# プログラミング応用 9班 信号処理技術レポート


## 学籍番号：33114066

## 名前：高木　空

## メンバー

- 33114066 鈴木　聡
- 33114072 高木　空
- 31114056 佐藤　優樹


## 使用計算機環境情報

CSEの環境を使用した．

# 画像処理課題

レベル1，2，3の制作担当者：31114056 佐藤　優樹
レベル4，5，6の制作担当者：33114066 鈴木　聡

# レベル１

レベル１の課題を，下記コマンドでコンパイルオプションの違いによって実行時間がどのように変わるか計測を行った．
結果は表1に示す．

```
time sh run.sh level1
sh answer.sh result level1
```

表１：level1の結果と実行時間

| オプション | 検出率 | 実行時間 |
|---|---|---|
| -O1 | 10/10(100%) | 55.460s |
| -O2 | 10/10(100%) | 59.024s |
| -O3 | 10/10(100%) | 19.562s |

オプション-O1と，-O2は違いがなかった．
-O3が他の二つに比べて実行時間が早かったため，以下のレベルで，コンパイルオプションを-O3にして課題を行った．

# レベル２

レベル２に対応するために，下記のようにテンプレートマッチングの閾値を1.5に変更した．
結果は表2に示す．

```
if [ $x = 0 ]
then
    ./matching $name "${tname}" $rotation 1.5 cwp #元は0.5
    x=1
else
    ./matching $name "${tname}" $rotation 1.5 wp #元は0.5
fi
```

表2：level2の結果と実行時間

| level | 検出率 | 実行時間 |
|---|---|---|
| level2 | 10/10(100%) | 34.675s |

**考察**

画像にノイズが生じていることで，二乗誤差の総和が大きくなるため，マッチング成功の判定を広げる必要があったと考えられる．

# レベル３

レベル3はレベル2同様，閾値1.5に設定することで，検出率が100%になった．
結果は表3に示す．

表3：level3の結果と実行時間

| level | 検出率 | 実行時間 |
|---|---|---|
| level3 | 10/10(100%) | 17.289s |

**考察**

明るさが変わる画像をテンプレートマッチングするレベルであるが，画像の明るさが全体的に変化しているだけであるため，閾値を変えるだけで検出率100%が出せたと考えられる．
また，コントラストが変化する倍率が決まっているため，すべての倍率で類似度を計算すると，時間はかかるが閾値を変化させずとも，成功率100%を出せるのではないかと考える．

# レベル４

レベル４に対応するために，下記のように`main.c`の類似度の計算部分にif文を追加した．
これは背景透過のために，黒い部分の類似度の計算をしない目的で追加したものである．
結果は表4に示す．

```
//以下のif文を追加した
if(template->data[pt2 + 0] == 0 && template->data[pt2 + 1] == 0 &&
template->data[pt2 + 2] == 0){}
else{
    int r = (src->data[pt + 0] - template->data[pt2 + 0]);
    int g = (src->data[pt + 1] - template->data[pt2 + 1]);
    int b = (src->data[pt + 2] - template->data[pt2 + 2]);

    distance += (r * r + g * g + b * b);
}
```

表4：level4の結果と実行時間

| level | 検出率 | 実行時間 |
|---|---|---|
| level4 | 10/10(100%) | 47.791s |

**考察**

今回背景が黒一色だったため，単純なif文の追加だけで検出できたが，日常の背景など，様々な色が含まれた背景を持つテンプレート画像で検出する場合，高度な背景処理が必要だと考える。

# レベル５

レベル5に対応するために，下記のように，`run.sh`を書き換えた．
主な変更箇所は，拡大率が決まっているため，それぞれでfor文を回して，テンプレート画像を拡大縮小して，テンプレートマッチングを実装したことである．
結果は表5に示す．

```
for size in 50 100 200; do #拡大率それぞれでfor文を回す
    for template in $1/*.ppm; do
        tempname=`basename ${template}`
        tname="imgproc/"$tempname
        echo `basename ${tname}`
        #テンプレート画像のサイズを変更するImageMagickのコマンド
        convert -resize $size% "${template}" "${tname}"
        if [ $x = 0 ]
        then
            ./matching $name "${tname}" $rotation 0.4 cwp
            x=1
        else
            ./matching $name "${tname}" $rotation 0.4 wp
        fi
    done
done
```

表5：level5の結果と実行時間

| level | 検出率 | 実行時間 |
|---|---|---|
| level5 | 8/10(80%) | 251.609s |

**考察**

この課題では，検出率が100%にならなかった．マッチング失敗した画像を調べると，テンプレート画像が縮小されている画像の検出ができていないことが分かった．
これは，画像に埋め込まれた，縮小サイズのテンプレート画像の縮小方法と，ImageMagickの縮小方法が異なるため，うまく検出できなかったと考えられる．

# レベル６

レベル6に対応するために，下記のように，`run.sh`を書き換えた．
主な変更箇所は，回転角が決まっているため，それぞれでfor文を回して，テンプレート画像を回転して，テンプレートマッチングを実装したことである．
結果は表6に示す．

```
for rot in 0 90 180 270; do  #回転角それぞれでfor文をまわす
    for template in $1/*.ppm; do
        tempname=`basename ${template}`
        tname="imgproc/"$tempname
        #テンプレート画像を回転するImageMagickのコマンド
        convert -rotate $rot "${template}" "${tname}"
        if [ $x = 0 ]
        then
            ./matching $name "${tname}" $rot 0.5 cwp
            x=1
        else
            ./matching $name "${tname}" $rot 0.5 wp
        fi
    done
done
```

表6：level6の結果と実行時間

| level | 検出率 | 実行時間 |
|---|---|---|
| level6 | 10/10(100%) | 308.063s |

**考察**

レベル5とは違い，回転するだけでは画素は変わらないため，検出率を100%に出来たと考えられる．

# レベル７

レベル7は高速化処理のあとに行ったため，ここでは実装せず，finalで実装した．

# 高速化処理

高速化処理の実現，及びfinalの課題担当者：33114072 高木　空

最後に，finalの課題に向けて高速処理を実現するために行ったことを以下に記す．

- OpenCVの利用
- 画像読み込みの一括化
- 並列化
- 無駄な処理の削除
- 閾値の削除

**高速化について**

まずImageMagickは遅いので，OpenCVに変更した．画像読み込みは各ソースとテンプレートの組ごとに読み込むと何回も読み込みが発生するので全ての処理をプログラム内で行い，リソースを使いまわすことにより効率化を図った．また，各画像の組に対し並列化を行うことで時間のかかるテンプレートマッチング処理を分散させている．具体的にはpythonのconcurrentを用いて並列化を行っている． 最後に標準出力や分岐による結果の表示を全て削除し，可能な限り不必要な処理を削減している． また、グレースケール画像を用いることで計算量を減らした。

**精度について**

ここまでのプログラムでは必要に応じて閾値を変えることによって精度向上を行ったが，ここでは全てのレベルに対応するため，各ソースに対し，全てのテンプレートのうち，類似度が最大になる点を採用することにした．全ての画像にテンプレートが必ず一つ含まれる前提を用いている． level4に関しては上記のことを全て実装すると検出率が35%まで落ちてしまうので、マスクを用いた。マスクはテンプレート画像を2値化したもので、黒の部分をマッチングに使わないことで背景透過に対応している。

# FINAL

最終テストの結果を示す．

| level | 検出率 | 実行時間 |
|---|---|---|
| level1 | 20/20(100%) | 0.322s |
| level2 | 20/20(100%) | 0.318s |
| level3 | 20/20(100%) | 0.308s |
| level4 | 20/20(100%) | 1.035s |
| level5 | 20/20(100%) | 0.312s |
| level6 | 20/20(100%) | 0.333s |
| level7 | 19/20(95%) | 0.315s |

**考察**

OpenCVは内部でかなり効率化がされているので，ただ実行するだけでもかなり高速化された．Pythonはインタプリタ言語なのでC++などで実装する他，GPUなどを用いてテンプレートマッチング自体も高速化するとさらなる速度向上が期待できる．

```
def matching(src, tmp, mask):
    result = cv2.matchTemplate(src, tmp, cv2.TM_CCORR_NORMED)
    # _best, _, _bestPos, _ = cv2.minMaxLoc(result)  # min
    _, _best, _, _bestPos = cv2.minMaxLoc(result)  # max
    return _best, _bestPos

# 画像読み込み
level = sys.argv[1]
srcs_path = glob.glob(level + "/final/*.ppm")
tmps_path = glob.glob(level + "/*.ppm")

srcs_name = [src.split("/")[2][0:-4] for src in srcs_path]
tmps_name = [tmp.split("/")[1][0:-4] for tmp in tmps_path]

src = [cv2.imread(src) for src in srcs_path]
tmp = [[cv2.imread(tmp)] for tmp in tmps_path]

# 処理開始
num = 6

with futures.ThreadPoolExecutor(max_workers=64) as executor:
    # 回転，拡縮したテンプレートを生成
    for i in range(len(tmp)):
        tmp[i].append(cv2.resize(tmp[i][0], None, None, 0.5, 0.5))
        tmp[i].append(cv2.resize(tmp[i][0], None, None, 2.0, 2.0))
        tmp[i].append(cv2.rotate(tmp[i][0], cv2.ROTATE_90_CLOCKWISE))
        tmp[i].append(cv2.rotate(tmp[i][0], cv2.ROTATE_180))
        tmp[i].append(cv2.rotate(tmp[i][0],
cv2.ROTATE_90_COUNTERCLOCKWISE))
    # マスクを生成
    mask = [
        [cv2.threshold(_tmp[i], 0, 255, cv2.THRESH_BINARY)[1] for i in
range(num)]
        for _tmp in tmp
    ]

    result_list = []

    # テンプレートマッチングを実行
    for i in range(len(src)):
        for j in range(len(tmp)):
            for k in range(num):
                result = executor.submit(matching, src[i], tmp[j][k],
mask[j][k])
                result_list.append(result)

    # 結果を比較し，結果を出力
    for i in range(len(src)):
```

```
        best = -float("inf")
        bestPos = [0, 0]
        idx = 0
        kdx = 0
        for j in range(len(tmp)):
            for k in range(num):
                _best, _bestPos = result_list[i * len(tmp) * num + j * num
+ k].result()
                if best < _best:
                    best = _best
                    bestPos = _bestPos
                    idx = j
                    kdx = k
        file = open("result/" + srcs_name[i] + ".txt", "w")
        file.write(
            "{} {} {} {} {} {}".format(
                tmps_name[idx],
                bestPos[0],
                bestPos[1],
                tmp[idx][kdx].shape[1],
                tmp[idx][kdx].shape[0],
                0 if (kdx < 3) else (kdx - 2) * 90,
            )
        )
```

レポート作成者：33114066 鈴木　聡，33114072 高木 空