ
使いやすいように自由度の限定されたトーンカーブのようなもの
Photoshop CSにはトーンカーブがある (あった)

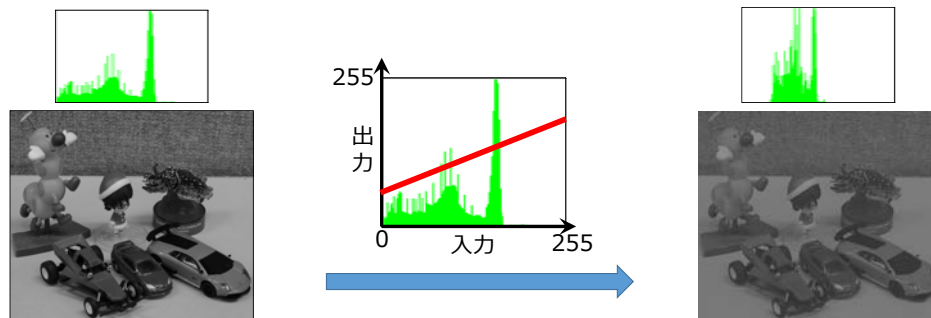
ImageJ: 自由編集でないのちょっと違うけど

トーンカーブ: コントラストを上げる



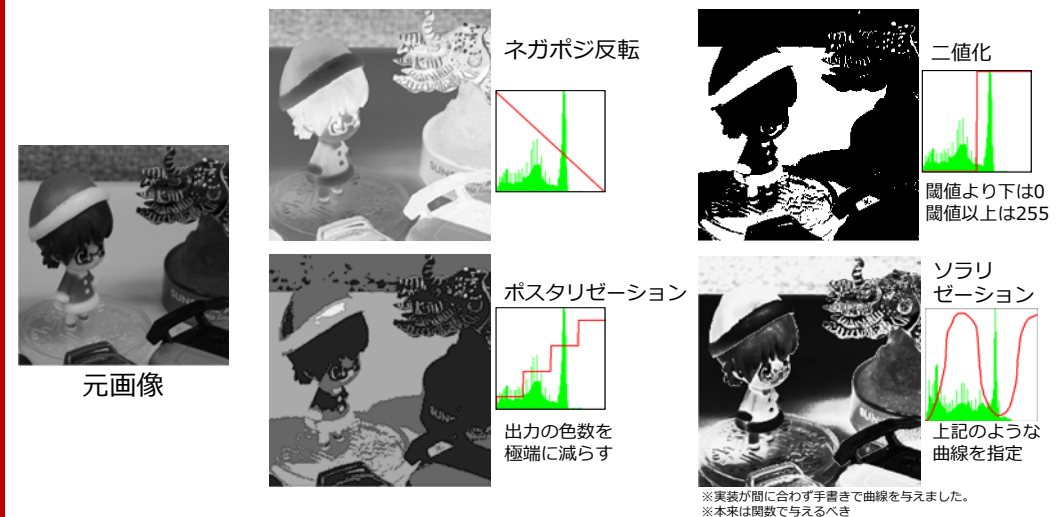
- 領域A: 出力画素値0となり黒つぶれ
- 領域C: 出力画素値255となり白飛び
- 領域B: 傾きが1より大きいため、画素値の取り得る範囲が広がりコントラストが上がる
画素値は離散値であるため出力ヒストグラムは飛び飛びに

トーンカーブ: コントラストをさげる



- 傾きが1より小さいため、出力画素値の取り得る範囲が縮まり、コントラストが下がる

トーンカーブ: 特殊効果

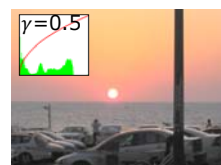
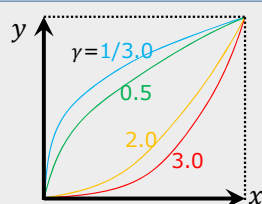


トーンカーブ：ガンマ補正

次のトーンカーブを利用した濃淡変換をガンマ変換と呼ぶ

$$y = 255 \left(\frac{x}{255} \right)^\gamma$$

x : 入力値 [0,255]
 y : 出力値 [0,255]
 γ : パラメータ (>0)



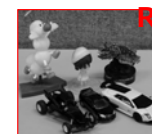
※ RGB各チャンネルに
ガンマ補正を適用

※ 画像出力デバイスには『出力値 = (入力値) $^\gamma$ 』という関係があり、この特性を補正する目的で上記の関数が用いられていた。これを画像の補正に利用したのがガンマ変換

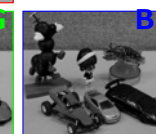
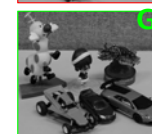
トーンカーブ：カラー画像への適用

カラー画像をトーンカーブで編集するとき …

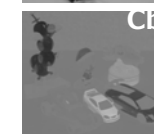
- RGBの各チャンネルにトーンカーブの画素値変換を適用
- YCbCr Colorに変換し輝度値成分 (Y) のみに変換を適用
- その他



RGB color



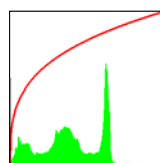
YCbCr color
輝度/青み/赤み



トーンカーブ：カラー画像への適用



入力画像



$\gamma=0.3$
のガンマ変換



RGB各チャンネル

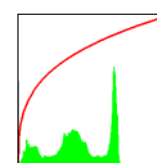


YCbCrの輝度Yのみ

トーンカーブ：カラー画像への適用



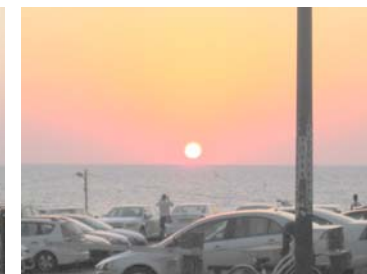
入力画像



$\gamma=0.3$
のガンマ変換



RGB各チャンネル

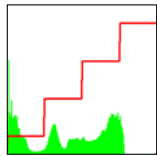


YCbCrの輝度Yのみ

トーンカーブ：カラー画像への適用



入力画像



ポスタリゼーション



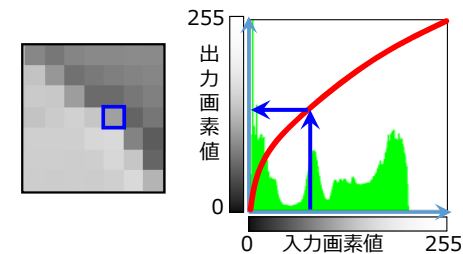
RGB各チャンネル



YCbCrの輝度Yのみ
(Cb・Crの階調数は減らない)

トーンカーブ：まとめ

- トーンカーブ：各画素の輝度値・色を変換する**階調変換関数**
- 画像の見栄えの編集に利用される
- キーワード: コントラスト変換・ネガポジ反転・ポスタリゼーション・ソラリゼーション・2値化・ガンマ補正



空間フィルタ(線形)

```
import numpy as np
import cv2
import itertools

def myConvolve(srcImg, filter):
    H = srcImg.shape[0]
    W = srcImg.shape[1]
    R = int(filter.shape[0] / 2)
    trgImg = np.zeros(srcImg.shape)

    for v, u in itertools.product(range(1, H-1), range(1, W-1)):
        pix = 0
        for vv, uu in itertools.product(range(-R, R+1), range(-R, R+1)):
            pix += filter[R + vv][R + uu] * srcImg[v+vv][u+uu]
        trgImg[v][u] = min(255, max(0, abs(pix)))

    return np.uint8(trgImg)

img = cv2.imread("imgs/lenaColor.png")
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

filter_smooth = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]]) / 9.0
filter_sobelV = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
filter_sobelH = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

img_smooth = myConvolve(img, filter_smooth)
img_sobelV = myConvolve(img, filter_sobelV)
img_sobelH = myConvolve(img, filter_sobelH)

cv2.imshow("original", img)
cv2.imshow("img_smooth", img_smooth)
cv2.imshow("img_sobelV", img_sobelV)
cv2.imshow("img_sobelH", img_sobelH)
cv2.waitKey(0)
```

Convolution1.py 線形フィルタの計算



線形フィルタの例



ぼかす

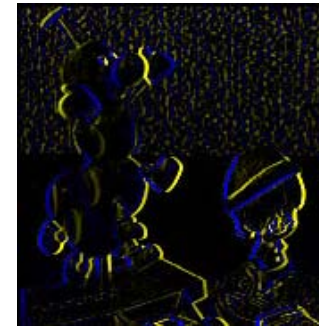


先鋭化

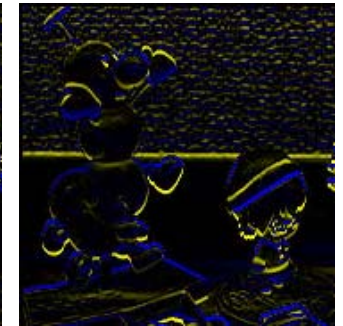
線形フィルタの例



エッジ抽出



横方向



縦方向

空間フィルタとは

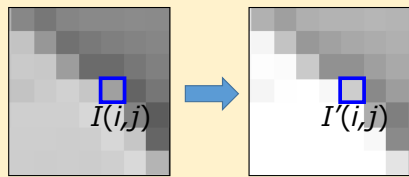
- 空間フィルタとは周囲の情報を利用して画素値を決めるフィルタ
- 空間フィルタは、線形フィルタと非線形フィルタに分けられる

トーンカーブ：

出力画素 $I'(i,j)$ を求めるのに
入力画素 $I(i,j)$ のみを利用

入力画像： $I(i,j)$

出力画像： $I'(i,j)$

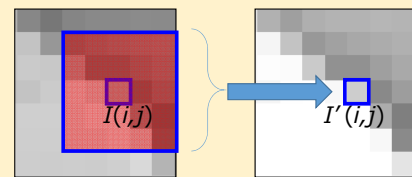


空間フィルタ：

出力画素 $I'(i,j)$ を求めるのに
入力画素 $I(i,j)$ の周囲画素も利用

入力画像： $I(i,j)$

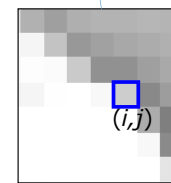
出力画像： $I'(i,j)$



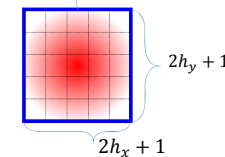
線形フィルタとは

出力画素値を周囲画素の重み付和で計算するフィルタ

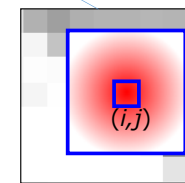
$$I'(i,j) = \sum_{m=-h_y}^{h_y} \sum_{n=-h_x}^{h_x} h(m,n) I(i+m, j+n)$$



$I'(i,j)$
出力画像

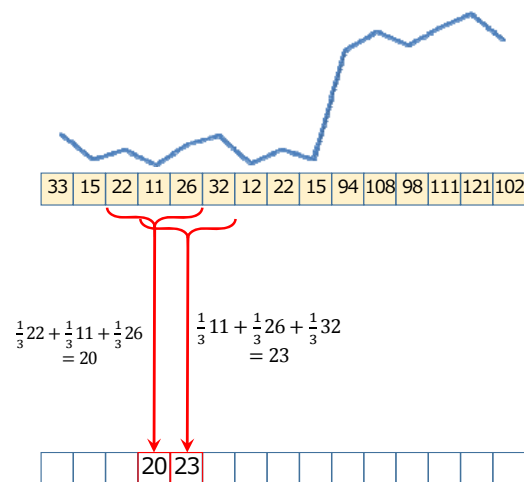


$h(i,j)$
フィルタ



$I(i,j)$
入力画像

線形フィルタの例 1D

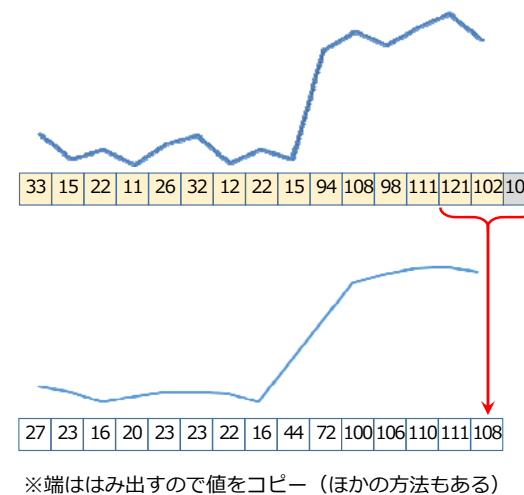


平滑化したい!

$\frac{1}{3} \quad \frac{1}{3} \quad \frac{1}{3}$

周囲3ピクセル
の平均を取る

線形フィルタの例 1D

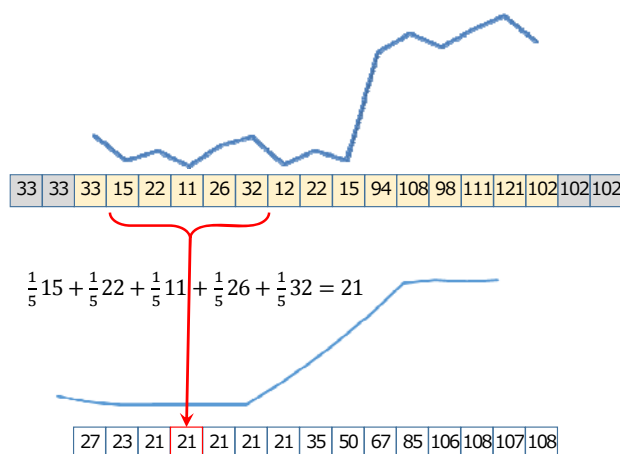


平滑化したい!

$\frac{1}{3} \quad \frac{1}{3} \quad \frac{1}{3}$

周囲3ピクセル
の平均を取る

線形フィルタの例 1D

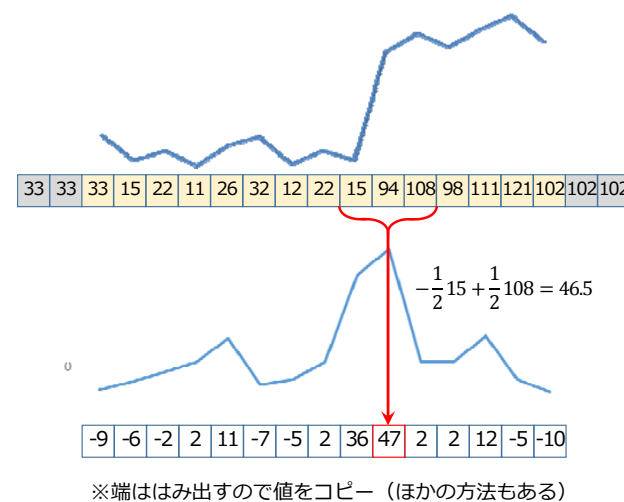


もっと
平滑化したい!

$\frac{1}{5} \quad \frac{1}{5} \quad \frac{1}{5} \quad \frac{1}{5} \quad \frac{1}{5}$

周囲5ピクセル
の平均を取る

線形フィルタの例 1D



エッジ
(変化の大きい部分)
を検出したい

$-0.5 \quad 0 \quad 0.5$

右と左のピクセルの
差をとる

線形フィルタ：平滑化

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$



LinaerFilter.exe (C++)
convolution1.py (python)
Process>Filters>Convolve (ImageJ)

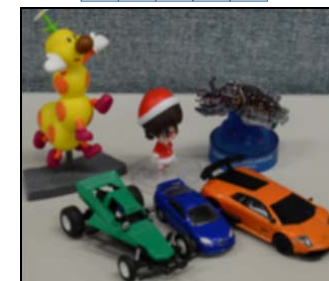
線形フィルタ：特定方向の平滑化

線形フィルタ：ガウシアンフィルタ

係数をガウス分布に近づけ
中央ほど強い重みに

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

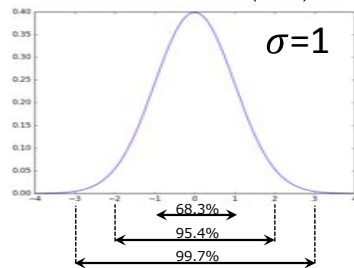
$$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$



線形フィルタ：ガウシアンフィルタ

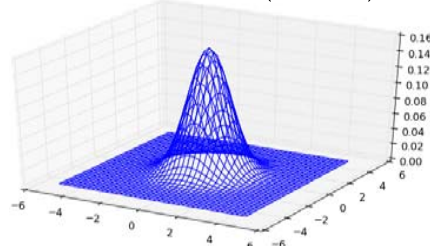
ガウス関数1D

$$g_{\sigma}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{\sigma^2}\right)$$

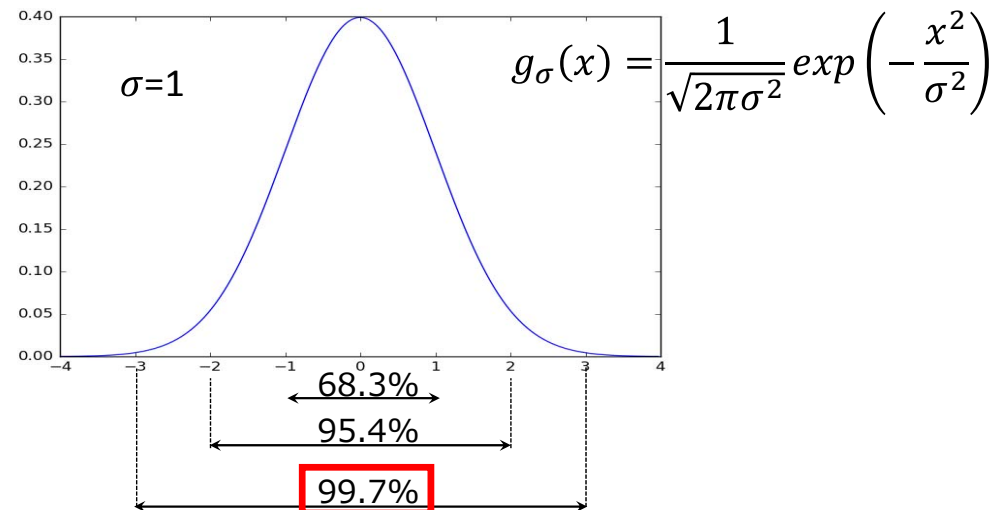


ガウス関数2D

$$g_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{\sigma^2}\right)$$



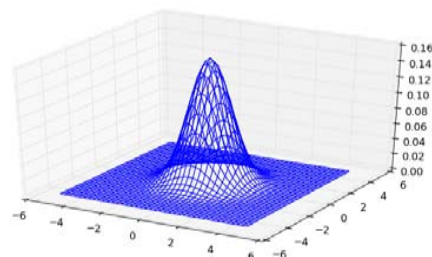
これを重みにして線形フィルタをしたい
さすがに3x3は精度が悪くない??



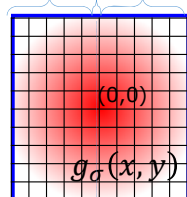
線形フィルタ：ガウシアンフィルタ

標準偏差 σ の大きなガウス関数の畳み込みを計算するとき『3x3』や『5x5』の窓では精度が悪い

→精度を出すには窓の半径を 3σ 程度にすべき
(計算時間はかかる)



(3 σ)1 (3 σ)



例) $\sigma = 5$ pixelの
ガウシアンフィルタ
↓
Window size は
31x31が適当

線形フィルタ：微分

関数 $f(x, y)$ の x 軸, y 軸方向の偏微分は以下の通り定義され、

$$\frac{\partial f(x, y)}{\partial x} = \lim_{h \rightarrow 0} \frac{f(x + h, y) - f(x - h, y)}{2h}$$

$$\frac{\partial f(x, y)}{\partial y} = \lim_{h \rightarrow 0} \frac{f(x, y + h) - f(x, y - h)}{2h}$$

点 (x, y) における x 軸, y 軸方向の関数 $f(x, y)$ の傾きを与える。

また, $f(x, y)$ の勾配 $\nabla f(x, y)$ は2次元ベクトルであり、

$$\nabla f(x, y) = \begin{pmatrix} \frac{\partial f(x, y)}{\partial x} \\ \frac{\partial f(x, y)}{\partial y} \end{pmatrix}$$

点 (x, y) において $f(x, y)$ の増加が一番大きくなる方向を示す

※微分の復習。大丈夫ですね?

練習。

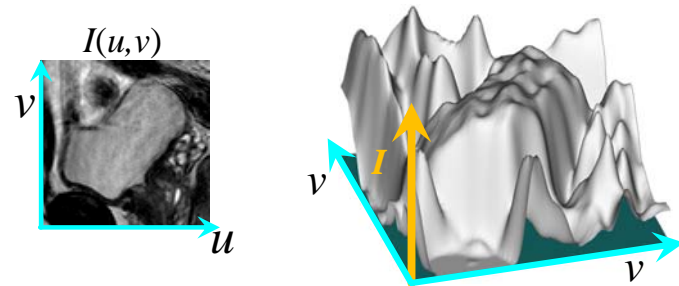
$$f(x, y) = x^2 + y^2$$

上記の関数の (1, 1), (2, 3)

における勾配を計算し、

さらに図示せよ

線形フィルタ：微分



グレースケール画像 $I(u,v)$ は、高さ関数 $z = I(u,v)$ と見なせる
 なので関数 $I(u,v)$ の勾配（微分）は計算できそう
 $I(u,v)$ の勾配は、画像の変化の大きい方向を表す

画像の出典 [Ijiri et al 2013, Eurographics]

線形フィルタ：微分

2次元関数 $z = f(x,y)$ のx方向偏微分

$$f_x = \frac{\partial f(x,y)}{\partial x} = \lim_{h \rightarrow 0} \frac{f(x+h,y) - f(x,y)}{h}$$

画像 $z = I(i,j)$ の横方向偏微分 (近似)

$$I_j(i,j) \approx f(i,j+1) - f(i,j) \quad \dots(a)$$

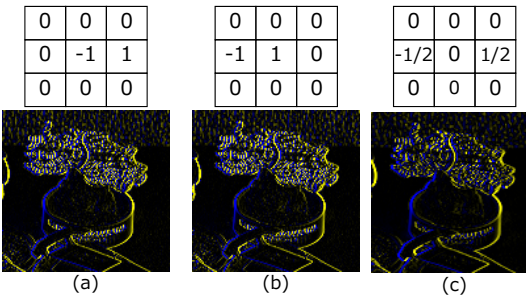
$$\approx f(i,j) - f(i,j-1) \quad \dots(b)$$

$$\approx \frac{f(i,j+1) - f(i,j-1)}{2} \quad \dots(c)$$

※ $h = \text{pitch}$ (画素サイズ) = 1 と近似



※ 正值:黄色,
負値:青 で可視化



線形フィルタ：微分



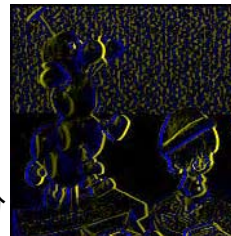
$I(i,j)$ 入力画像

教科書 図5.26

微分フィルタには画像のエッジ
で強く応答する

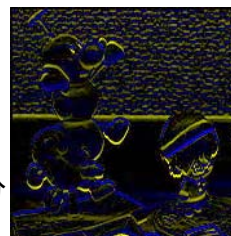
0	0	0
-1/2	0	1/2
0	0	0

$I_j(i,j)$
横方向微分



0	-1/2	0
0	0	0
0	1/2	0

$I_i(i,j)$
縦方向微分



線形フィルタ：微分

- 前述の単純なフィルタはノイズにも鋭敏に反応する
- ノイズを押さえつつエッジを検出するフィルタが必要

横方向微分 : 横方向微分 し 縦方向平滑化 する

縦方向微分 : 縦方向微分 し 横方向平滑化 する

Prewitt filter

-1	0	1
-1	0	1
-1	0	1

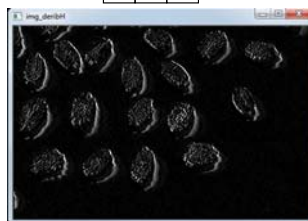
Sobel filter

-1	0	1
-2	0	2
-1	0	1

元画像

1	0	1
-2	0	2
-1	0	1

0	0	0
-4	0	4
0	0	0



微分フィルタの正値を可視化
Sobelフィルタではノイズが削減されているのが分かる

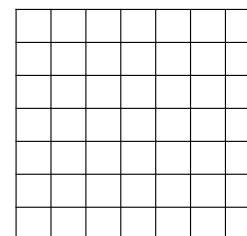
フィルタ処理

• 右の7x7 画像に対して…

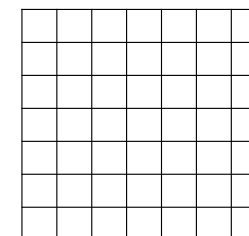
1. 横方向Sobelフィルタを適用せよ
2. 縦方向Sobelフィルタを適用せよ
3. ガウシアンフィルタを適用せよ

4	4	4	1	2	3	3
4	4	4	1	2	3	3
4	4	4	1	2	3	3
4	4	4	1	2	3	3
4	4	4	1	2	3	3
4	4	4	1	2	3	3
4	4	4	1	2	3	3

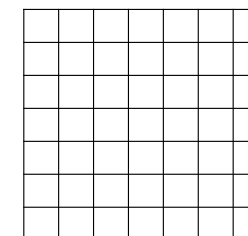
入力画像



横Sobel



縦Sobel



ガウシアン

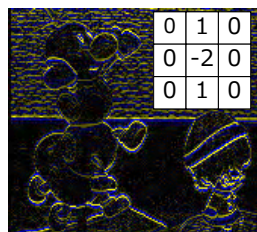
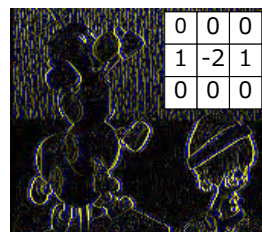
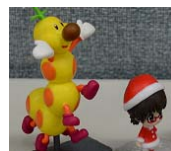
線形フィルタ：2階微分フィルタ

関数 $f(x, y)$ の2階偏微分は、以下の通り定義される

$$f_{xx} = \frac{\partial^2 f(x, y)}{\partial x^2} = \lim_{h \rightarrow 0} \frac{f(x + h, y) - 2f(x, y) + f(x - h, y)}{h^2}$$

画像 $I(i, j)$ の2階偏微分の近似は…

$$I_{ij} = f(i, j + 1) - 2f(i, j) + f(i, j - 1)$$



出典[CGArts協会, デジタル画像処理 図5.26]

線形フィルタ：ラプラシアンフィルタ

関数 $f(x, y)$ のラプラシアン

$$\Delta f(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}$$

画像 $I(u, v)$ のラプラシアン

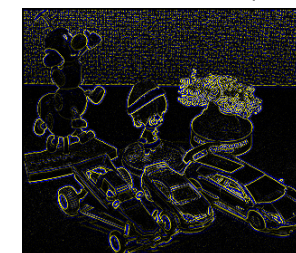
$$I(u, v) = I_{uu} + I_{vv}$$

$$\Delta I(u, v) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \text{image} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} * \text{image}$$

$$= \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * \text{image}$$

↑
ラプラシアンフィルタ

『*』は
convolution



方向に依存しないエッジが一度で得られる
エッジをまたぎ正負の対が現れる
白→黒 なら [0-+0]が現れる

線形フィルタ：先鋭化フィルタ

2回微分に関するラプラシアンフィルタを改良すると
画像のエッジを強調する先鋭化フィルタが設計できる

	?	



[CGArts協会, デジタル画像処理]
図 5.26, 5.30

まとめ：空間フィルタ（線形）

出力画素値を周囲画素の重み付和で計算するフィルタ

$$I'(i,j) = \sum_{m=-h_y}^{h_y} \sum_{n=-h_x}^{h_x} h(m,n) I(i+m,j+n)$$

平滑化フィルタ

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

ガウシアンフィルタ

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

先鋭化フィルタ

	?	

Sobelフィルタ(横)

-1	0	1
-2	0	2
-1	0	1

Sobelフィルタ(縦)

-1	-2	-1
0	0	0
1	2	1

ラプラシアンフィルタ

0	1	0
1	-4	1
0	1	0