

# パターン認識と学習 テンプレートマッチング

管理工学科

篠沢佳久

# 資料の内容

- パターン認識の基礎
  - テンプレートマッチング
- テンプレートマッチングの例題

# パターン認識の基礎

テンプレートマッチング

# テンプレートマッチング

- 電子通信情報学会

- パターン認識・メディア理解研究会

- アルゴリズムコンテスト

<http://www.ieice.org/~prmu/jpn/>

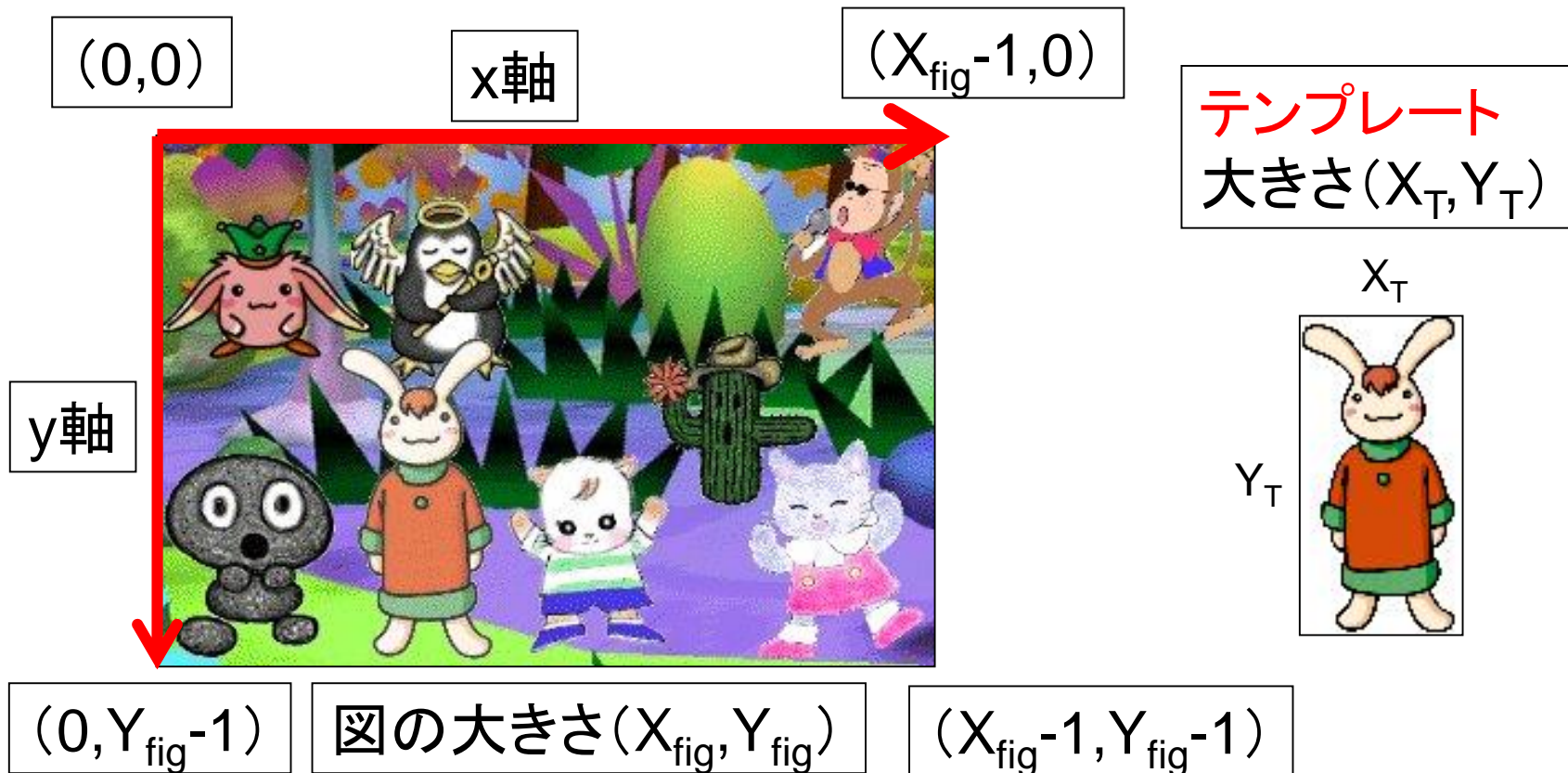


「うさぎ」は右図中のどこにいますか

# 人の場合, どのようにするでしょうか？

- 「うさぎ」(テンプレート)を記憶する
- 目で追って「うさぎ」を「探す」
- 「探す」とは
  - 記憶したものと一致しているかを判断する(マッチング)

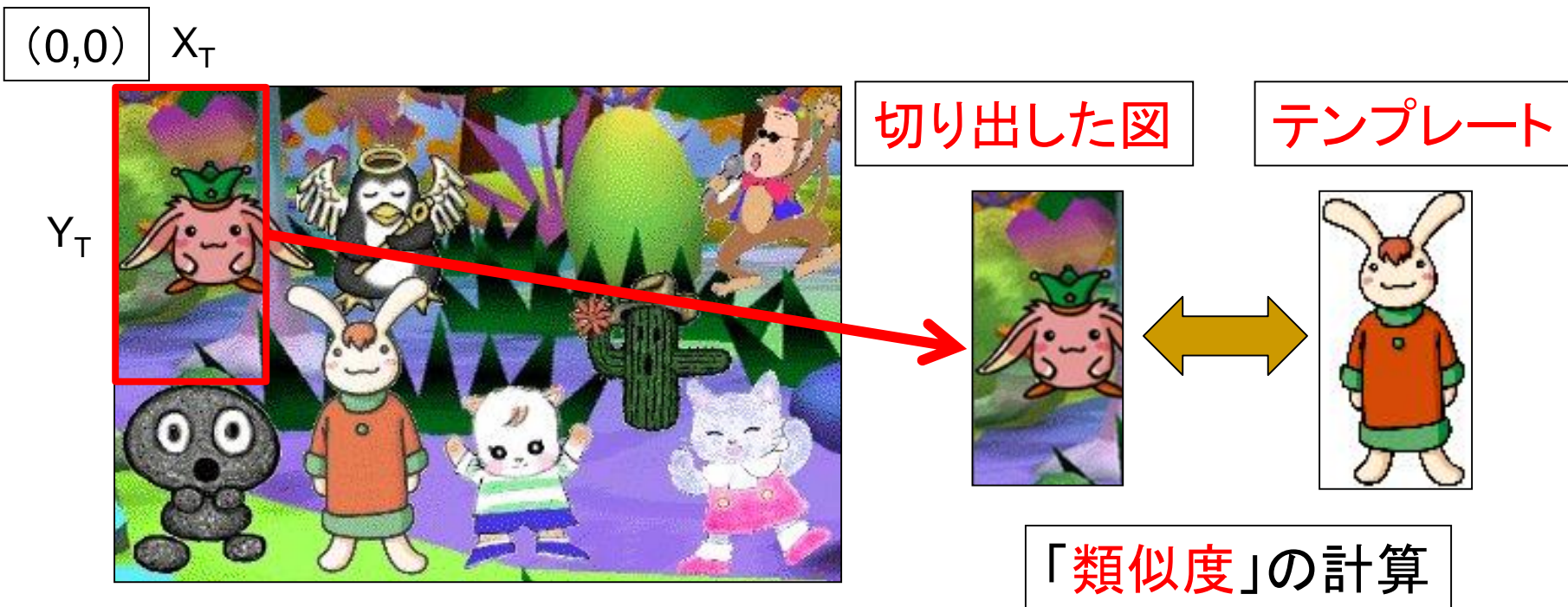
# テンプレートマッチングの流れ①



図中の座標(0,0)から大きさ( $X_T, Y_T$ )の図を切り出し、テンプレートとの「類似度」を計算→最も「類似度」の高い図を「うさぎ」と判断する

# テンプレートマッチングの流れ②

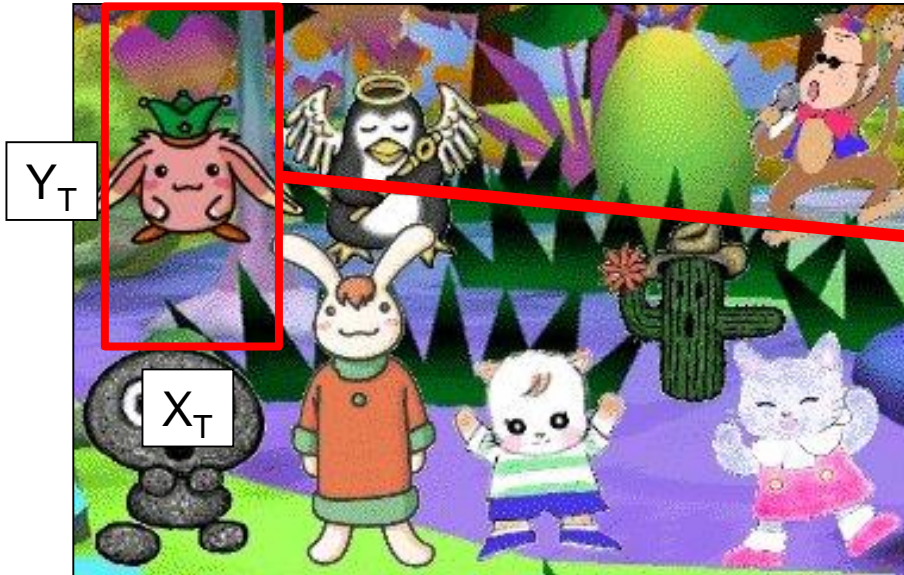
- 図中の座標  $(0,0)$  から大きさ  $(X_T, Y_T)$  の図を切り出す
- テンプレートとの「類似度」を計算





# テンプレートマッチングの流れ③

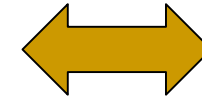
(0,1)



切り出した図



テンプレート

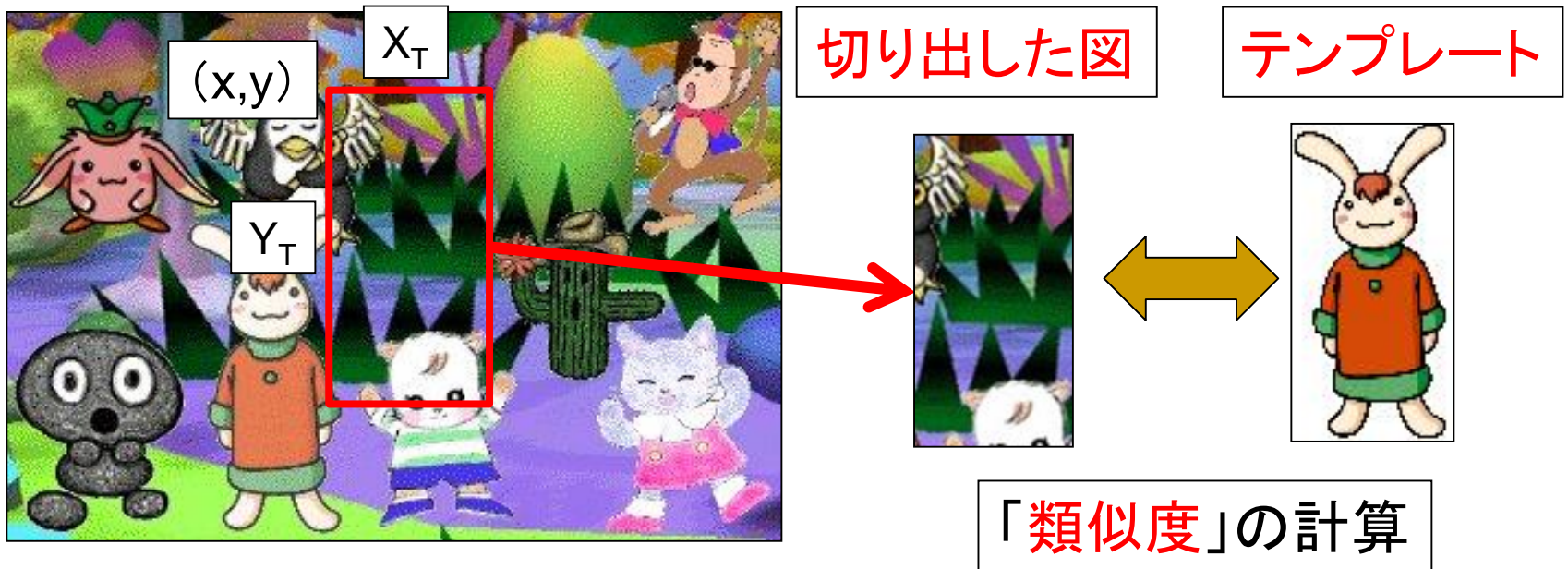


「類似度」の計算

X軸, Y軸の切り出し位置をずらして, 図を切り出し,  
「類似度」を計算

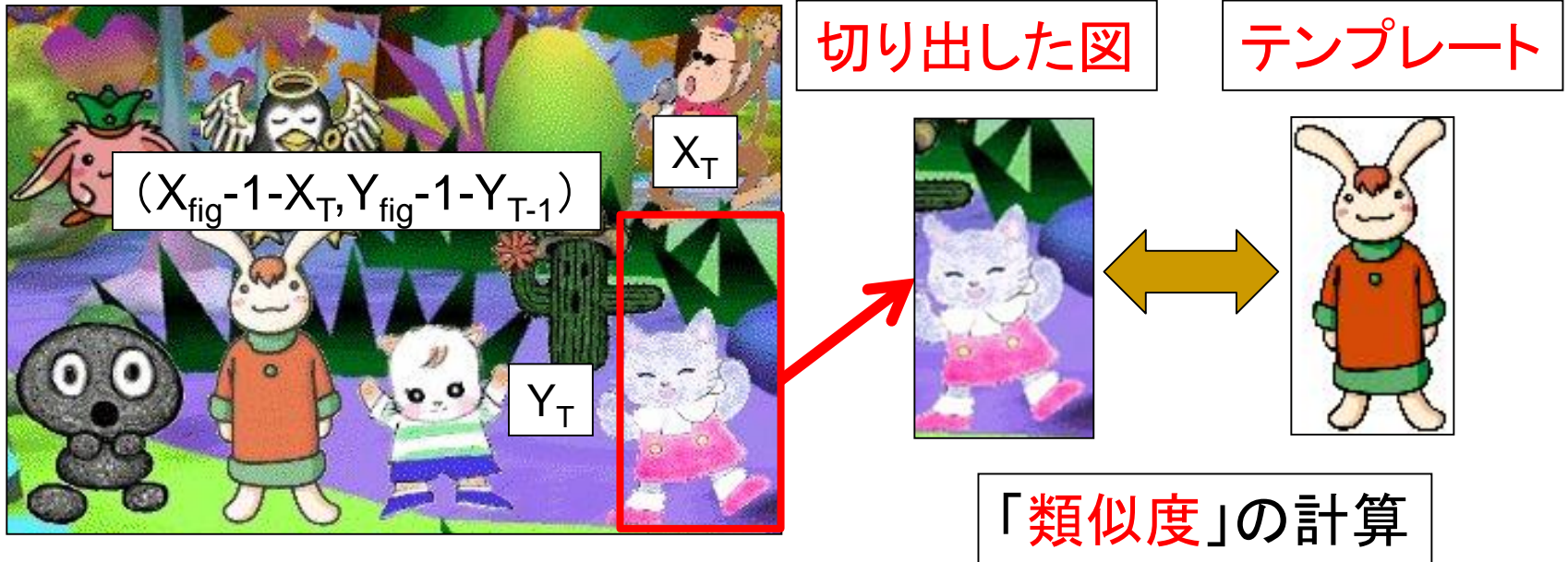


# テンプレートマッチングの流れ④



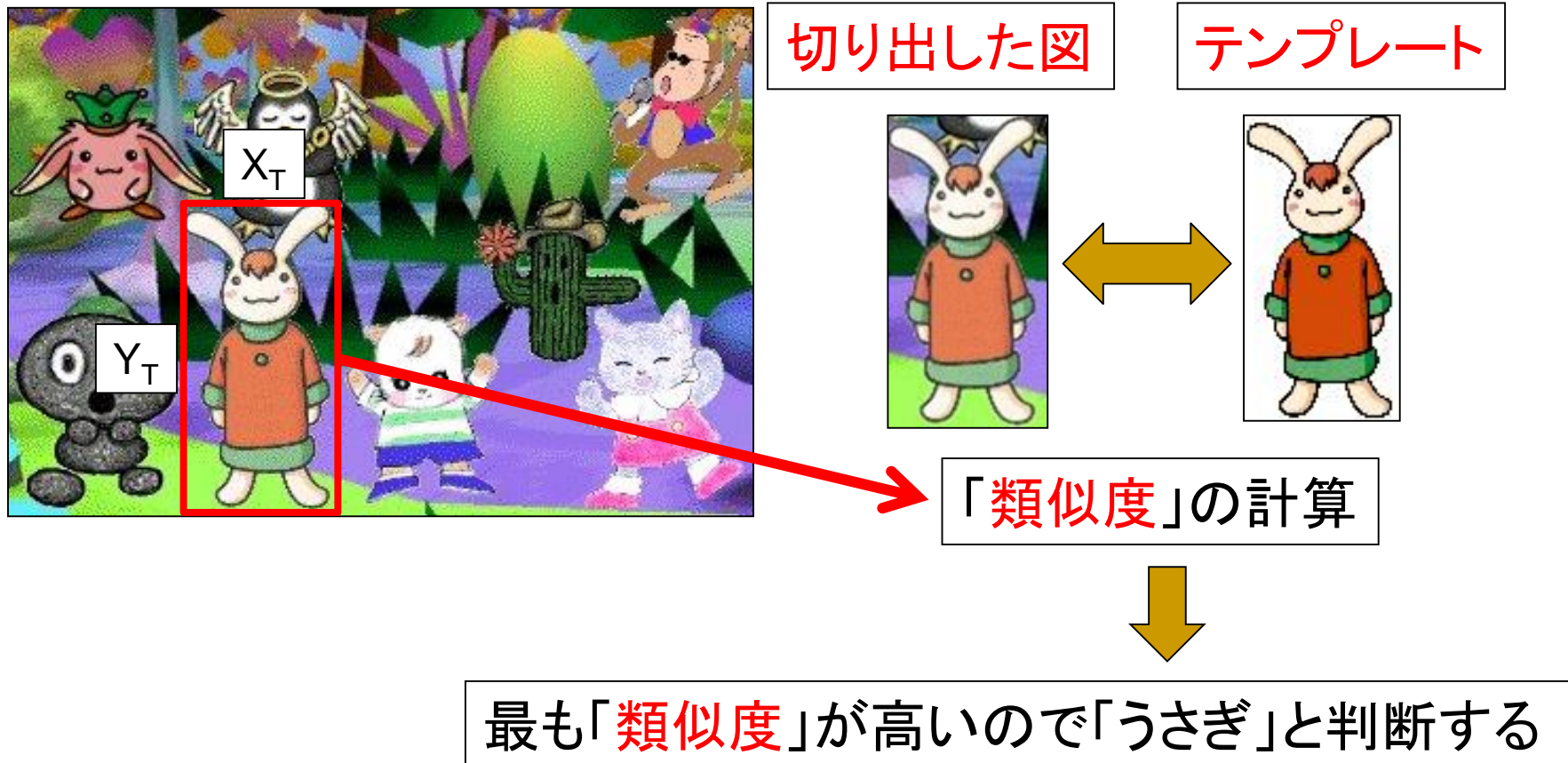
全ての座標上で図を切り出し、「類似度」を計算

# テンプレートマッチングの流れ⑤



全ての座標上で図を切り出し、「類似度」を計算

# テンプレートマッチングの流れ⑥



# テンプレートマッチングのアルゴリズム

```
max =  $-\infty$ 
```

```
for( y = 0 ; y <  $Y_{\text{fig}} - Y_T$  ; y++ ) {
```

```
    for( x = 0 ; x <  $X_{\text{fig}} - X_T$  ; x++ ) {
```

```
        座標(x,y)から大きさ( $X_T, Y_T$ )の図を切り出す
```

```
        similarity = 類似度(切り出した図, テンプレート)
```

```
        if( similarity > max ) {
```

```
            max = similarity
```

```
             $X_{\text{max}} = x$ 
```

```
             $Y_{\text{max}} = y$ 
```

```
        }
```

```
    }
```

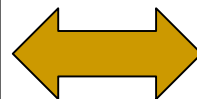
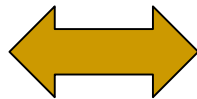
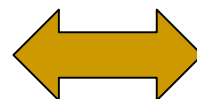
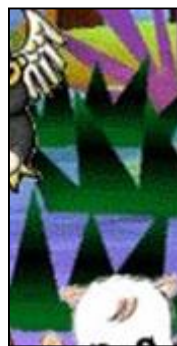
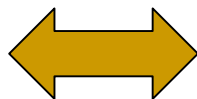
```
}
```

座標( $X_{\text{max}}, Y_{\text{max}}$ )から, 大きさ( $X_T, Y_T$ )の図を一致した図とみなす

# 類似度の求め方

## ■ 類似度の求め方

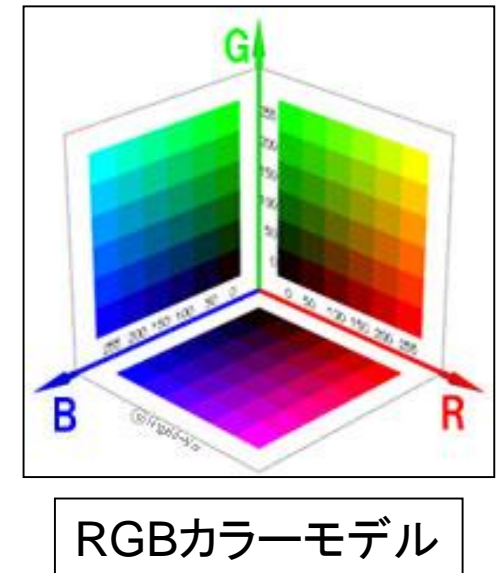
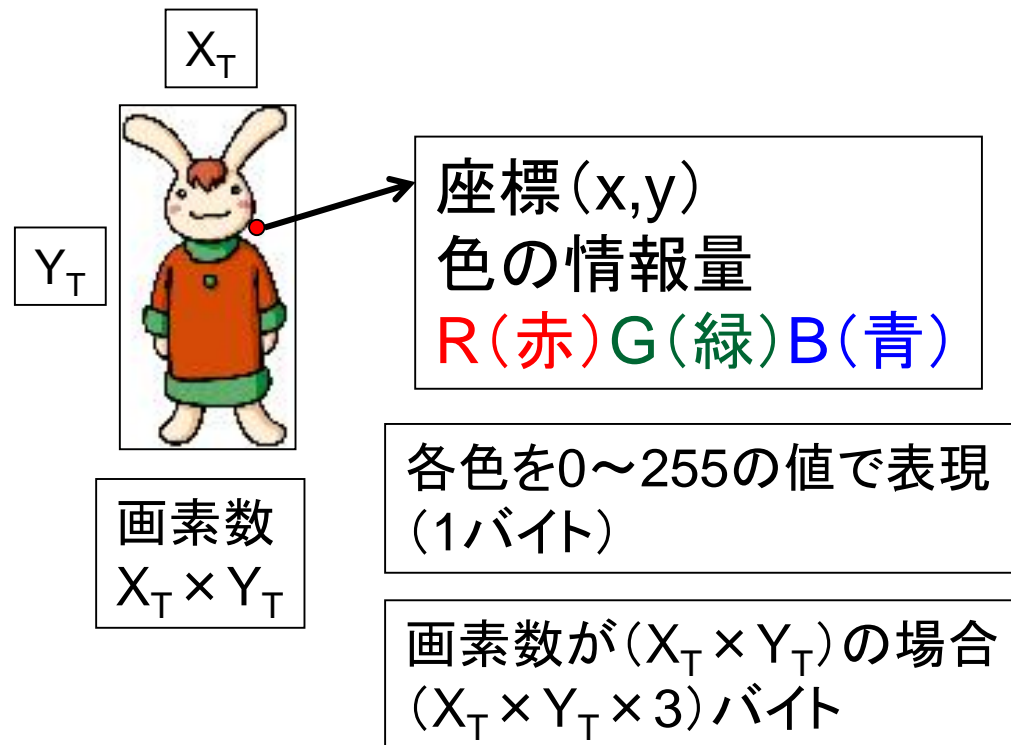
- 二つの画像間で**どういった値**を比較するのか？
- **類似度**をどのように定義するのか？





# カラー画像(デジタル画像)①

- ビットマップ形式, ラスターイメージ
- 画素(ピクセル, メッシュ)単位に情報量を持つ





# カラー画像(デジタル画像)②

- 画像は $(X_T \times Y_T)$ の行列で表現できる

$$X = \begin{pmatrix} x_{11} & x_{21} & \cdots & x_{X_T 1} \\ x_{12} & x_{22} & & x_{X_T 2} \\ \vdots & & \ddots & \vdots \\ x_{1Y_T} & x_{2Y_T} & \cdots & x_{X_T Y_T} \end{pmatrix}$$

$x_{ij} : (r_{ij}, g_{ij}, b_{ij})$   
 $r_{ij}$ : 赤,  $g_{ij}$ : 緑,  $b_{ij}$ : 青  
それぞれ0~255の値

- 一般的にはベクトルにて表現
  - これを特徴量もしくは特徴ベクトルと呼ぶ

$$\mathbf{x}^t = (x_1, x_2, \cdots, x_{X_T Y_T})$$

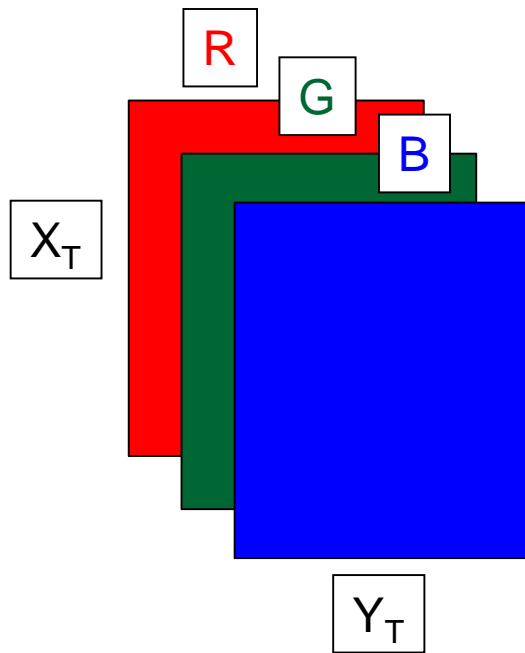
$$x_i : (r_i, g_i, b_i)$$



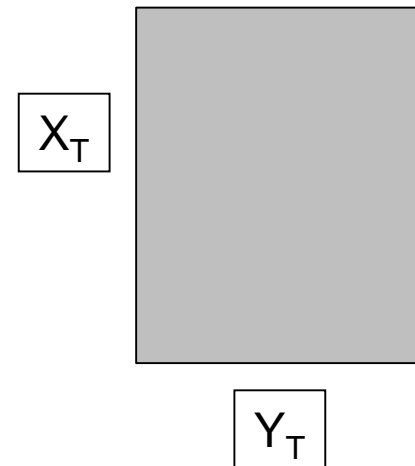
$$\mathbf{x}^t = (x_1, x_2, \cdots, x_{X_T \times Y_T \times 3})$$

# カラー画像(デジタル画像)③

- 最近(?), 画像は( $X_T \times Y_T \times \text{チャンネル数}$ )の配列で処理



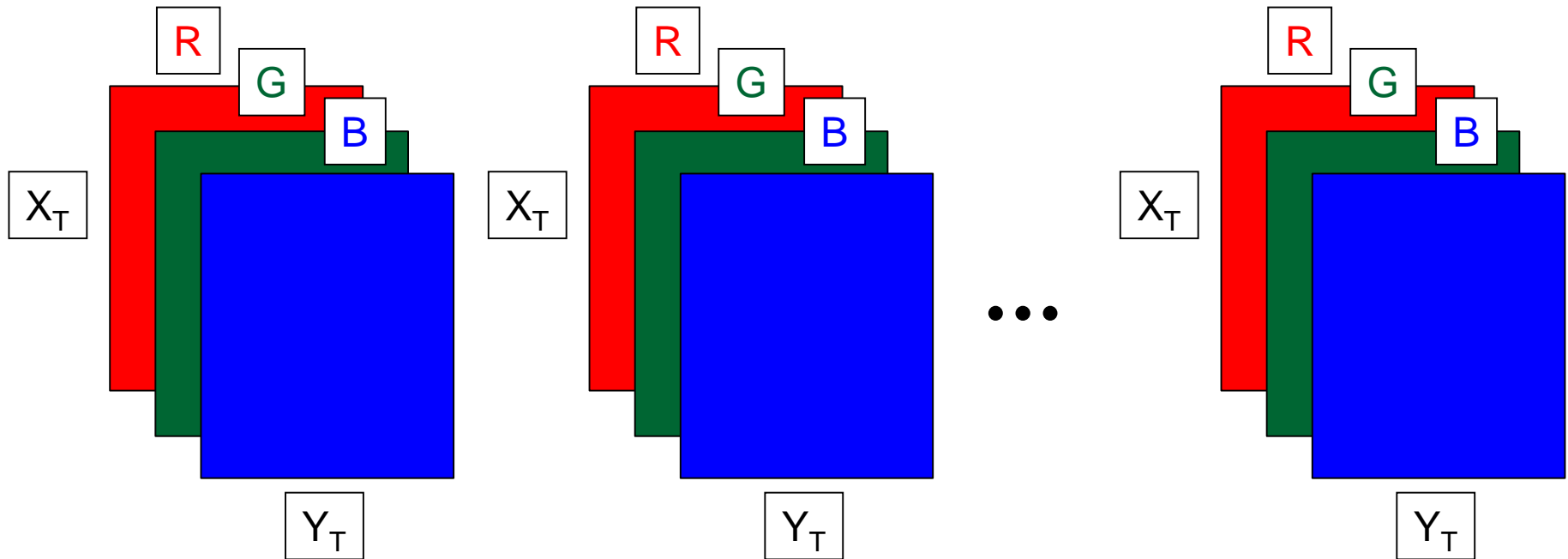
カラー画像の場合  
 $X_T \times Y_T \times 3$ チャンネル



グレースケール画像の場合  
 $X_T \times Y_T \times 1$ チャンネル

# カラー画像(デジタル画像)④

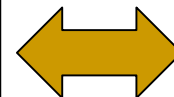
- さらに最近(?), 画像は(画像の枚数  $\times$   $X_T \times Y_T \times$  チャンネル数)の配列で処理



# 類似度①

切り出した画像  
の特徴ベクトル

n次元  $\mathbf{x}_p$



テンプレートの  
特徴ベクトル

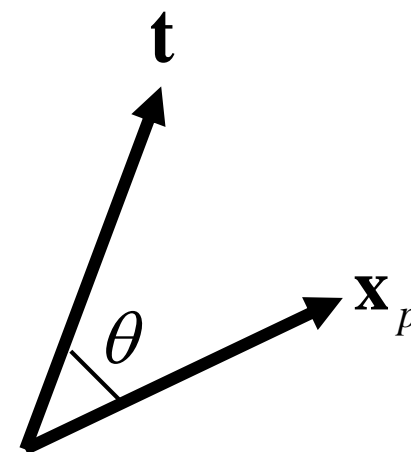
n次元  $\mathbf{t}$



## ■ 類似度

- 二つのベクトルが一致する場合は  $\theta = 0$
- 従って  $R_p$  は1となる

$$R_p = \cos \theta = \frac{\mathbf{t}^t \mathbf{x}_p}{\|\mathbf{t}\| \cdot \|\mathbf{x}_p\|} = \frac{\sum_{i=1}^n t_i x_{pi}}{\sqrt{\sum_{i=1}^n t_i^2} \sqrt{\sum_{i=1}^n x_{pi}^2}}$$



- $R_p$  が**最大の画像**を認識結果とする

# 類似度②

## ■ 相互相関係数

### □ 特徴ベクトルが n次元の場合

$$R'_p = \cos \theta = \frac{\sum_{i=1}^n (t_i - \bar{t})(x_{pi} - \overline{x_p})}{\sqrt{\sum_{i=1}^n (t_i - \bar{t})^2} \sqrt{\sum_{i=1}^n (x_{pi} - \overline{x_p})^2}}$$

平均値

$$\bar{t} = \frac{1}{n} \sum_{i=1}^n t_i$$

$$\overline{x_p} = \frac{1}{n} \sum_{i=1}^n x_{pi}$$

# 距離を用いた場合

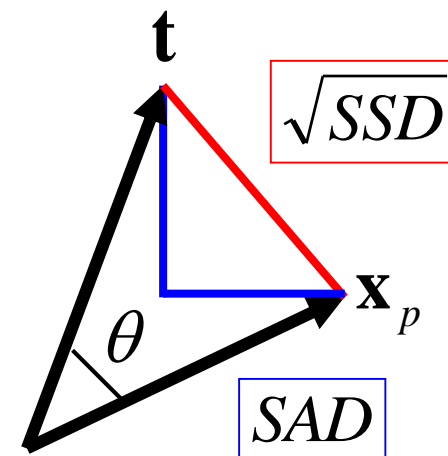
- SSD (Sum of Squared Difference)

$$SSD = \sum_{i=1}^n (t_i - x_{pi})^2$$

- SAD (Sum of Absolute Difference)

$$SAD = \sum_{i=1}^n |t_i - x_{pi}|$$

- 距離を用いた場合, SSDもしくはSADが最小の画像を認識結果とする





# 距離尺度①

## ■ ユークリッド距離

$$\sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

## ■ べき乗距離(ミンコスキー距離)

$$\left( \sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

p=r=1の場合, マンハッタン距離  
p=r=2の場合, ユークリッド距離

# 距離尺度②

## ■ チェビシェフ距離

$$\max_{i=1,2,\dots,n} |x_i - y_i|$$

各特徴間要素の最大値を距離とする

## ■ マハラノビス距離

- 分布を考慮
- 次回以降に説明します

# 類似度のまとめ

- 二つのベクトル間 (x と y) の類似度

n次元ベクトル

大きいほど類似している

- 各要素の差の合計 (距離)

$$\left( \sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

小さいほど類似している

# 空間フィルタリング処理(畳み込み処理)①

- 各画素について、その画素周辺の $N \times N$ 画素の小領域と、 $N \times N$ の空間フィルタとの積和を行なう
- 入力画像を $f$ 、空間フィルタを $h$ とした場合、下記の式に基づいて変換後の画素値 $g$ を求める

畳み込み処理

$$g(x, y) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} h(k, l) f(x+k, y+l)$$

## 空間フィルタリング処理(畳み込み処理)②

3×3の空間フィルタ h

0	-1	0
-1	5	-1
0	-1	0

変換する画素 (x,y)

40	60	40
80	110	80
100	100	100

$$\begin{aligned} &40 \times 0 + 60 \times (-1) + 40 \times 0 + \\ &80 \times (-1) + 110 \times 5 + 80 \times (-1) + \\ &100 \times 0 + 100 \times (-1) + 100 \times 0 \\ &= 230 \end{aligned}$$

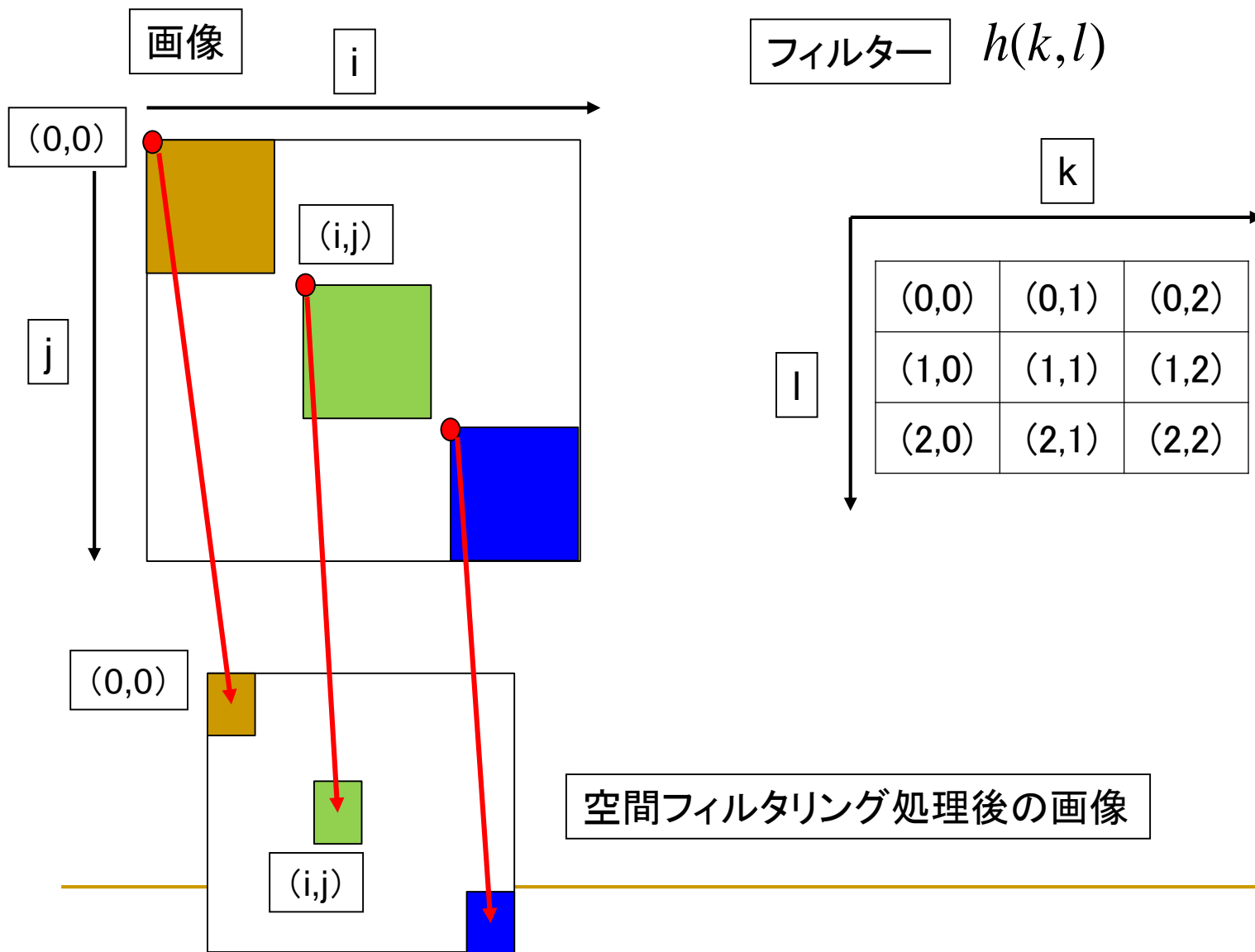
変換後の画素値 g

230

周辺の3×3の小領域 f

以上の処理を全ての画素で行なう

# 空間フィルタリング処理(畳み込み処理)③





# 空間フィルタリング処理(畳み込み処理)④

入力画像  $f(x,y)$

2	4	1	3	5
3	2	6	2	8
1	0	3	4	2
6	2	1	7	5
5	3	2	5	6

平滑化フィルタ  $h$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

出力画像  $g(x,y)$

$$(2+4+1+3+2+6+1+0+3)/9$$

2.444444	2.777778	3.777778
2.666667	3	4.222222
2.555556	3	3.888889

$$(3+4+2+1+7+5+2+5+6)/9$$

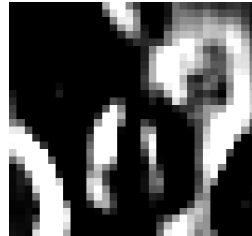
元の画像よりも小さくなってしまいますが、同じ大きさにすることも可能です(パディング)

# 空間フィルタリング処理(畳み込み処理)⑤

元画像



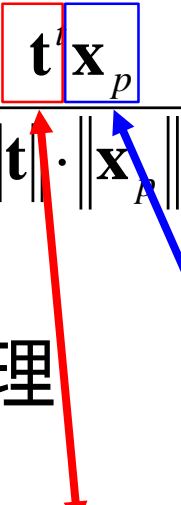
ガボールフィルター



フィルタリング後の画像

# テンプレートマッチングと畳み込み処理①

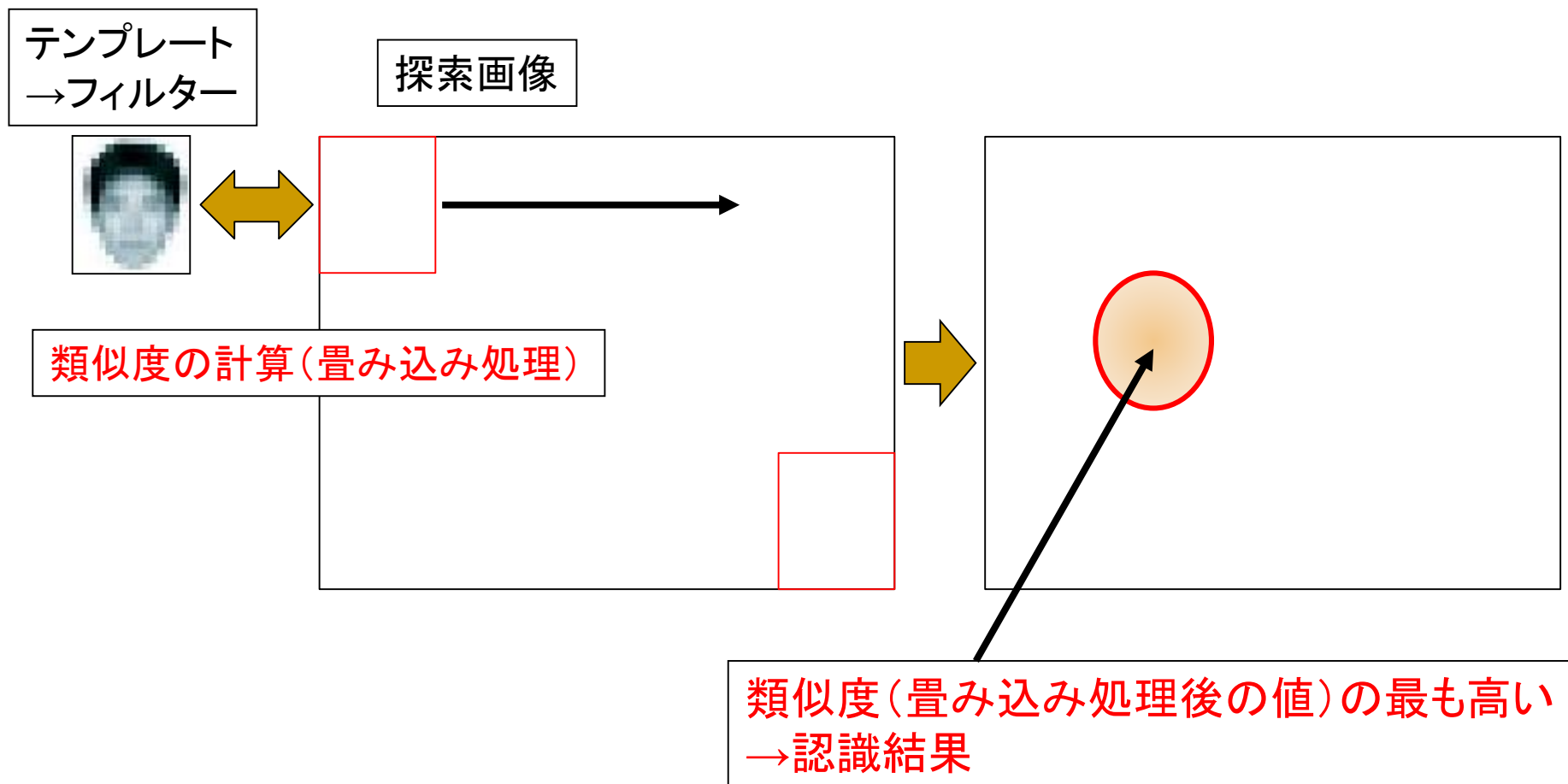
## ■ 類似度

$$R_p = \cos \theta = \frac{\mathbf{t}^T \mathbf{x}_p}{\|\mathbf{t}\| \cdot \|\mathbf{x}_p\|}$$


## ■ 畳み込み処理

$$g(x, y) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} h(k, l) f(x+k, y+l)$$

# テンプレートマッチングと畳み込み処理②



# テンプレートマッチングの工夫①(高速化)

- SSD (Sum of Squared Difference)

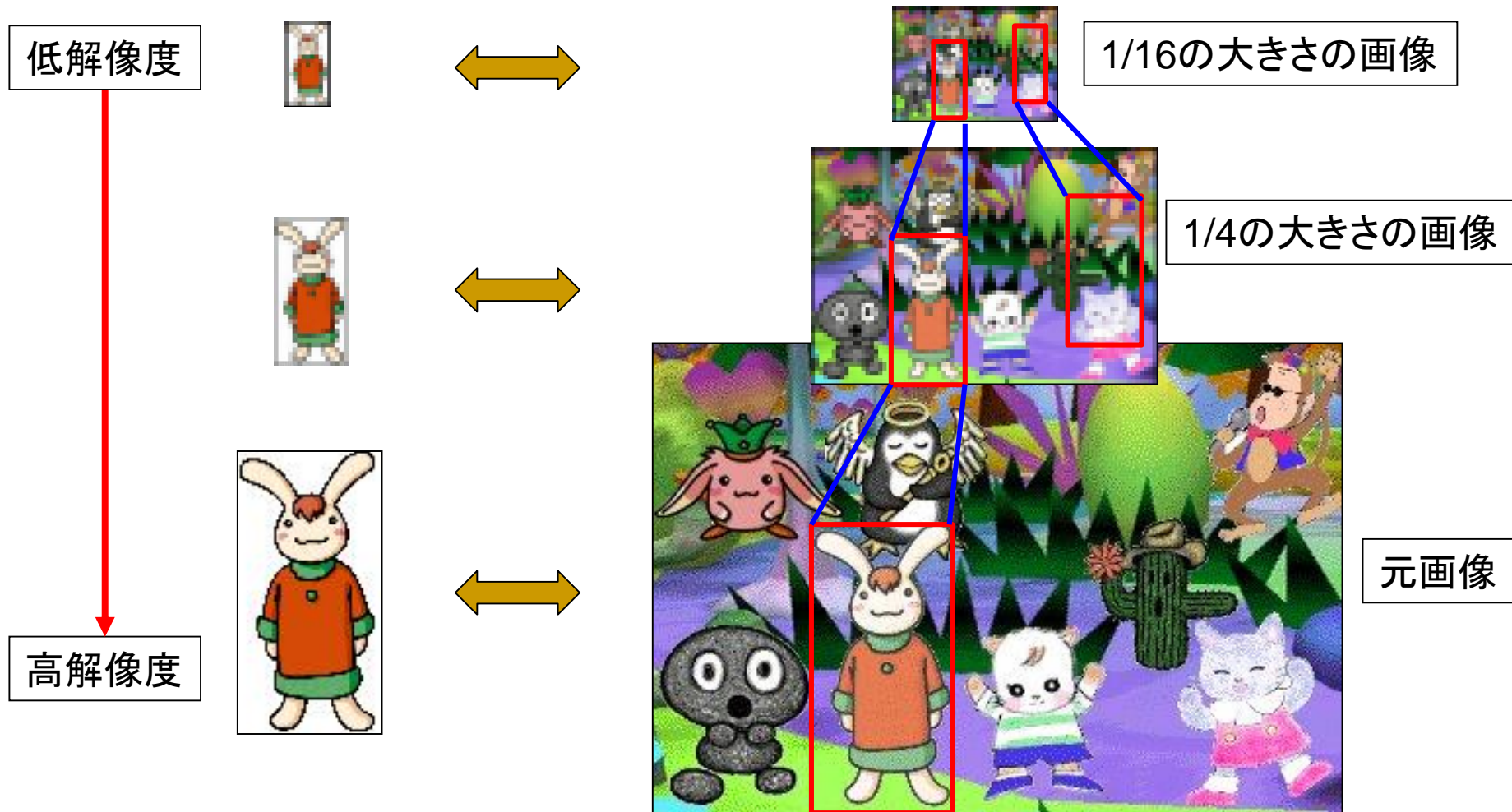
$$SSD = \sum_{i=1}^n (t_i - x_{pi})^2$$

- SAD (Sum of Absolute Difference)

$$SAD = \sum_{i=1}^n |t_i - x_{pi}|$$

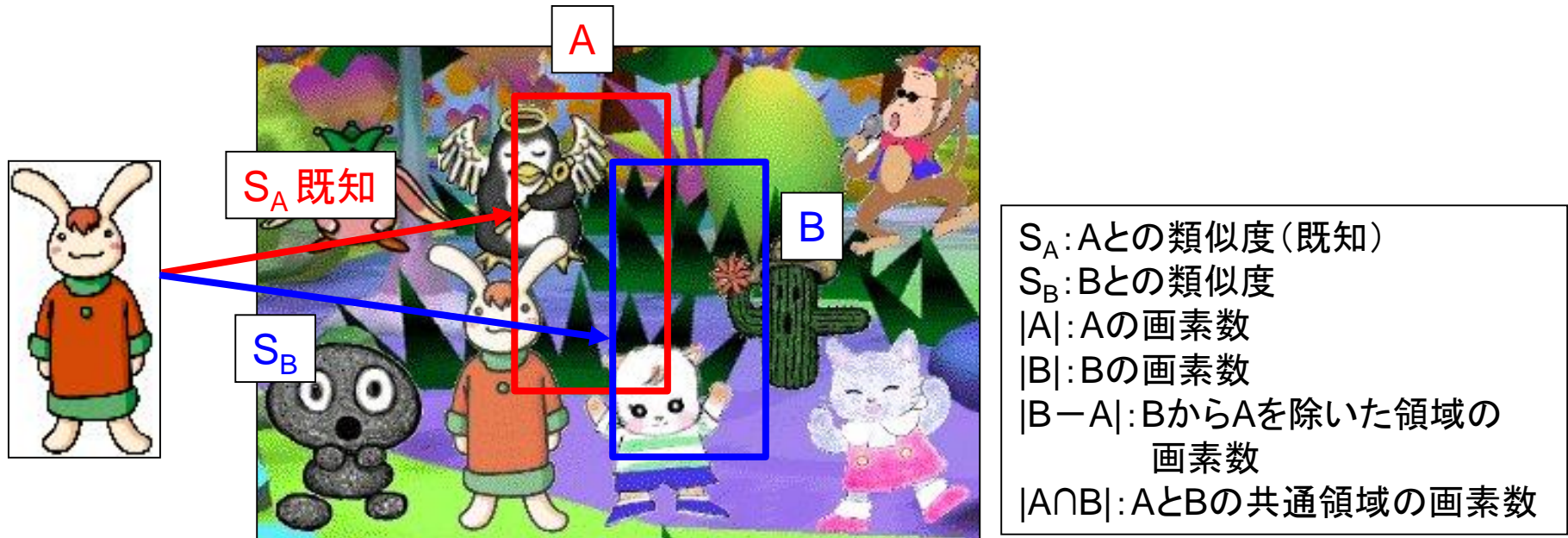
- SSD, SADを求める際, 閾値を超えたら停止し, 次の探索位置へ移動
- 閾値 = それまでの最小値

# テンプレートマッチングの工夫②(ピラミッド探索)





# テンプレートマッチングの工夫③(高速化)



Bとの類似度の上限値  $S_{up}$

→これまでの最大類似度より大きい場合のみ,  $S_B$ を計算

$$S_{up} = \frac{\min(S_A |A|, |A \cap B|) + |B - A|}{|B|}$$

# 問題点

- 大きさが異なる場合
- 傾きがある(回転している)場合
- 他の図形と重なっている(オクルージョン)場合



「うさぎ」は図中のどこにいるでしょうか

# テンプレートマッチングの例題

例題①

例題②(ステレオマッチング)

# テンプレートマッチングの例題①

- うさぎ (template.jpg) は右図 (search.jpg) のどこにいるかを調べるプログラム (Python)

template.jpg



search.jpg



# テンプレートマッチングのプログラム

- matching.py
  - 画素単位での実行
- matching\_numpy.py
  - ベクトル演算(numpy)での実行

# 実行方法

## ■ 必要なパッケージ

- numpy, PIL (pillow) が必要
- 理工学ITCのAnaconda上にはインストール済
- 仮想環境下ではインストールが必要
- > pip install numpy
- > pip install pillow

## ■ 実行方法

- > python matching.py
- > python matching-numpy.py

# 探索画像の読み込み(matching.py)①

matching.py

```
import sys
import os
from PIL import Image, ImageDraw
```

ライブラリのインポート

```
# 探索画像の読み込み
```

```
search_file = "search.jpg"
```

RGB画像として読み込み

```
search_img = Image.open(search_file).convert('RGB')
```

```
search_width , search_height = search_img.size
```

```
print( search_width , search_height )
```

size[0] 横の長さ  
size[1] 縦の長さ

search\_img

探索画像を読み込む変数



# 探索画像の読み込み(matching.py)②

```
search_img = Image.open(search_file).convert('RGB')
```

読み込んだ探索画像

search.jpg

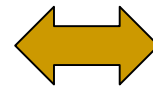
search\_width

search\_height



(x,y)のRGB値

search\_img



search\_height

search\_img.getpixel((x,y))  
(0~255, 0~255, 0~255)

search\_width



# テンプレートの読み込み (matching.py) ①

```
# テンプレートの読み込み
```

```
template_file = "template.jpg"
```

RGB画像として読み込み

```
template_img = Image.open(template_file).convert('RGB')
```

```
template_width , template_height = template_img.size
```

```
print( template_width , template_height )
```

size[0] 横の長さ  
size[1] 縦の長さ

template\_img

テンプレートを読み込む変数

# テンプレートの読み込み (matching.py) ②

```
template_img = Image.open(template_file).convert('RGB')
```

テンプレートの画像

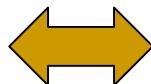
template\_img

template\_width

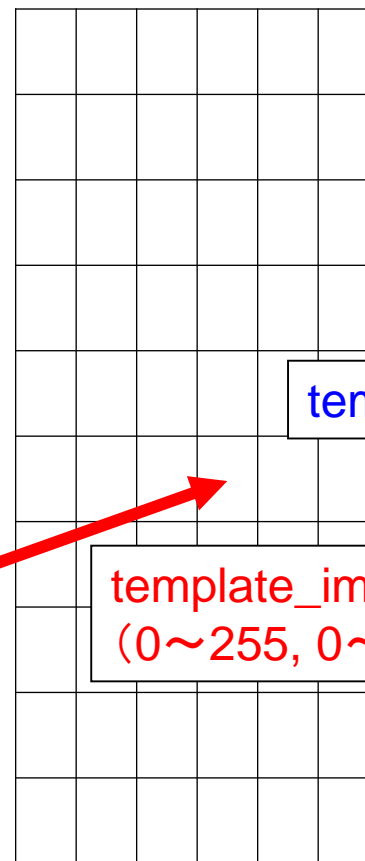
template.jpg



template\_height



(x,y) のRGB値



template\_height

template\_img.getpixel((x,y))  
(0~255, 0~255, 0~255)

template\_width

# 画像の読み込みのまとめ

- 探索画像 (search.jpg)
  - 大きさ search\_width × search\_height
  - 変数 search\_img
  - 画素値 search\_img.getpixel((x,y))
  - RGB値 (整数値) (0～255, 0～255, 0～255)
- テンプレート画像 (template.jpg)
  - 大きさ template\_width × template\_height
  - 変数 template\_img
  - 画素値 template\_img.getpixel((x,y))
  - RGB値 (整数値) (0～255, 0～255, 0～255)

# テンプレートマッチングのプログラム (matching.py)

```
# テンプレートマッチング
```

```
min_val = float('inf')
```

```
ans_x = 0
```

```
ans_y = 0
```

```
for y in range(0,search_height-template_height,5):
```

```
    for x in range(0,search_width-template_width,5):
```

```
        sum = 0.0
```

```
        # SSDの計算
```

```
        for yy in range(template_height):
```

```
            for xx in range(template_width):
```

```
                s = search_img.getpixel((x+xx,y+yy))
```

```
                t = template_img.getpixel((xx,yy))
```

```
                for i in range(3):
```

```
                    sum += ( s[i] - t[i] ) * ( s[i] - t[i] )
```

```
# 最小値を記憶
```

```
if min_val > sum:
```

```
    min_val = sum
```

```
    ans_x = x
```

```
    ans_y = y
```

結果を格納する座標  
(ans\_x,ans\_y)

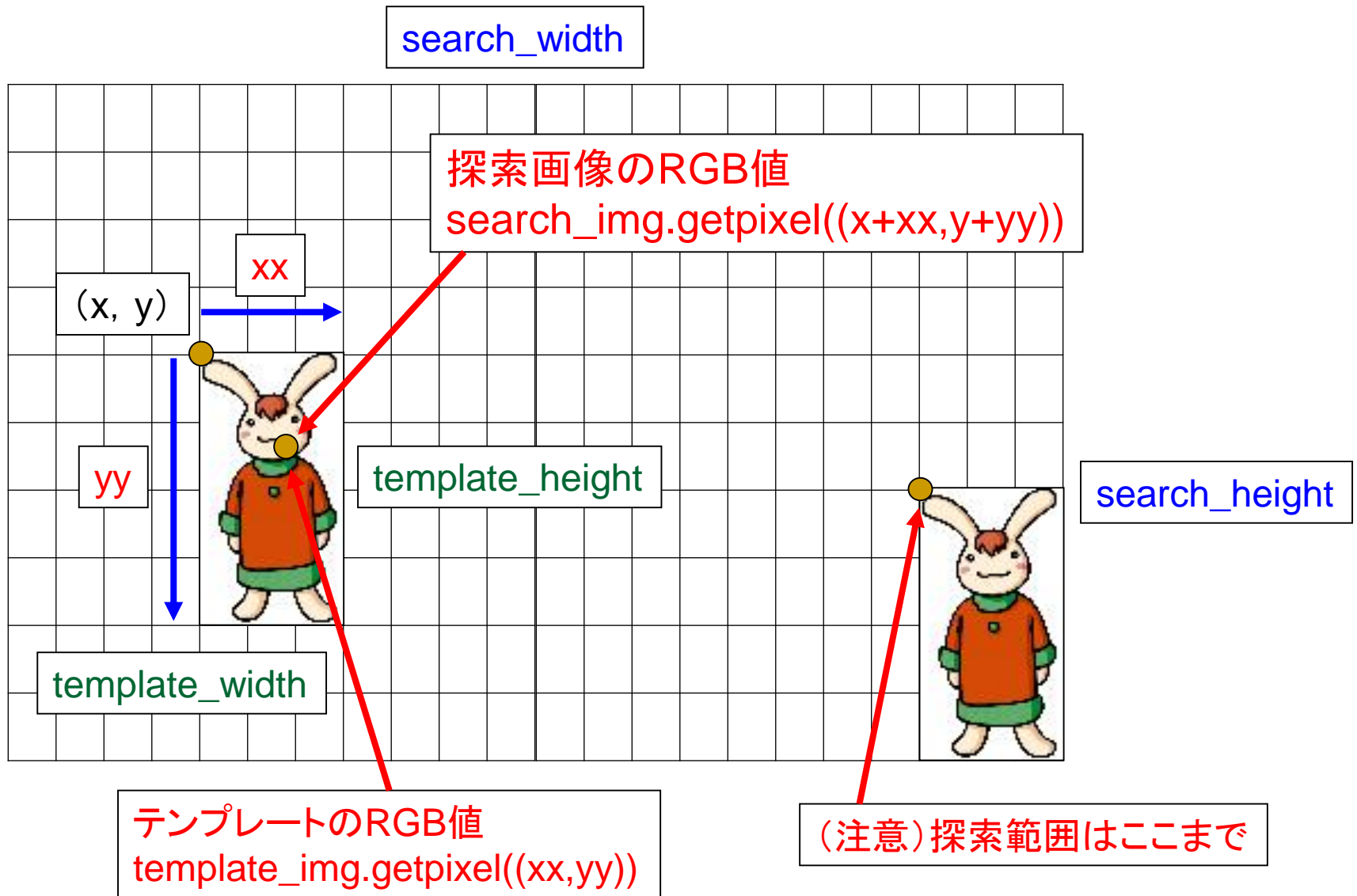
探索できる範囲に注意

計算時間がかかるので、このプログラム  
では5画素ごとに探索

(s[0],s[1],s[2])→(R,G,B)

SSDの計算

SSDの最小値、その座標を保存



# テンプレートと探索画像の座標関係

search\_img

template\_img

$(x,y)$

$(x+template\_width-1,y)$

$(0,0)$

$(template\_width-1,0)$

$(x+template\_width-1,y+template\_height-1)$

$(x,y+template\_height-1)$

$(template\_width-1,template\_height-1)$

$(0,template\_height-1)$

# 結果の表示(matching.py)①

# 枠の描画

```
draw = ImageDraw.Draw(search_img)
```

```
draw.rectangle((ans_x, ans_y, ans_x+template_width, ans_y+template_height),  
outline=(255,0,0))
```

左上座標(ans\_x,ans\_y)  
横template\_width, 縦template\_height  
の大きさの四角形を描画

# 結果の保存

```
search_img.save("result.jpg")
```

「result.jpg」に保存

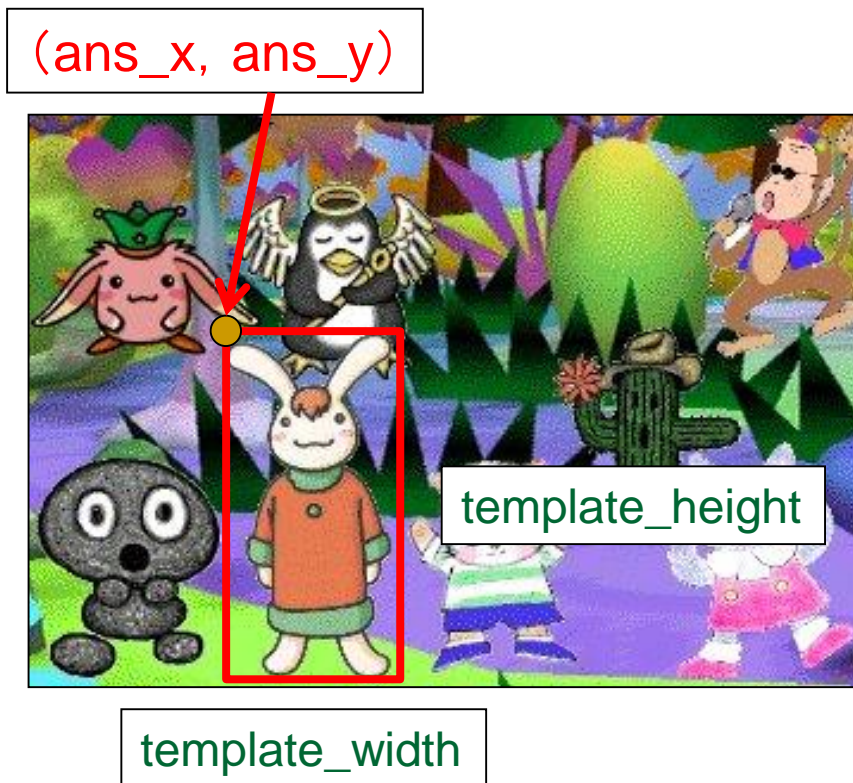
# 結果の表示

```
search_img.show()
```

結果の表示

# 結果の表示 (matching.py) ②

- 枠の描画→「result.jpg」に保存





# 探索画像の読み込み (matching\_numpy.py) ①

matching\_numpy.py

```
import sys
import os
import numpy as np
from PIL import Image, ImageDraw
```

ライブラリのインポート

```
# 探索画像の読み込み
search_file = "search.jpg"
search_img = Image.open(search_file).convert('RGB')
```

RGB画像として読み込み

```
# numpyに変換(Y,X,channel)
search = np.asarray(search_img).astype(np.float32)
print( search.shape )
search_width = search.shape[1]
search_height = search.shape[0]
print( search_width , search_height )
```

shape[0] 縦の長さ  
shape[1] 横の長さ  
shape[2] チャンネル数(この場合は3)

search  
探索画像を読み込む変数

# 探索画像の読み込み (matching\_numpy.py) ②

```
search_img = Image.open(search_file).convert('RGB')  
search = np.asarray(search_img).astype(np.float32)
```

読み込んだ探索画像

search.jpg

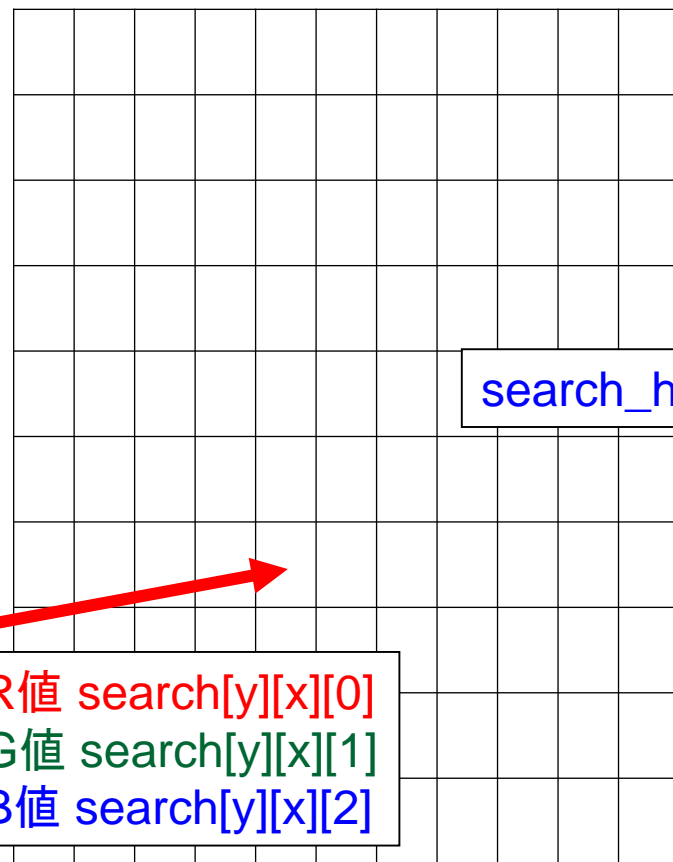
search\_width

search\_height



(x,y) のRGB値

search (三次元配列)



search\_height

R値 search[y][x][0]  
G値 search[y][x][1]  
B値 search[y][x][2]

search\_width

# テンプレートの読み込み (matching\_numpy.py) ①

```
# テンプレートの読み込み
```

```
template_file = "template.jpg"
```

```
template_img = Image.open(template_file).convert('RGB')
```

RGB画像として読み込み

```
# numpyに変換
```

```
template = np.asarray(template_img).astype(np.float32)
```

```
print( template.shape )
```

```
template_width = template.shape[1]
```

```
template_height = template.shape[0]
```

```
print( template_width , template_height )
```

shape[0] 縦の長さ

shape[1] 横の長さ

shape[2] チャンネル数(この場合は3)

template

テンプレートを読み込む変数

# テンプレートの読み込み (matching\_numpy.py) ②

```
template_img = Image.open(template_file).convert('RGB')  
template = np.asarray(template_img).astype(np.float32)
```

テンプレートの画像

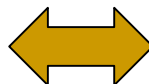
template (三次元配列)

template\_width

template.jpg



template\_height



(x,y)のRGB値

template\_height

R値 `template[y][x][0]`  
G値 `template[y][x][1]`  
B値 `template[y][x][2]`

template\_width

# テンプレートマッチングのプログラム (matching\_numpy.py)

numpy演算を利用しない場合

```
# テンプレートマッチング
```

```
min_val = float('inf')
```

```
ans_x = 0
```

```
ans_y = 0
```

```
for y in range(0,search_height-template_height,5):
```

```
    print(y)
```

```
    for x in range(0,search_width-template_width,5):
```

```
        sum = 0.0
```

```
        # SSDの計算
```

```
        for i in range(3):
```

```
            for yy in range(template_height):
```

```
                for xx in range(template_width):
```

```
                    sum += ( search[y+yy][x+xx][i] - template[yy][xx][i] ) *  
                           ( search[y+yy][x+xx][i] - template[yy][xx][i] )
```

```
# 最小値を記憶
```

```
if min_val > sum:
```

```
    min_val = sum
```

```
    ans_x = x
```

```
    ans_y = y
```

結果を格納する座標  
(ans\_x,ans\_y)

探索できる範囲に注意

プログラム中ではコメントしている

SSDの計算

SSDの最小値, その座標を保存

# テンプレートマッチングのプログラム (matching\_numpy.py)

numpy演算を利用する場合

```
# テンプレートマッチング
```

```
min_val = float('inf')
```

```
ans_x = 0
```

```
ans_y = 0
```

```
for y in range(0,search_height-template_height,5):
```

```
    for x in range(0,search_width-template_width,5):
```

```
        sum = 0.0
```

```
        # SSDの計算
```

```
        s = search[y:y+template_height,x:x+template_width,:].flatten()
```

```
        t = template.flatten()
```

```
        sum = np.dot( (t-s).T , (t-s) )
```

結果を格納する座標  
(ans\_x,ans\_y)

探索できる範囲に注意

SSDの計算

```
# 最小値を記憶
```

```
if min_val > sum:
```

```
    min_val = sum
```

```
    ans_x = x
```

```
    ans_y = y
```

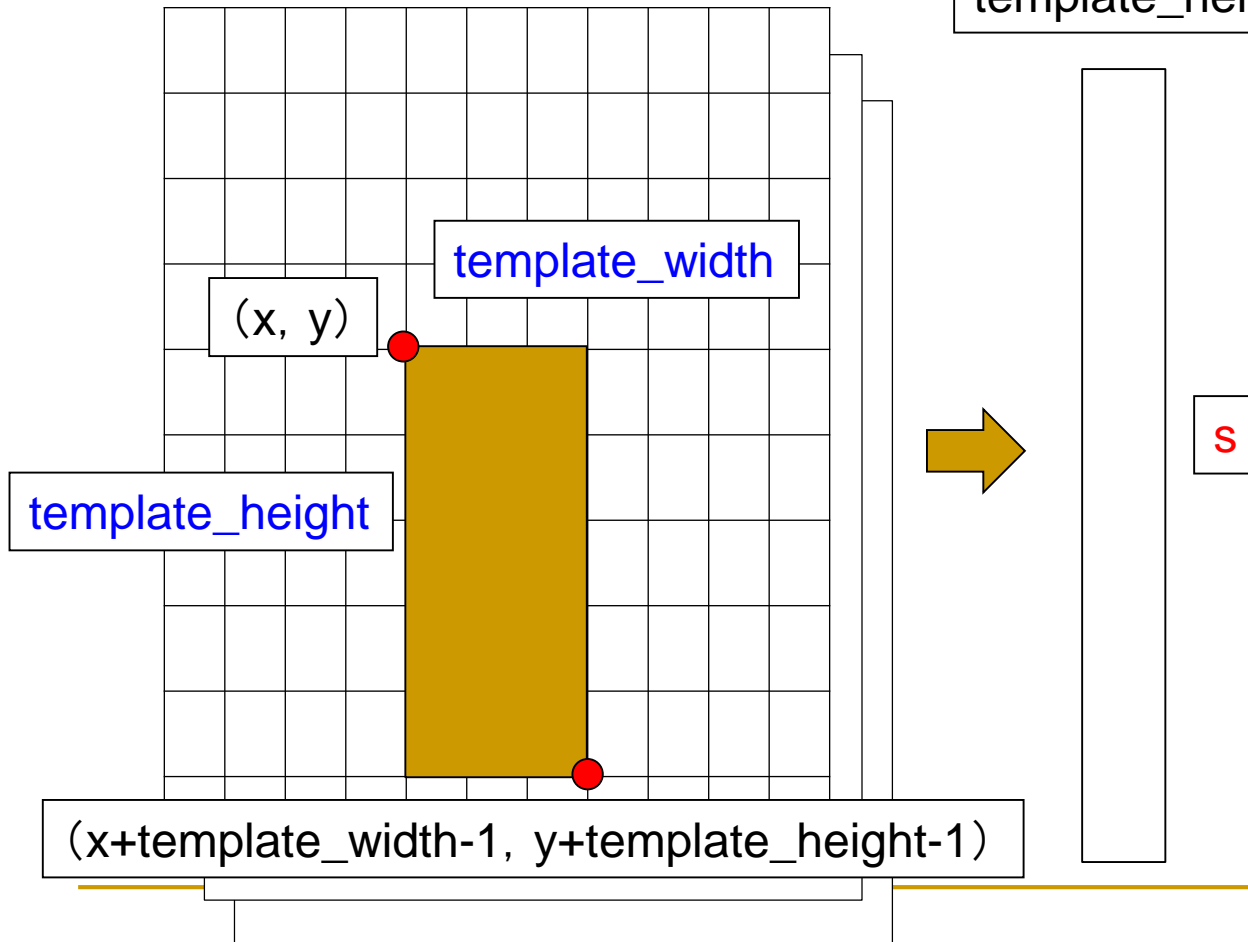
SSDの最小値, その座標を保存

# SSDの計算①

```
s = search[y:y+template_height,x:x+template_width,0:3].flatten()
```

search(三次元配列)

$\text{template\_height} \times \text{template\_width} \times 3$



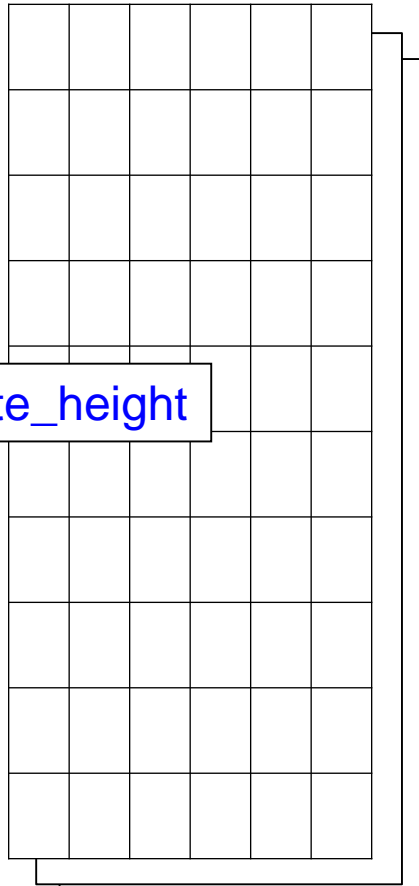
## SSDの計算②

```
t = template.flatten()
```

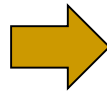
template (三次元配列)

template\_height × template\_width × 3

template\_height



template\_width



t

$$SSD = (\mathbf{t} - \mathbf{x}_p)^t (\mathbf{t} - \mathbf{x}_p) = \sum_{i=1}^n (t_i - x_{pi})^2$$

転置

```
sum = np.dot( (t-s).T , (t-s) )
```

内積



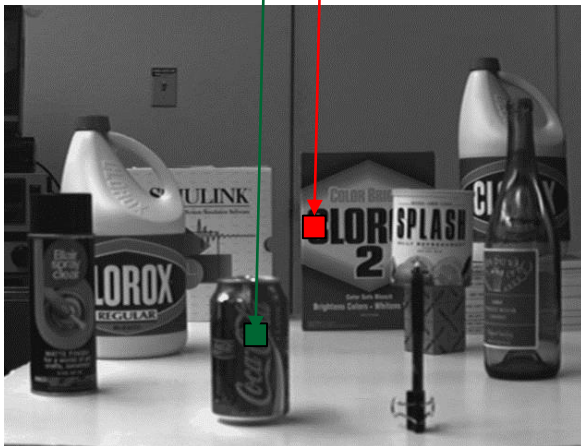
# ステレオマッチング(例題②)

ステレオ画像

左画像



右画像



左画像と右画像の位置に差が生じる

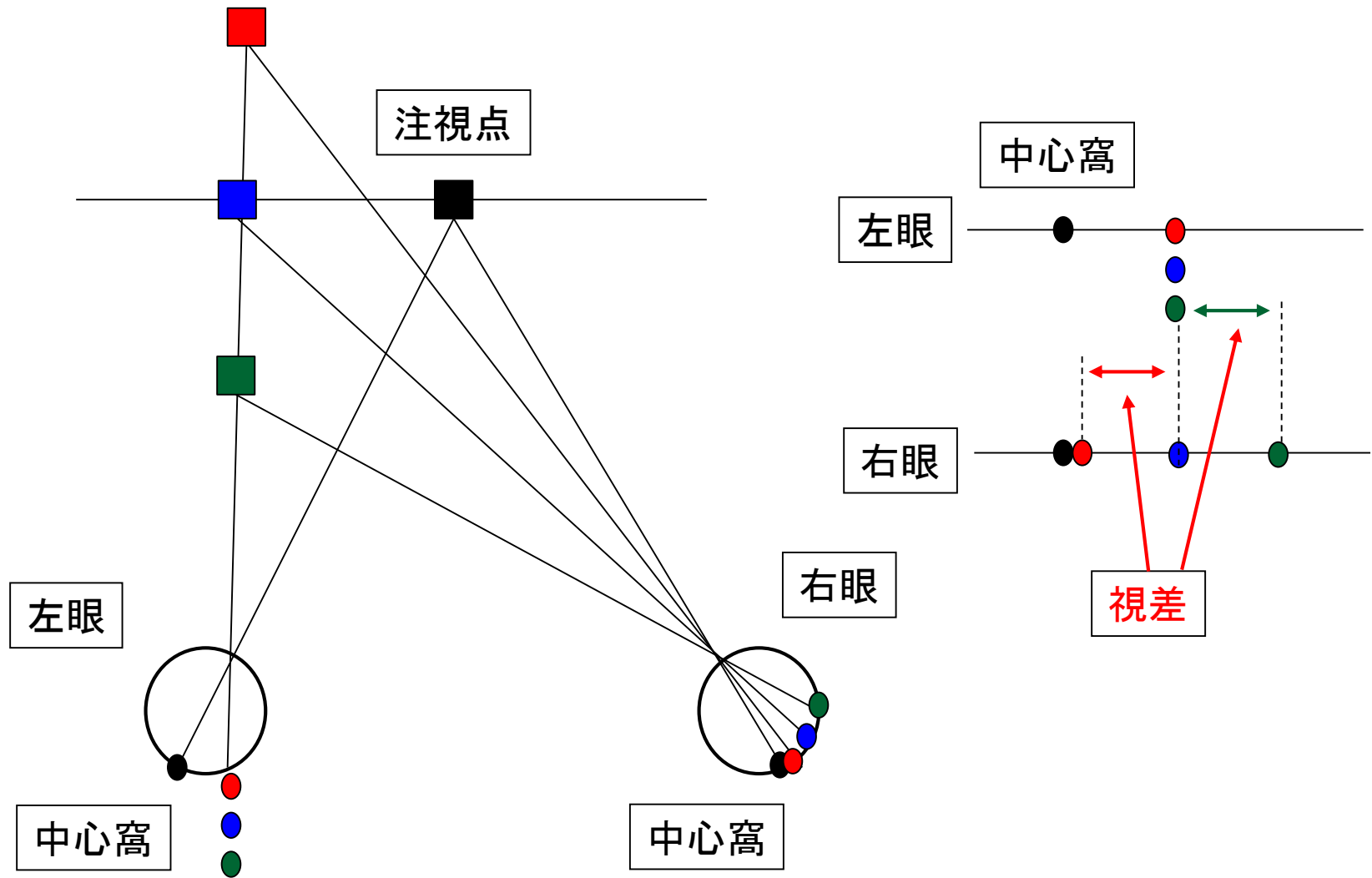


「近く」と「遠く」にある物においては、その差(の量)にも違いが生じる



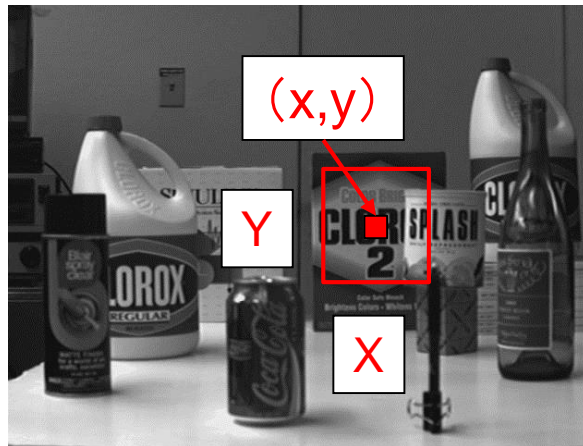
これを「**視差**」と呼び、奥行き(距離ではない)を認識することができる

# 両眼視差



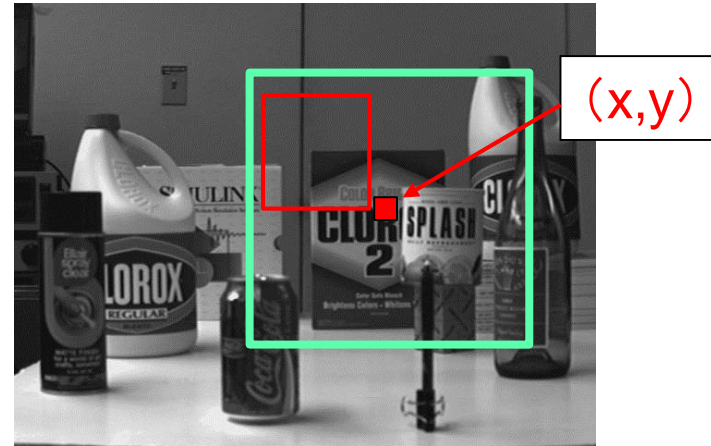
# ステレオマッチングの原理①

左画像



左画像の座標  $(x,y)$  を中心に、大きさ  $(X \times Y)$  のテンプレート(赤枠)を切り出す

右画像

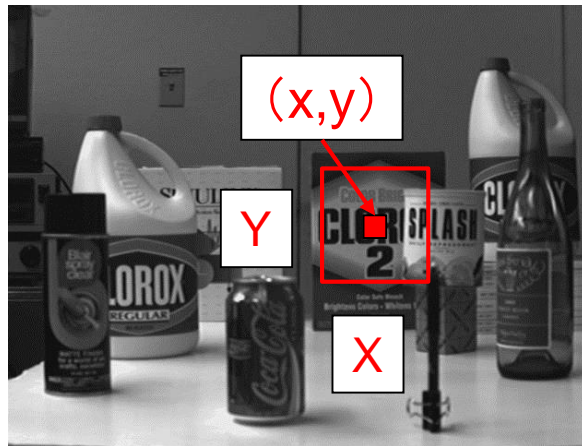


右画像の座標  $(x,y)$  を中心に、一定範囲内(緑枠)\*で、テンプレートと最も類似した領域を探索(テンプレートマッチング)

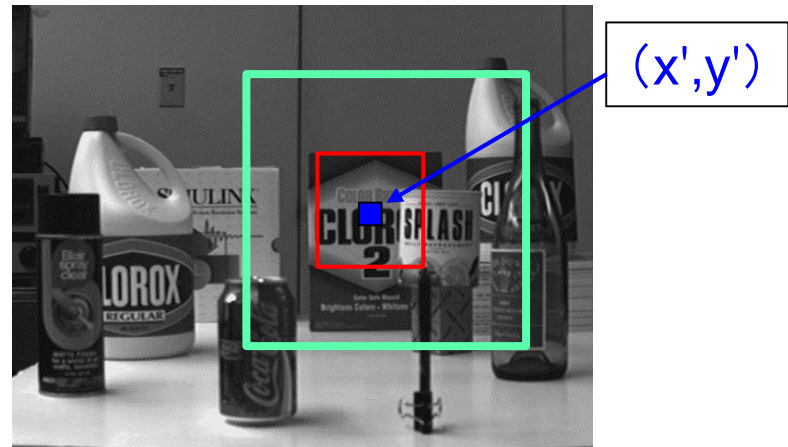
\*計算時間の削減のため、y軸方向には探索しなくても良い

# ステレオマッチングの原理②

左画像



右画像



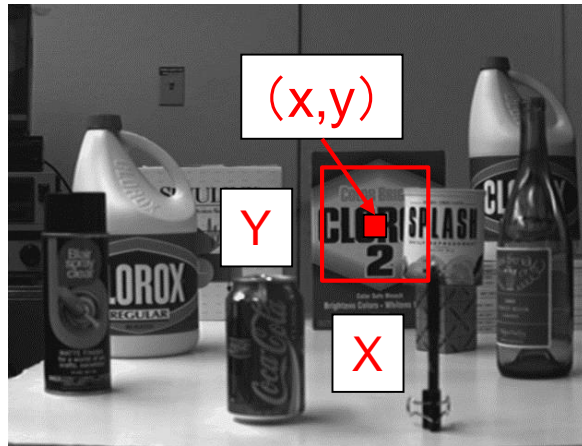
テンプレートマッチングの結果の座標  $(x',y')$

視差

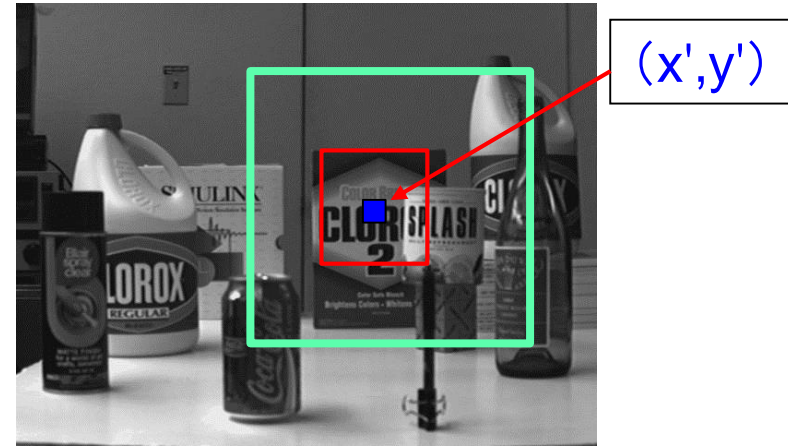
$$d = (x - x')^2 + (y - y')^2$$

# ステレオマッチングの原理②

左画像



右画像



全ての画素において視差を求める



視差マップ

白領域→近い  
黒領域→遠い

\*0~255の範囲に正規化

# ステレオマッチングのプログラム①

## (stereo\_matching.py)

### 左画像の読み込み

```
import sys
import os
import numpy as np
from PIL import Image
```

ライブラリのインポート

```
# 左画像の読み込み
```

```
left_file = "left.jpg"
```

```
left_img = Image.open(left_file).convert('L')
```

グレースケール画像として読み込み

```
# numpyに変換 -> (Y,X,channel)
```

```
left = np.asarray(left_img).astype(np.float32)
```

```
print( left.shape )
```

```
left_width = left.shape[1]
```

```
left_height = left.shape[0]
```

```
print( left_width , left_height )
```

shape[0] 縦の長さ  
shape[1] 横の長さ  
shape[2] チャンネル数(この場合は1)

# ステレオマッチングのプログラム②

## (stereo\_matching.py)

### 右画像の読み込み

```
# 右画像の読み込み
right_file = "right.jpg"
right_img = Image.open(right_file).convert('L')
```

グレースケール画像として読み込み

```
# numpyに変換 -> (Y,X,channel)
right = np.asarray(right_img).astype(np.float32)
print( right.shape )
right_width = right.shape[1]
right_height = right.shape[0]
print( right_width , right_height )
```

shape[0] 縦の長さ  
shape[1] 横の長さ  
shape[2] チャンネル数(この場合は1)

```
# 視差マップ
result = np.ones((left_height, left_width))
```

視差マップの二次元配列

# ステレオマッチングのプログラム③

## (stereo\_matching.py)

```
# 探索領域の大きさ
search_size = 21 // 2
# テンプレートの大きさ
template_size = 21 // 2
for y in range(0, left_height, 1):
    for x in range(0, left_width, 1):
        ans_x = 0
        ans_y = 0
        min_val = float( 'inf' )
```

左画像の座標(x,y)を中心としたテンプレートに最も類似した領域を  
右画像から探索し, その座標(ans\_x,ans\_y)を求めなさい

```
result[y,x]=(x-ans_x)*(x-ans_x)+(y-ans_y)*(y-ans_y)
```



# 左画像から切り出すテンプレート

左画像

配列left

$(x - \text{template\_size}, y - \text{template\_size})$

$(x + \text{template\_size}, y - \text{template\_size})$

$(x, y)$

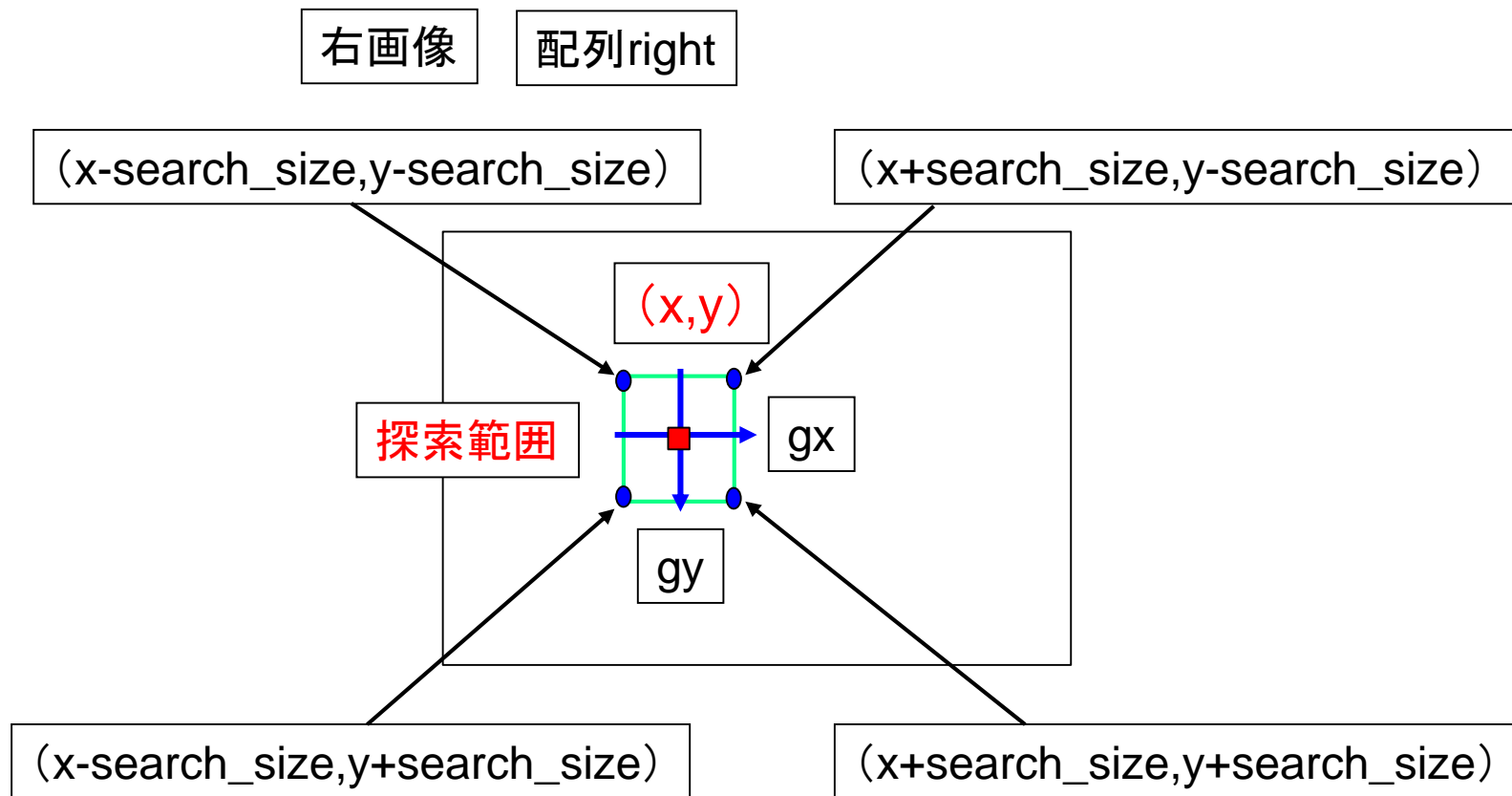
テンプレート

左画像の座標  $(x, y)$  を中心に、縦横の大きさが  $(\text{template\_size} \times 2 + 1)$  のテンプレートを切り出す

$(x - \text{template\_size}, y + \text{template\_size})$

$(x + \text{template\_size}, y + \text{template\_size})$

# 右画像でのテンプレートマッチング①



\*繰り返しになりますが計算時間の削減のため、y軸方向には探索しなくても良いです

## 右画像でのテンプレートマッチング②

右画像

配列right

$(x+gx-search\_size, y+gy-search\_size)$

$(x+gx+search\_size, y+gy-search\_size)$

赤枠: テンプレート

緑枠: 探索範囲

$(x+gx, y+gy)$

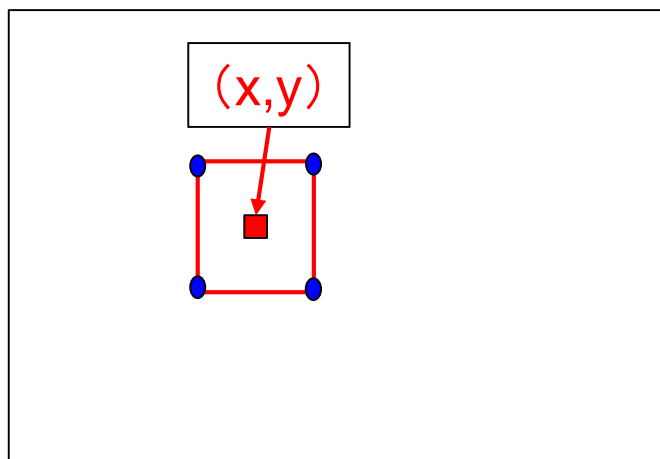
$(x+gx-search\_size, y+gy+search\_size)$

$(x+gx+search\_size, y+gy+search\_size)$

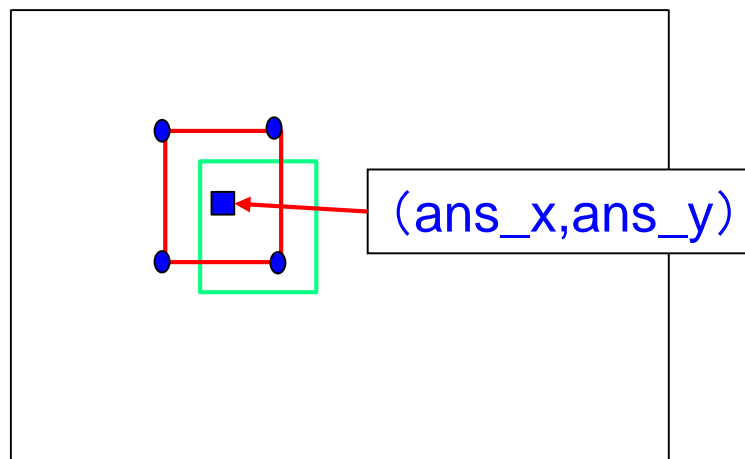
右画像の座標  $(x+gx, y+gy)$  を中心に、縦横の大きさが  $(template\_size \times 2 + 1)$  の画像とテンプレートの類似度を求める

# 視差の計算

左画像



右画像



視差

$$d = (x - x')^2 + (y - y')^2$$

```
result[y,x]=(x-ans_x)*(x-ans_x)+(y-ans_y)*(y-ans_y)
```

# ステレオマッチングのプログラム④

## (stereo\_matching.py)

```
min = np.min( result )  
max = np.max( result )  
result = (result-min)/(max-min) * 255
```

0～255に正規化

```
result_img = Image.fromarray(np.uint8(result))  
result_img.save('result.jpg')
```

numpyから画像に変換

「result.jpg」として保存

# 実行方法

## ■ 実行方法

- （完成しないと実行しません）
- > python stereo\_matching.py

# 実行例

左画像



右画像



視差マップ



# 宿題①

- ステレオマッチングのプログラム (stereo\_matching.py) を完成させなさい
- 類似度はSSD (もしくはSAD) を用いなさい
- テンプレートの大きさは  $(\text{template\_size} \times 2 + 1)$  です
- 探索範囲は  $(\text{search\_size} \times 2 + 1)$  です
- y軸方向には探索しなくてもかまいません



# (本日の)参考文献

- 石井健一郎他: わかりやすいパターン認識, オーム社(1998)
- 塩入諭, 大町真一郎: 画像情報処理工学, 朝倉書店(2011)
- 平井有三: はじめてのパターン認識, 森北出版(2012)