Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej

Sprawozdanie # 03

Analiza i Przetwarzanie Obrazów Biometrycznych

Ścienianie - KMM oraz K3M

Tomasz Koter

v1.0

1 gru 2016

Spis treści

1. Wstęp

Niniejsze sprawozdanie dotyczy laboratoriów # 4 oraz # 5 Analizy i Przetwarzania Obrazów Biometrycznych 2016Z, czyli algorytmów ścieniania KMM oraz K3M. Zadaniem była implementacja obu.

W celu realizacji tego zadania posłużyłem się Octave (na 99% programy są kompatybilne z Matlabem) ze względu na wygodę i szybkość obróbki obrazów za jego pomocą. Octave oferuje możliwość prostego przetwarzania całego obrazu za pomocą pojedynczej linii kodu zawierającej jakieś wyrażenie logiczne lub arytmetyczne, dzięki czemu na ogół można uniknąć pisania podwójnych pętli po kolejnych pikselach obrazu lub korzystania z generatorów, co nadal jest dłuższe niż jednolinijkowe wyrażenie.

Ponadto w podstawowym pakiecie oraz w rozszerzeniu *image* dostępne są przydatne funkcje, takie jak splot czy dodawanie marginesu do obrazu, co ułatwia korzystanie z masek (nie można wyjść poza zakres tablicy obrazu). Są to funkcje dość trywialne do napisania, lecz mimo tego są czasochłonne i zaciemniałyby idee algorytmów swoją objętością.

Oba te algorytmy są świetnie opisane w odpowiadającym im pracach naukowych, o których informacje zamieściłem w sekcji *źródła*. Niemniej opiszę w kolejnej sekcji swój autorski kod i dalej pokażę kilka przykładowych wyników jego działania. Przykłady wyników obu programów umieściłem w jednym rozdziale, żeby łatwiej było porównać je ze sobą. Wszystkie pliki z kodem źródłowym można znaleźć na końcu sprawozdania lub pod tym adresem[1].

Implementacja to dwie funkcje w plikach *.m*, przyjmujące na wejściu obraz zbinaryzowany (macierz/tablicę dwuwymiarową), gdzie 0 (czerń) to kolor obiektów, a 1 to kolor tła. Zwracają nowy obraz, bez modyfikacji wejściowego, dlatego nie trzeba martwić się o utratę danych. Do własnych testów, jeśli taka wola czytelnika, wystarczy załadować obraz np. za pomocą *imread*, a po wykonaniu na nim funkcji *KMM* lub *K3M* wyświetlić za pomocą *imshow*.

2. KMM

2.1. Algorytm i implementacja

Załączona implementacja pozwala wyznaczyć szkielet lub kontur wprowadzonego obrazu, zależnie od przekazanego do funkcji argumentu (opis para-

[1] https://github.com/caravard/aipob-thinning

metrów znajduje się w pliku funkcji, można też o nich przeczytać za pomocą *help kmm*).

Poniżej przedstawiony kod jest opisany komentarzami w pliku, więc nie będę opisywał tutaj każdego fragmentu kodu osobno. Przedstawię jedynie algorytm, według którego jest napisany - czyli przedstawię moją interpretację opisu w źródłach. Sam algorytm wykorzystuje strategię "palenia trawy", to znaczy iteracyjnie usuwa kontur obiektu do momentu uzyskania szkieletu.

Program przyjmuje na wejściu obraz zbinaryzowany, gdzie 0 (czerń) to kolor obiektu, 1 (biel) to kolor tła. Następnie odwraca kolory tego obrazu, by tło miało wartość 0, a obiekt 1. Potem wchodzi w główną pętlę algorytmu, stanowiącą jego ciało:

1. Zaznacz kontur obiektu za pomocą *2*
2. Spośród pikseli konturu zaznacz na podstawie wag "łokcieźa pomocą *3*
3. Zaznacz piksele konturu, które mają 2, 3 lub 4 przylegających do siebie sąsiadów za pomocą *4*
4. Usuń wszystkie *4*
5. Dla każdego piksela, który jest oznaczony jako *2*:
   (a) Oblicz jego wagę
   (b) Jeśli ta waga jest w stablicowana jako do usunięcia, ustaw ten piksel jako 0, wpp. jako 1
6. Dla każdego piksela, który jest oznaczony jako *3*:
   (a) Oblicz jego wagę
   (b) Jeśli ta waga jest w stablicowana jako waga piksela do usunięcia, ustaw ten piksel jako 0, wpp. jako 1
7. Jeśli nastąpiła jakaś zmiana, wykonaj pętlę jeszcze raz. Wpp. zakończ działanie

Ostatecznie otrzymany obraz znów odwracamy, by był w tej samej palecie barw, co przed uruchomieniem programu.

Oczywiście rodzi się kilka pytań, to znaczy jak zaznaczyć kontur, o co chodzi z wagami oraz o co chodzi z tablicą wag pikseli do usunięcia.

Waga piksela to nic innego jak liczba binarna, oznaczająca które z pikseli w 8-sąsiedztwie rozważanego pisela obiektu również należy do obiektu. Skoro

takich pikseli może być maksymalnie 8, to będzie to wartość z przedziału *{0, ..., 255}*. W mojej implementacji oblicza się ją za pomocą splotu obrazu z poniższą maską oraz wyzerowanie przypadkowo obliczonych wag dla pikseli tła.

$$\begin{bmatrix} 128 & 1 & 2 \\ 64 & 0 & 4 \\ 32 & 16 & 8 \end{bmatrix} = \begin{bmatrix} 1000\,0000 & 0000\,0001 & 0000\,0010 \\ 0100\,0000 & 0000\,0000 & 0000\,0100 \\ 0010\,0000 & 0001\,0000 & 0000\,1000 \end{bmatrix}$$

Jak widać, kontur będą stanowić wszystkie te piksele, dla których obliczona waga jest różna od *0* (tło) i od *255* (piksel obiektu nie sąsiadujący z tłem). Przyjmujemy, że jednopikselowe elementy obiektu, tzn. takie piksele, które należą do obiektu i nie posiadają żadnych sąsiadów należących do obiektów, nie istnieją (bo algorytm oczekuje 8-spójnych obiektów), a jeśli nawet by istniały, to nie dałoby się ich ścienić, bo są maksymalnie zredukowane.

Stąd powinno być jasne, że stablicowane są pewne konfiguracje ułożeń sąsiadów, które oznaczają, że usunięcie danego piksela jest bezpieczne dla struktury obrazu i przyczyni się ku stworzeniu szkieletu. Autorzy algorytmu twierdzą, że te konfiguracje zostały wyznaczone empirycznie dla uzyskania efektów najbardziej pożądanych.

3. K3M

3.1. Algorytm i implementacja

Implementacja tego algorytmu nie pozwala wyznaczyć samego konturu obrazu, gdyż robi to już implementacja KMM. Poza tym korzystanie z funkcji K3M wygląda tak samo. Zanim czytelnik przeczyta tę część sprawozdania, należy zapoznać się z rozdziałem dotyczącym KMM, gdyż duża część metod tam opisanych jest ponownie wykorzystana w K3M.

K3M jest ulepszonym i bardziej usystematyzowanym algorytmem KMM. W jego skład wchodzi siedem logicznie wyodrębnionych faz. To tak samo jak KMM algorytm iteracyjny typu "palenia trawy".

Wyżej wspomniane fazy ustalają, które piksele obiektu należy zamienić na piksele tła. Pierwszych sześć faz jest powtarzanych iteracyjnie do momentu, gdy w ciągu jednego powtórzenia nie nastąpi żadna zmiana w obrazie. Sposób, w jaki usuwane są piksele, może pozostawić w pewnych miejscach drobne dwupikselowe zgrubienia, dlatego na zakończenie wykonania algorytmu, już poza pętlą, wykonuje się operacje ostatniej fazy, by pozostawić szkielet o grubości jednego piksela.

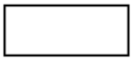
Poszczególne fazy opisane są następująco:
Faza 0: Oznaczanie konturu - w taki sam sposób jak w KMM oznaczony zostaje kontur (w implementacji jako *2*).
Faza 1: Usunięcie pikseli konturu z 3 przylegającymi do siebie sąsiadami.
Faza 2: Usunięcie pikseli konturu z 3 lub 4 przylegającymi do siebie sąsiadami.
Faza 3: Usunięcie pikseli konturu z 3, 4 lub 5 przylegającymi do siebie sąsiadami.
Faza 4: Usunięcie pikseli konturu z 3, 4, 5 lub 6 przylegającymi do siebie sąsiadami.
Faza 5: Usunięcie pikseli konturu z 3, 4, 5, 6 lub 7 przylegającymi do siebie sąsiadami.
Faza 6: Usunięcie zgrubień na szkielecie za pomocą dodatkowej tablicy wag. Wszystkie wagi pikseli o powyżej podanych konfiguracjach sąsiadów są stablicowane ze względów optymalizacyjnych.

4. Przykłady

Chciałbym przedstawić wyniki dla kilku rodzajów obrazów: prymitywów: prostokąta, koła, pierścienia; plamy z wypustkami, małego obrazka oraz pisma maszynowego i odręcznego.

Rysunek 1: Prostokąt

Rysunek 2: Obraz wejściowy

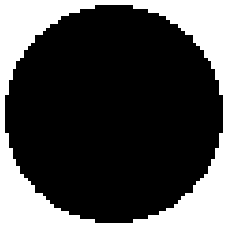


Rysunek 3: KMM - kontur

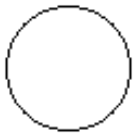
Rysunek 4: KMM

Rysunek 5: K3M

Rysunek 6: Koło



Rysunek 7: Obraz wejściowy



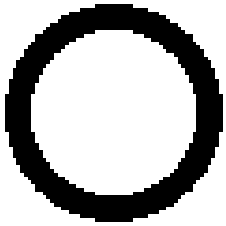
Rysunek 8: KMM - kontur

Rysunek 9: KMM

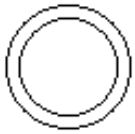
Rysunek 10: K3M

Jak widać na tych dwóch przykładach, dla prostokąta czy koła bardzo trudno zdefiniować szkielet w ten sposób. W końcu nie wiadomo, czy powinien to być po prostu punkt, czy ten szkielet powinien rozpinać figurę, czy w wypadku prostokąta może odcinek równoległy do dłuższego boku. Stąd biorą się dziwne kształty wynikowych szkieletów - w wypadku koła dla K3M mieszanka próby rozpięcia figury ze zbiegnięciem się do punktu, dla prostokąta coś między rozpięciem figury a odcinkiem. Jak się potem okaże, K3M ma tendencję do zachowywania linii pod kątem 135°.

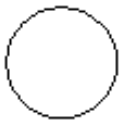
Rysunek 11: Pierścień



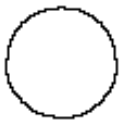
Rysunek 12: Obraz wejściowy



Rysunek 13: KMM - kontur



Rysunek 14: KMM



Rysunek 15: K3M

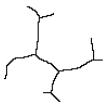
Pierścień jako figura zamknięta, w porównaniu do koła czy prostokąta, które mogą być po prostu fragmentami lub punktami "grubegoódcinka czy krzywej, zachowuje się dobrze pod wpływem tych programów. Nie jest to raczej zbyt pasjonujący przypadek, choć warto zauważyć, że K3M zostawia grubszy szkielet niż KMM - kwestia zachowania kątów prostych.

Rysunek 16: Plama

Rysunek 17: Obraz wejściowy

Rysunek 18: KMM - kontur

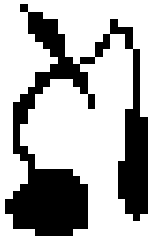
Rysunek 19: KMM



Rysunek 20: K3M

Ten przykład dobrze ilustruje jak K3M zdecydowanie bardziej stara się wyznaczyć szkielet, który lepiej rozpina kształt niż KMM. Z drugiej strony nie musi to być działanie pożądane - czy potrzebujemy tych dodatkowych gałązek w szkielecie plamy? To pewnie zależy od przypadku.

Rysunek 21: Obrazek



Rysunek 22: Obraz wejściowy

Rysunek 23: KMM - kontur

Rysunek 24: KMM

Rysunek 25: K3M

To nieskomplikowany obrazek z kilkoma interesującymi punktami. Posiada większą plamę czerni, jakiś dłuższy kształt, jakiś pochyły kształt i kilka połączeń. Dobrze tu widać, że K3M zostawia bardziej kanciasty szkielet. Ponadto znów widać, że w przypadku rozległego, okrągłego kształtu algorytmy nie są zgodne co do zachowania.

Rysunek 26: Tekst maszynowy

Rysunek 27: Obraz wejściowy

Rysunek 28: KMM - kontur

Rysunek 29: KMM

Rysunek 30: K3M

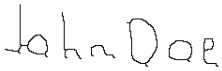
Pismo maszynowe nie jest szczególnie fascynujące przy ścienianiu, ale widać, że efekty są pożądane.

Rysunek 31: Pismo odręczne

Rysunek 32: Obraz wejściowy

Rysunek 33: KMM - kontur

Rysunek 34: KMM



Rysunek 35: K3M

Ostatni już przykład, pismo odręczne, jest chyba najbardziej interesujący i jedyny praktyczny. Pokazuje się tu to, o czym była mowa wyżej - K3M pozostawia gałązki pod kątem 135°. Ciekawie też wypadło połączenie liter *J* i *O*.

5. Źródła

1. K. Saeed, M. Rybnik, M. Tabedzki,
"Implementation and Advanced Results on the Non-Interrupted Skeletonization Algorithm",
– `http://aragorn.pb.bialystok.pl/~zspinfo/arts/2001%20CAIP.pdf`

2. K. Saeed, M. Tabędzki, M. Rybnik, M. Adamski,
"K3M: A UNIVERSAL ALGORITHM FOR IMAGE SKELETONIZATION AND A REVIEW OF THINNING TECHNIQUES",
Int. J. Appl. Math. Comput. Sci., 2010, Vol. 20, No. 2, 317–335
DOI: 10.2478/v10006-010-0024-4
– `http://matwbn.icm.edu.pl/ksiazki/amc/amc20/amc2029.pdf`

6. Kod źródłowy

6.1. KMM

Listing 1: Kod źródłowy KMM (Octave/Matlab)

```
1  function outputImage = kmm(image, aim)
2  % usage: outputImage = kmm(image, aim)
3  %
4  % This function performs thinning on given image (1 is
       background, 0 is foreground)
5  % using KMM algorithm and returns the result
6  %
7  % aim is an optional parameter which can be either
8  %   'skeleton' or 's' for short, which yields 1-pixel wide
       skeleton or
9  %   'contour' or 'c' for short, which yields the contour of the
       shape
10
11 if ~(ismember(image, [0 1]) && 1)
12   disp("Error: image must be binary");
13   return;
14 endif
15
16 if nargin < 2
17   aim = 's';
18 elseif nargin > 2
19   disp("Error: function takes 2 arguments");
20   return;
21 else
22   if aim == 'skeleton'
23     aim = 's';
24   elseif aim == 'contour'
25     aim = 'c';
26   elseif ~any(aim == ['s' 'c'])
27     disp("Error: function argument 2 is invalid");
28     return;
29   endif
30 endif
31
32 % padarray
33 pkg load image;
34
35 % set background as 0's, image as 1's
36 markImage = double(bitxor(image, 1));
37
38 deletionArray = [3      5      7      12     13     14     15
         20   ...
```

```
39                    21      22      23      28      29      30      31
                               48      ...
40                    52      53      54      55      56      60      61
                               62      ...
41                    63      65      67      69      71      77      79
                               80      ...
42                    81      83      84      85      86      87      88
                               89      ...
43                    91      92      93      94      95      97      99
                               101     ...
44                    103     109     111     112     113     115     116
                              117     ...
45                    118     119     120     121     123     124     125
                              126     ...
46                    127     131     133     135     141     143     149
                              151     ...
47                    157     159     181     183     189     191     192
                              193     ...
48                    195     197     199     205     207     208     209
                              211     ...
49                    212     213     214     215     216     217     219
                              220     ...
50                    221     222     223     224     225     227     229
                              231     ...
51                    237     239     240     241     243     244     245
                              246     ...
52                    247     248     249     251     252     253     254
                              255];
53
54  tagFourArray = bitxor([3    6    12   24   48   96   192 129  ...
55                     7    14   28   56   112 224 193 131  ...
56                     15   30   60   120 240 225 195 135],255);
57
58  shift = 256;
59  mask = [128 1 2; ...
60          64 0 4; ...
61          32 16 8];
62
63  change = 1;
64
65  while change
66    change = 0;
67    outputImage = markImage > 0;
68    workingImage = markImage;
69    workingImage = padarray(workingImage, [1 1], 0);
70    workingImage = workingImage == 0;
71    workingImage = conv2(workingImage, mask, 'valid');
72
73
```

```
74
75    % set 2's on mark image -- contour
76
77    markImage = markImage + (workingImage > 0 & markImage > 0);
78
79    % set 3's on tag image -- elbow points
80    % this means that it has bits corresponding to any of numbers:
          2, 8, 32, 128 on and to all of: 1, 4, 16, 64 off.
81
82    on = sum([2 8 32 128]);
83    off = sum([1 4 16 64]);
84    markImage = markImage + ((markImage == 2) & bitand(
          workingImage,on) & ~bitand(workingImage,off));
85
86    if aim == 'c'
87      outputImage = ~(markImage == 2);
88      return;
89    endif
90
91    % set 4's on tag image
92
93    mask1 = (markImage > 1) & ismember(workingImage, tagFourArray)
          ;
94    markImage = markImage .* ~mask1 + 4 * mask1;
95
96    mask1 = (markImage == 4);
97    markImage = markImage .* ~mask1;
98
99    workingImage = zeros(size(markImage));
100   w = size(markImage, 2);
101   h = size(markImage, 1);
102   markImage = padarray(markImage, [1 1], 0);
103   for n=2:3
104     for i=1:h
105       for j=1:w
106         if markImage(i+1,j+1) != n
107           continue;
108         endif
109
110         workingImage(i,j) = sum(sum((markImage(i:i+2,j:j+2) > 0)
                .* mask));
111
112         if ismember(workingImage(i,j), deletionArray)
113           markImage(i+1,j+1) = 0;
114         else
115           markImage(i+1,j+1) = 1;
116         endif
117
118       end
```

```
119       end
120     end
121     markImage = markImage(2:end-1,2:end-1);
122
123     change = ~all(all(outputImage == (markImage > 0)));
124     %imshow(outputImage);
125     end
126
127   outputImage = ~outputImage;
128   end
```

6.2. K3M

Listing 2: Kod źródłowy K3M (Octave/Matlab)

```
1   function outputImage = k3m(image)
2   % usage: outputImage = k3m(image)
3   %
4   % This function performs K3M thinning algorithm on given binary
        image (1 is background, 0 is foreground)
5
6
7     % initial preparations
8     pkg load image;
9
10    neighbourMask = [128 1   2; ...
11                     64  0   4; ...
12                     32  16  8];
13
14    phase0Lookup = [3,  6,  7,  12,  14,  15,  24,  28,  30,  31,
          48, 56, 60, ...
15                    62, 63, 96, 112, 120, 124, 126, 127, 129, 131,
                          135, ...
16                    143, 159, 191, 192, 193, 195, 199, 207, 223,
                          224, ...
17                    225, 227, 231, 239, 240, 241, 243, 247, 248,
                          249, ...
18                    251, 252, 253, 254];
19
20    phase1Lookup = [7, 14, 28, 56, 112, 131, 193, 224];
21
22    phase2Lookup = [7, 14, 15, 28, 30, 56, 60, 112, 120, 131, 135,
          ...
23                    193, 195, 224, 225, 240];
24
25    phase3Lookup = [7, 14, 15, 28, 30, 31, 56, 60, 62, 112, 120,
          ...
26                    124, 131, 135, 143, 193, 195, 199, 224, 225,
                          227, ...
```

```
27                    240, 241, 248];
28
29    phase4Lookup = [7, 14, 15, 28, 30, 31, 56, 60, 62, 63, 112,
          120, ...
30                    124, 126, 131, 135, 143, 159, 193, 195, 199,
                          207, ...
31                    224, 225, 227, 231, 240, 241, 243, 248, 249,
                          252];
32
33    phase5Lookup = [7, 14, 15, 28, 30, 31, 56, 60, 62, 63, 112,
          120, ...
34                    124, 126, 131, 135, 143, 159, 191, 193, 195,
                          199, ...
35                    207, 224, 225, 227, 231, 239, 240, 241, 243,
                          248, ...
36                    249, 251, 252, 254];
37
38    phase1pixLookup = [3, 6, 7, 12, 14, 15, 24, 28, 30, 31, 48,
          56, ...
39                       60, 62, 63, 96, 112, 120, 124, 126, 127,
                             129, 131, ...
40                       135, 143, 159, 191, 192, 193, 195, 199,
                             207, 223, ...
41                       224, 225, 227, 231, 239, 240, 241, 243,
                             247, 248, ...
42                       249, 251, 252, 253, 254];
43
44
45    % bitmap for manipulation
46    workingImage = padarray(bitxor(image, 1), [1 1], 0);
47    % bitmap storing pixel neighbour weights
48    weightImage = zeros(size(image));
49
50    [height width] = size(image);
51
52    % actual algorithm body
53
54    %--for performance purposes only
55    iterationCounter = 0;
56
57    change = 1;
58    while change
59      change = 0;
60
61      %--for performance purposes only
62      iterationCounter = iterationCounter + 1;
63
64      % phase 0 - marking borders
65
```

```
66      % calculate neighbour flags
67      % Convolution is useful in this phase, but later changes in
            the image structure will be made
68      % that will require to recalculate weights before every
            other pixel
69      weightImage = conv2(workingImage, neighbourMask, 'valid') .*
            (workingImage(2:end-1,2:end-1) > 0);
70      % mark borders with 2's
71      workingImage = workingImage + padarray(ismember(weightImage,
            phase0Lookup), [1 1], 0);
72
73      % phase 1 - deleting borders with 3 linked neighbours
74
75      for i=1:height
76        for j=1:width
77
78          if workingImage(i+1,j+1) != 2
79            continue;
80          endif
81
82          weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:
                j+2) > 0)));
83          if ismember(weight, phase1Lookup)
84            workingImage(i+1,j+1) = 0;
85            change = 1;
86          endif
87
88        end
89      end
90
91      % phase 2 - deleting borders with 3 or 4 linked neighbours
92
93      for i=1:height
94        for j=1:width
95
96          if workingImage(i+1,j+1) != 2
97            continue;
98          endif
99
100         weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:
                j+2) > 0)));
101         if ismember(weight, phase2Lookup)
102           workingImage(i+1,j+1) = 0;
103           change = 1;
104         endif
105
106       end
107     end
108
```

```
109       % phase 3 - deleting borders with 3, 4 or 5 linked
              neighbours
110
111       for i=1:height
112         for j=1:width
113
114           if workingImage(i+1,j+1) != 2
115             continue;
116           endif
117
118           weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:
                  j+2) > 0)));
119           if ismember(weight, phase3Lookup)
120             workingImage(i+1,j+1) = 0;
121             change = 1;
122           endif
123
124         end
125       end
126
127       % phase 4 - deleting borders with 3, 4, 5 or 6 linked
              neighbours
128
129       for i=1:height
130         for j=1:width
131
132           if workingImage(i+1,j+1) != 2
133             continue;
134           endif
135
136           weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:
                  j+2) > 0)));
137           if ismember(weight, phase4Lookup)
138             workingImage(i+1,j+1) = 0;
139             change = 1;
140           endif
141
142         end
143       end
144
145       % phase 5 - deleting borders with 3, 4, 5, 6 or 7 linked
              neighbours
146
147       for i=1:height
148         for j=1:width
149
150           if workingImage(i+1,j+1) != 2
151             continue;
152           endif
```

```
153
154           weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:
                  j+2) > 0)));
155           if ismember(weight, phase5Lookup)
156             workingImage(i+1,j+1) = 0;
157             change = 1;
158           endif
159
160         end
161       end
162
163       % phase 6 - unmarking remaining borders
164
165       workingImage = workingImage > 0;
166
167     end
168
169     printf("\nImage thinned after %d iterations.\n\n",
            iterationCounter);
170
171
172     % 1-pixel width phase
173
174     for i=1:height
175       for j=1:width
176
177         if workingImage(i+1,j+1) == 0
178           continue;
179         endif
180
181         weight = sum(sum(neighbourMask .* (workingImage(i:i+2,j:j
                    +2) > 0)));
182         if ismember(weight, phase1pixLookup)
183           %DEBUG/PERFORMANCE printf("%d, coords: %d,%d\n",
                  ismember(weight, phase1pixLookup),i,j);
184           workingImage(i+1,j+1) = 0;
185         endif
186
187       end
188     end
189
190     % assign output
191     outputImage = ~workingImage(2:end-1,2:end-1);
192   end
```