Automatic segmentation and visualization of skin lesions

Jurgen Jans[1] Ralph Kiers[1] Alexandru Telea[1,2]

Technical Report TR-13-06-2013

[1] Institute Johann Bernoulli, University of Groningen, the Netherlands
[2] University of Medicine and Pharmacy "Carol Davila", Bucharest, Romania

1 Introduction

The diagnosis and prognosis of skin tumors is an important problem. A key part of tumor image analysis is the segmentation of the tumor from the surrounding skin. This means detecting which part of the skin belongs to the tumor and which part is just normal healthy skin. A typical example of such a skin tumor image and a possible segmentation of the tumor is shown in **Figure 1**. Recognizing the tumors from such images is hard, since a tumor may

- have weak-contrast, fuzzy, complex-shaped borders
- consist of several types of sub-patterns (e.g. network, globular, unstructured, or salt-and-pepper)
- be partially overlapped by hair structures

For our project, we received a matlab implementation of an image segmentation method[8] and we wanted to make a C++/CUDA implementation of this segmentation method and expand it with a graphical user interface with several visualization methods that would allow the user to compare two or more segmentations of the same skin image with each other. An important reason for us to port the matlab implementation to C++/CUDA is the possible speed up that we would obtain since the matlab code can not be executed in parallel and since skin images usually have a high resolution, the execution of the code can take a long time (especially when multiple images are executed in batch mode). Implementing this method in C++/CUDA instead would allow us to run certain parts of the code in parallel making it also perfectly scalable for the amount of threads that a GPU is able to run.

This report is organized as follows: In section 2 we first start with explaining how we have implemented the segmentation algorithm in C++ and CUDA, followed by a comparison with the matlab implementation of this segmentation algorithm and a benchmark to see how fast our version performs. Section 3 goes into more detail about the implementation of the algorithms used for the segmentation. Section 4 discusses the different visualization techniques that we have implemented in our program in order to compare two or more segmentations with each other. The implementation of every technique is first discussed, followed by a discussion about the results that they produce. Finally section 5 presents the conclusion of our project and describes some future work that could be done in order to improve and expand our work.

2 Segmentation

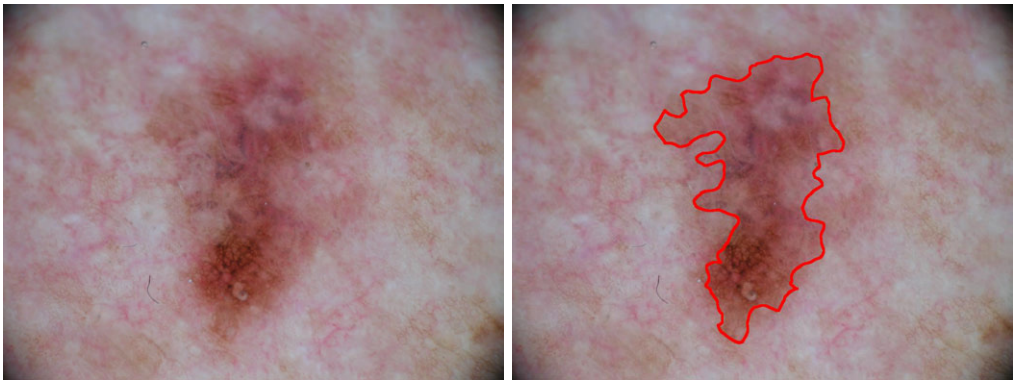
In this section the implementation of the segmentation algorithm based on a matlab implementation of an image segmentation method[8] is detailed. The algorithm is implemented in C++ and CUDA in order to try and make the segmentation as fast as possible.

2.1 Segmentation Algorithm

The segmentation algorithm makes use of various other image processing algorithms. More details about the implementation of the non-trivial algorithms used can be found in **sections 3.1-3.14**.

The segmentation of an image starts off by

This project has been financially supported by the research grant PN-II-RU-TE-2011-3-0249 offered by ANCS/UEFISCDI, Romania.



(a) original tumor skin image. **(b)** Possible segmentation of the tumor.

Figure 1: Skin image

converting it to grayscale. From the grayscale image an image pyramid of depth n is created, n is a value set by the user and $n > 0$. Let $I_i(x, y)$ denote the image at image pyramid level i, using the image $I_n(x, y)$ with n being the maximum depth of the pyramid, an initial segmentation contour is created. In order to create this initial contour Otsu's threshold τ of $I_n(x, y)$ is calculated. Next $B_n(x, y)$ is created by converting $I_n(x, y)$ to binary based on threshold τ, and then inverting the result. The connected components in $B_n(x, y)$ are then labeled and put into a list L together with their area size, with the list L being sorted descending on the areas. If L is empty then no initial contour can be created and an error is returned, reducing the minimum area and/or image pyramid depth might help in this case. Looping through L a suitable connected component is searched for, the connected component has to meet two conditions.

1. The area must be greater than or equal to a minimum area, the minimum area is defined by the user.

2. The connected component must not touch the border of the image.

If both conditions are met then the boundary of the connected component is traced. The traced boundary is then returned as the initial contour. If L contains no suitable connected component then τ is reduced by 5% and the algorithm restarts from the point of converting $I_n(x, y)$ to $B_n(x, y)$. Pseudo-code for extracting the initial contour can be found in **Code listing 1**.

Deformation of the contour is done iteratively by looping over the image pyramid from n to 2. At the start of each iteration i, a Gaussian filter is applied to the $M_i \times N_i$ image $I_i(x, y)$ by using convolution. The Gaussian filter kernel $k_{gaus}(x, y)$ used is a $P \times Q$ Gaussian kernel with standard deviation σ, where

$$P = 1 + \text{round}\left(\frac{M_i}{30}\right)$$
$$Q = 1 + \text{round}\left(\frac{N_i}{30}\right)$$
$$\sigma = \frac{\max(M_i, N_i)}{50}.$$

The filtered image $F_i(x, y)$ is then obtained as $F_i(x, y) = I_i(x, y) \star k_{gaus}(x, y)$, where $\star$ denotes convolution and the padding used for $I_i(x, y)$ is replication padding. After obtaining $F_i(x, y)$ a Sobel edge filter is applied on $F_i(x, y)$, the result of this operation is denoted as $S_{F_i}(x, y)$. Depending on the current iteration i, an edge map $E_i(x, y)$ is created. If $i = n$ then $E_i(x, y) = S_{F_i}(x, y)$, otherwise $E_i(x, y) = S_{F_i}(x, y) * S_{R_i}(x, y)$, where the multiplication is point-wise. Here $R_i(x, y)$ is $I_{i+1}(x, y)$ resized to $M_i \times N_i$ and $S_{R_i}(x, y)$ is the result of the Sobel edge filter applied to $R_i(x, y)$. From E_i the gradient vector flow in both di-

```
Snake init(image, minArea) {
  threshold = image.otsu()
  reduceThresh = 0.05
  while true do
    bw = image.convertToBW(threshold)
    bw.invert()
    /* L contains the label pixels and
       area, sorted desc on the area */
    L = bw.bwlabel()
    if connectedComponents.size() == 0
        then throw Exception

    for c ∈ L do
      if (c.area < minArea) then
        threshold -= reduceThresh
        break
      end

      if touchesBorder(c.pixels) then
        if c == L.end() then
          threshold -= reduceThresh
          break
        else
          continue
        end
      else
        return traceBoundary(c.pixels)
      end
    end
  end
  return NULL
}
```

Code listing 1: Initial contour extraction from $I_n(x,y)$.

```
/* one-based array indices */
for i = n downto 2 do
  img = pyramid[i]
  P = 1 + round(img.width() / 30)
  Q = 1 + round(img.height() / 30)
  σ = max(img.width(),img.height()) / 50

  kgaus = createGaussianKernel(w, h, σ)
  F = img.filter(kgaus,
      PADDING_REPLICATION)
  if i < n then
    R = pyramid[i + 1]
    R.resize(img.width(), img.height())

    SF = sobelEdge(F)
    SR = sobelEdge(R)
    E = SF * SR
    <u, v> = GVF(E, .1, 5)
  else
    E = sobelEdge(F)
    <u, v> = GVF(E, .02, 20)
  end

  /* point-wise multiplication, sqrt */
  mag = sqrt(u*u + v*v)
  /* add 1e-10 to every entry in mag */
  mag += 1e-10
  /* point-wise division */
  u /= mag
  v /= mag

  snake = snakeAdvection(snake, α[i], β
      [i], γ[i], κ[i], u, v, iter[i], r)
  snake = upsampleSnake(snake)
end
```

Code listing 2: Snake deformation loop

rections ($u_i(x,y)$ and $v_i(x,y)$) is created, by $\langle u_i(x,y), v_i(x,y)\rangle = \mathrm{GVF}(E_i, \mu, n_{iter})$. The parameters for the GVF also depend on the current iteration i, if $i = n$ then $\mu = 0.02$, $n_{iter} = 20$, otherwise $\mu = 0.1$, $n_{iter} = 5$. The GVF results are then normalized as follows:

$$mag(x,y) = \sqrt{u_i(x,y)^2 + v_i(x,y)^2}$$

$$u_i(x,y) = \frac{u_i(x,y)}{mag(x,y) + 1e^{-10}}$$

$$v_i(x,y) = \frac{v_i(x,y)}{mag(x,y) + 1e^{-10}}$$

where each operation is point-wise. Having calculated $u_i(x,y)$ and $v_i(x,y)$ snake advection can now be performed. The snake advection requires a couple of parameters set by the user,

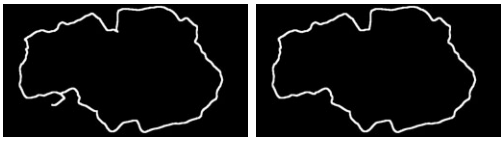
α_i	elasticity;	β_i	rigidity
γ_i	viscosity;	κ_i	external force weight
$iter_i$	number of iterations;	r	resample after r iterations

All the parameters except r are set per level of the image pyramid. Finally after advection the points on the snake are upsampled. That is,

$$\forall p \in Snake : p = 2p.$$

Pseudo-code for the snake deformation loop can be found in **Code listing 2**.

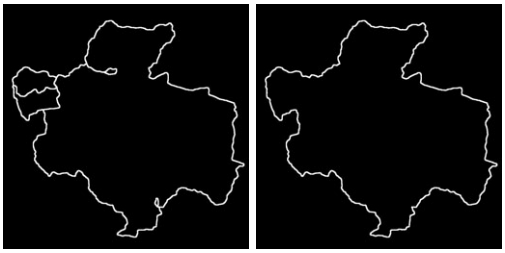
After the deformation is finished, there is still some post-processing to be done. **Figure 2a** shows a peninsula typed artifact on the bottom left of the contour. These type of artifacts form when the snake advection moves points on



(a) Peninsula like artifact on a contour.

(b) Peninsula like artifact removed through curvature-based Laplacian smoothing.

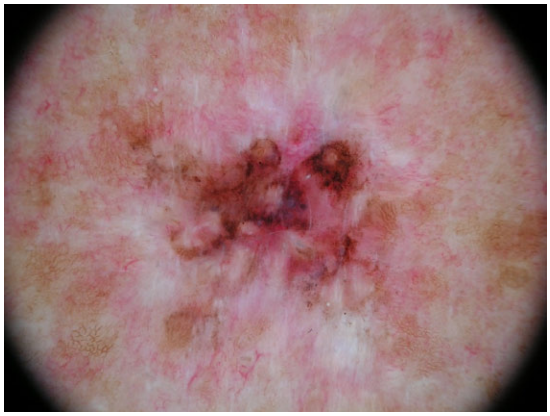
Figure 2: Peninsula like artifact removal.



(a) Self intersections.

(b) Self intersections removed through hole filling and boundary extraction.

Figure 3: Self intersection artifact removal.



(a) Original image.

(b) Matlab segmentation.

(c) C++/CUDA segmentation.

Figure 4: Segmentation comparison.

the snake into an order where they are interleaved, but still form one line. The resampling of snake points does not remove these type of artifacts, because the distance between two consecutive points stays within the threshold. The last point of the peninsula and its consecutive point, is guaranteed to form a sharp corner. This allows curvature-based Laplacian smoothing to remove these points, the difference between pre- and post smoothing results is shown in **Figure 2**. It should be noted that the matlab code where this implementation is based on, uses a different method to get rid of these type of artifacts. The matlab implementation uses `roipoly` which internally uses `poly2mask`[1], followed by only keeping the largest connected component. The second type of artifacts that need to be removed are self intersections. Self intersections are removed through first filling all the holes in the contour, followed by boundary extraction. The difference between pre- and post self intersection cleanup is shown in **Figure 3**.

After all the artifact cleanup is done, the contour is fully connected and one pixel thick. If preferred the contour can be dilated as a final post-processing step, this allows for any contour thickness that is desired.

2.2 Comparison

In this section some segmentations made with the matlab and C++/CUDA implementation are shown side by side for visual comparison. Note that the images of the segmentations are cropped to show the contour only. The segmentations in **Figures (4)-(6)** are all created using the parameters:

$$\alpha = \{.5, .2, .1\} \qquad \beta = \{.5, .2, .1\}$$
$$\gamma = \{.8, .8, .8\} \qquad \kappa = \{.9, .9, .9\}$$
$$iter = \{30, 40, 50\} \qquad r = 5$$
$$n = 4 \qquad A_{\min} = 200$$

System 1		
CPU	**Model**	Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
	Cores	4*
	Clock	3901MHz
GPU	**Model**	2 x GeForce GTX 690**
	CUDA Cores	1536
	GPU Clock rate	1020MHz
	Memory	2 GiB
	Memory Clock rate	3004Mhz
Memory	31GiB	

System 2		
CPU	**Model**	Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00GHz
	Cores	2
	Clock	2001MHz
GPU	**Model**	GeForce 8600M GT
	CUDA Cores	32
	GPU Clock rate	950MHz
	Memory	256 MiB
	Memory Clock rate	400 Mhz
Memory	2GiB	

* 4 physical cores, 8 logical due to hyper-threading
** only one GPU is used for CUDA calculations

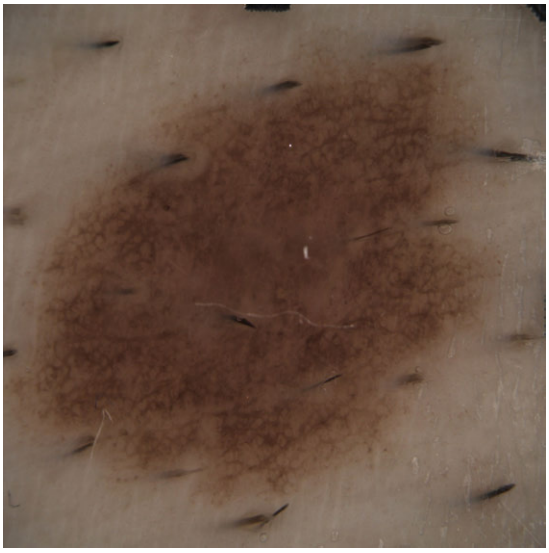
Table 1: System setups

System	Figure	Image Size	Implementation	Time
1	4	1023×767	Matlab	0.536s
1	4	1023×767	C++/CUDA	0.371s
2	4	1023×767	Matlab	2.6s
2	4	1023×767	C++/CUDA	8.4s
1	5	1944×1944	Matlab	56s
1	5	1944×1944	C++/CUDA	1.7s
2	5	1944×1944	Matlab	516s
2	5	1944×1944	C++/CUDA	40s
1	6	1936×2592	Matlab	59s
1	6	1936×2592	C++/CUDA	2.8s
2	6	1936×2592	Matlab	550s
2	6	1936×2592	C++/CUDA	N/A*

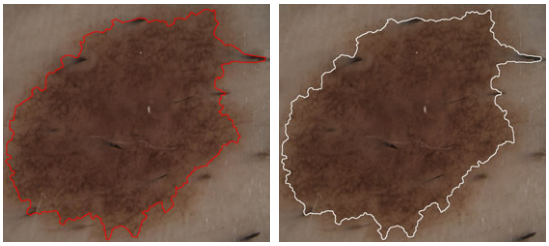
* The GPU runs out of memory.

Table 2: Execution times of the C++/CUDA and matlab implementations on several images. The final times are an average of ten consecutive runs.

where $\{\_, \_, \_\}$ denotes the values for image pyramid levels $2 \dots n$, α is the snake elasticity, β is the snake rigidity, γ is the snake viscosity, κ is the external force weight, *iter* is the number of iterations for snake advection, r is after how many advection iterations resampling happens, n the depth of the image pyramid, and $A_{\min}$ the minimum area for the initial con-



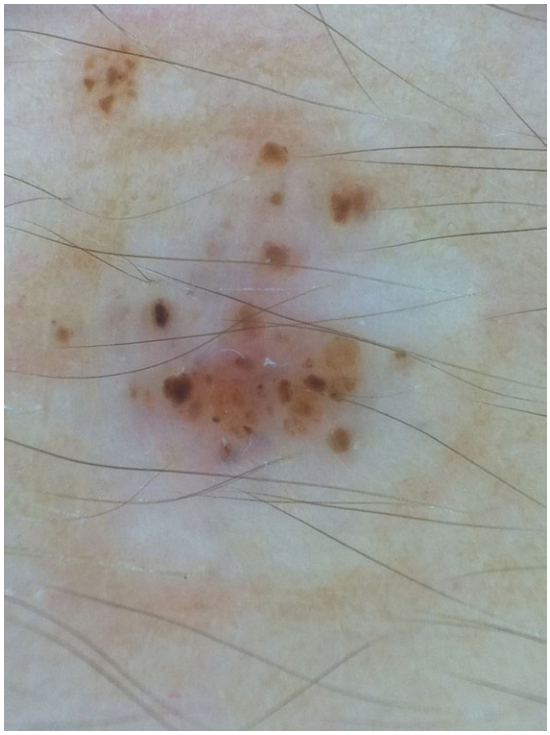
(a) Original image.



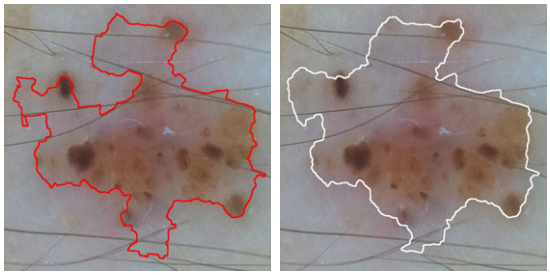
(b) Matlab segmentation.

(c) C++/CUDA segmentation.

Figure 5: Segmentation comparison.



(a) Original image.



(b) Matlab segmentation.

(c) C++/CUDA segmentation.

Figure 6: Segmentation comparison.

tour. The C++/CUDA implementation is also run against the matlab implementation, in order to compare the speed of both implementations. The systems on which the run times were measured can be found in **Table 1**. The execution times shown in **Table 2** are excluding reading the input from disk, and writing the result to disk. In both cases the actual elapsed time of the algorithm is measured. The final timing is the result of the average of ten runs.

Figures (4)-(6) show that given the same parameters, the matlab and C++/CUDA implementations of the segmentation algorithm produce similar visual results. The difference in the images is likely due to matlab using doubles, while in CUDA everything is done with floats, also downscaling in CUDA does not apply a low-pass filter to reduced the effect of moiré patterns, while matlab does. **Table 2** shows that the execution time is in favor of the C++/CUDA implementation, if there is a decent NVIDIA GPU present. For the smallest image(1023×767) in the table, the speedup is only $\frac{536}{371} = 1.44 = 44\%$ for `System 1` and there is even a slowdown of $1 - \frac{2615}{8380} = 0.69 = 69\%$ for `System 2`. However, on the bigger im-

```
float bicubic(float* p, float ix, float
    iy) {
  float r0 = cubic(p[0] , p[1] , p[2] ,
      p[3] , ix);
  float r1 = cubic(p[4] , p[5] , p[6] ,
      p[7] , ix);
  float r2 = cubic(p[8] , p[9] , p[10],
      p[11], ix);
  float r3 = cubic(p[12], p[13], p[14],
      p[15], ix);
  return cubic(r0, r1, r2, r3, iy);
}
```

Code listing 3: Bicubic-spline interpolation with, ix and iy as the vertical and horizontal interpolation points.

ages the C++/CUDA implementation is significantly faster on both systems. `System 2` also runs out of GPU memory during the segmentation of **Figure 6a**. Running out of memory is due to whole levels of the image pyramid **3.4 Image Pyramid** being loaded into the GPU memory at once. The CUDA implementation could benefit from performing some operations on blocks of the image, where possible.

3 Algorithms

Next the algorithms implemented in C++ and CUDA will be described.

3.1 Convert to Grayscale

Conversion to grayscale is only implemented for RGB images, conversion is done using CUDA since data for each pixel is independent. The formula used for the conversion is given by:

$$G_{val} = \text{round}\left(0.299R + 0.587G + 0.114B\right)$$

where R is the red value, G is the green value, B is the blue value and G_{val} is the gray value, and R, G, B, G_{val} are integer values in the interval $[0, 255]$.

3.2 Resizing

When rescaling an image the same algorithm as matlab version 7.9.0.529 is used to calculate the sampling points. The sampling points s_x, s_y are given by the following equations:

$$s_x = \frac{x + \frac{1}{2}}{f_x} - \frac{1}{2}$$
$$s_y = \frac{y + \frac{1}{2}}{f_y} - \frac{1}{2}$$

where x, y are the indices in the rescaled image and f_x, f_y are the scaling factors in the x, y direction. Note that the equation given here is not exactly the same as the one found in matlab, due to matlab using one-based indexing while C++ uses zero-based indexing.

Having found the sampling points, the next step is to interpolate the values at the sampling points, as it is unlikely that they will correspond to actual discrete pixel coordinates. The interpolation method used for obtaining the values at the sampling points is bicubic-spline interpolation. Bicubic-spline interpolation is implemented using a series of five cubic-spline interpolations in 1-D. The cubic-spline interpolation implementation itself is deduced from eq. (1).

$$y = ax^3 + bx^2 + cx + d \tag{1}$$

Let P_n denote a point n and let p_n denote the value at point n, then given two points P_1 and P_2 in between which to interpolate, $y(0)$ and $y(1)$ can be calculated by:

$$p_1 = y(0) = d \tag{2}$$
$$p_2 = y(1) = a + b + c + d \tag{3}$$

this results into two equations with four unknowns which means there is no unique solution, thus two more equations are needed. The next two equations are deduced from $\frac{d}{dx}y(0)$ and $\frac{d}{dx}y(1)$. The values corresponding to the derivatives are found by taking the discrete derivatives in P_1 and P_2. This gives the following equations:

$$\frac{1}{2}p_2 - \frac{1}{2}p_0 = \frac{d}{dx}y(0) = c \tag{4}$$
$$\frac{1}{2}p_3 - \frac{1}{2}p_1 = \frac{d}{dx}y(1) = 3a + 2b + c. \tag{5}$$

Using eqs. (2)–(5), a, b, c and d can now be solved in terms of $p_0 \dots p_3$ through Gaussian elimination. Performing said Gaussian elimination yields:

$$a = -\frac{1}{2}p_0 + \frac{3}{2}p_1 - \frac{3}{2}p_2 + \frac{1}{2}p_3 \quad (6)$$

$$b = p_0 - \frac{5}{2}p_1 + 2p_2 - \frac{1}{2}p_3 \quad (7)$$

$$c = -\frac{1}{2}p_0 + \frac{1}{2}p_2 \quad (8)$$

$$d = p_1 \quad (9)$$

substituting the exponents in eqs. (6)–(9) into eq. (1) defines to polynomial for $P_0 \dots P_3$. Interpolating between P_1 and P_2 is now simply done by solving eq. (1) for x, where $0 \leq x \leq 1$. Note that $x = 0$ corresponds to p_1 and $x = 1$ corresponds to p_2. Pseudo-code of the CUDA implementation for bicubic-spline interpolation using the 1-D cubic-spline interpolation can be found in **Code listing 3**.

3.3 Convolution

Convolution is implemented to perform spatial filtering, in favor of correlation. This is done using the CUDA `cufft` library, to allow convolution to be a fast point-wise complex multiplication in the frequency domain. This does however mean that all the non-symmetrical filter kernels are rotated 180° in the code.

The padding types implemented for convolution aside from no padding, are zero- and replication padding. Both zero- and replication padding pad the input to size $M+P-1 \times N+Q-1$, where M is the image width, N is the image height, P is the kernel width and Q is the kernel height. **Figure 7a** and **Figure 7b** show an example of zero padding and replication padding, as implemented in CUDA. **Figure 7c** shows how **Figure 7b** is indeed replicating the nearest border, due to the circularity of the FFT.

Filter kernels need to start at the center of the signal. Therefore the padded kernel is transformed so that the middle of the kernel starts at the start of the image signal. **Figure 8** shows an example of the transformation, as implemented in CUDA.

```
if img(x,y) <= τ then
  bw(x,y) = 0
else
  bw(x,y) = 1
end
```

Code listing 4: Converting grayscale to binary using a threshold τ.

3.4 Image Pyramid

The image pyramid is created by using the input image of size $M \times N$ as base of the pyramid. Let the base level be level 1 and let $I_n(x, y)$ denote the image at pyramid level n with dimensions $P_n \times Q_n$, for $n > 0$. then P_n and Q_n are calculated by $P_n = \left\lceil \frac{P_{n+1}}{2} \right\rceil$, $Q_n = \left\lceil \frac{Q_{n+1}}{2} \right\rceil$ and $I_n(x, y)$ is calculated by $I_n(x, y) = I_{n+1}(x, y) \star k(x, y)$ scaled down to $P_n \times Q_n$ using the method described in **3.2 Resizing**. Here $\star$ denotes the correlation implemented as in **3.3 Convolution** and $k(x, y)$ is the Gaussian kernel given by

$$k(x, y) = \frac{g(x - \frac{a-1}{2}, y - \frac{b-1}{2})}{\sum_{s=0}^{a-1} \sum_{t=0}^{b-1} g(s - \frac{a-1}{2}, t - \frac{b-1}{2})}$$

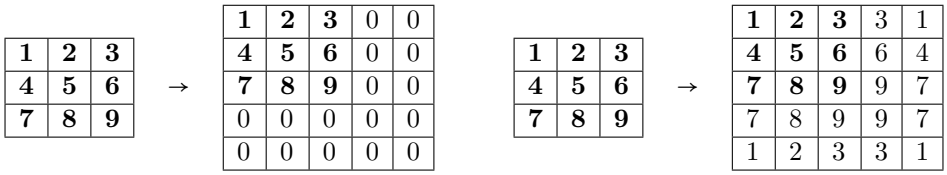
with a is the width of k, b is the height of k, $0 \leq x < a$, $0 \leq y < b$ and x, y are integers. $g(x, y)$ is the standard Gaussian function given by

$$g(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where σ denotes the standard deviation. The pyramid implementation uses a 7×7 Gaussian kernel (implemented in C++) with $\sigma = 3$.

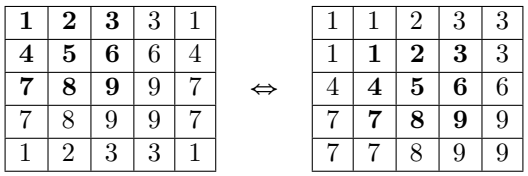
3.5 Grayscale to Binary

Conversion from a grayscale image to binary is done by first calculating a threshold, the threshold is then used to determine which pixels belong to the foreground and background. Calculating the threshold is done using Otsu's method[7], the entire method is implemented in CUDA. Once the threshold is calculated the binary image is simply obtained by a CUDA



(a) Padding a 3×3 image to a 5×5 image using zero padding.

(b) Padding a 3×3 image to a 5×5 image using replication padding.



(c) Equivalence due to FFT circularity

Figure 7: Padding examples.

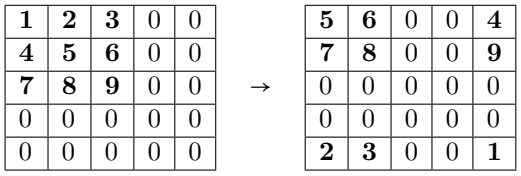


Figure 8: Centering a 3×3 kernel; zero padded to 5×5; on $(0, 0)$.

kernel implementing the thresholding shown in **Code listing 4**.

3.6 Connected Component Labeling

Connected component labeling is implemented in C++. In order to label the connected components in a binary image a two-pass algorithm is used. The algorithm requires an equivalence list which keeps track of which labels belong to the same connected component. during the first-pass the pixels are scanned row based, once a foreground pixel is found a label is assigned to it. The assigned label is a new unused label if the 8-connected neighbors do not have a label, otherwise it is the smallest label found in the neighbors. If a connected component gets labeled, but still has neighbors with a higher label, then an entry is added to the equivalence list stating the two labels belong two the same connected component. **Figure 9** shows the labeling process during the first-pass. The created equivalence list after the first pass consists of the pair $\langle 2, 1 \rangle$. The second pass solves equivalences and produces a set of labels which are not equivalent. Leading to the final result in **Figure 10**.

3.7 Boundary Tracing

Boundary tracing is implemented as Moore boundary tracking[6]. The whole algorithm is implemented in C++, because it is a sequential algorithm and thus using CUDA will not provide a lot of benefit.

3.8 Snake resampling

Snake resampling takes a connected snake, with points in clockwise or counterclockwise order and resamples it in two-passes. The algorithm is implemented in C++, because each point is checked and changed sequentially.

During the first-pass all the points on the snake that are too close to each other are re-

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0	0

(a) Binary input image.

0	0	0	0	0	0	0	0	0
0	0	0	**1**	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(b) First foreground pixel found gets labeled 1.

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	**2**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(c) First pixel in row five gets labeled 2, since the 8-connected neighbors do not have a label.

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	2	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	3	3	0
0	0	0	0	0	0	3	3	0
0	0	0	0	0	0	0	0	0

(d) Labels at the end of the first pass.

Figure 9: First-pass of connected component labeling.

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	2	2	0
0	0	0	0	0	0	2	2	0
0	0	0	0	0	0	0	0	0

Figure 10: Labels at the end of the second pass.

```
for each p_i ∈ snake
  if dist(p_i, p_{i+1}) >= d_min
    newSnake.add(p_i)
  endif
end
snake = newSnake
```

Code listing 5: First-pass of the snake resampling algorithm.

moved according to a specified minimum distance $d_{\min}$. The pseudo-code in **Code listing 5** shows how the new snake gets constructed. The distance function in the code is defined as

$$\mathrm{dist}(p_1, p_2) = \|p_1.x - p_2.x\| + \|p_1.y - p_2.y\|, \quad (10)$$

which is the Manhattan distance. A thing to note is that $p_{n+1} = p_1$ where n is the total number of snake points.

In the second-pass new points are added to

```
current = p_1
while current != p_n do
  /* set the next point */
  next = current + 1
  if dist(current, next) > d_max
    /* q is the point interpolated half way between current and next */
    q = interp(current, next, 0.5)
    if next != snake.begin()
      /* current is the new point inserted before next */
      current = snake.insert(next, q)
    else
      /* current is the newly added point */
      current = snake.add(p);
    end
    /* set current to the previous point */
    --current
  else
    /* set current to the next point */
    ++current
  end
end
```

Code listing 6: Second-pass of the snake resampling algorithm

the snake, if the specified maximum distance $d_{\max}$ between two consecutive points p_i and p_{i+1} is exceeded. If the distance exceeds $d_{\max}$ then a new point p_j is created by linear interpolating half way between p_i and p_{i+1}. If after adding p_j the distance is still exceeding $d_{\max}$, then a new point is created again by interpolating between p_i and p_j, this process continues until the distance does not exceed $d_{\max}$. The pseudo-code in **Code listing 6** depicts the process. Again dist is defined as in eq. (10), and $p_{n+1} = p_1$, $p_{1-1} = p_n$.

3.9 Sobel Edge Filter

Sobel edge filtering is implemented using convolution, as shown in **3.3 Convolution**. The image is filtered using a horizontal kernel followed by filtering with a vertical kernel. Giving

$$s_x(x,y) = (f(x,y) \star k_x(x,y))$$
$$s_y(x,y) = (f(x,y) \star k_y(x,y))$$

where $f(x,y)$ is the image, $s_x(x,y)$ is the horizontal sobel edge filter result and $s_y(x,y)$ is the vertical sobel edge filter result. The filter kernels are defined as

$$k_x(x,y) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$
$$k_y(x,y) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

note that both kernels are rotated 180°, because convolution is used instead of correlation. After filtering, the distance is calculated point-wise in CUDA as $d(x,y) = \sqrt{(s_x(x,y))^2 + (s_y(x,y))^2}$. Finally the normalized distance is returned as result, where normalizing is done as follows:

$$\frac{d(x,y)}{\max(d(x,y))}.$$

3.10 Gradient Vector Flow

The gradient vector flow[9] calculation is implemented in CUDA. The algorithm takes three arguments:

1. $f(x,y)$ the result of the sobel edge filter, or the multiplication of sobel edge filter results.

```
for i from 0 to nIter - 1 do
  gvf_x(x,y) = gvf_x(x,y) + μ(gvf_x(x,y) ☆ k_l(x,y)) − mag(x,y) ∗ (gvf_x(x,y) − g_x(x,y))
  gvf_y(x,y) = gvf_y(x,y) + μ(gvf_y(x,y) ☆ k_l(x,y)) − mag(x,y) ∗ (gvf_y(x,y) − g_y(x,y))
end
```

Code listing 7: Iterative calculation of the gradient vector flow. The value of μ is passed to the algorithm and k_l is a discrete laplacian kernel.

2. μ a multiplication factor for the result of a laplacian filter
3. *nIter* number of iterations the algorithm should perform.

At first the normalization of $f(x,y)$ is calculated as

$$\bar{f}(x,y) = \frac{f(x,y) - \min(f(x,y))}{\max(f(x,y)) - \min(f(x,y))}.$$

Then $\bar{f}(x,y)$ is used to calculate the gradient in both horizontal and vertical direction:

$$g_x(x,y) = \bar{f}(x,y) \star k_x$$
$$g_y(x,y) = \bar{f}(x,y) \star k_y$$

where g_x, g_y are the gradient in horizontal and vertical direction. The gradient filter kernels are defined as:

$$k_x(x,y) = \begin{bmatrix} \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix}; \qquad k_y(x,y) = \begin{bmatrix} \frac{1}{2} \\ 0 \\ -\frac{1}{2} \end{bmatrix}$$

note that both kernels are again rotated 180°, because convolution is used instead of correlation. Next the squared magnitude is calculated point-wise as:

$$mag(x,y) = (g_x(x,y))^2 + (g_y(x,y))^2.$$

Initially the gradient vector flow in the horizontal and vertical direction is set to:

$$gvf_x(x,y) = g_x(x,y)$$
$$gvf_y(x,y) = g_y(x,y).$$

After setting the initial GVF values, the final values for $gvf_x(x,y)$ and $gvf_y(x,y)$ are calculated iteratively, as shown in **Code listing 7**. The discrete laplacian kernel used is defined as

$$k_l = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

3.11 Snake Advection

Snake advection tries to minimize the snakes internal energy[5]. The implementation in C++ follows the same equations(eqs. (11)–(12)) as given in [5], with the exception that no inverse matrix is used.

$$x_t = (A + \gamma I)^{-1} (\gamma x_{t-1} - f_x(x_{t-1}, y_{t-1})) \quad (11)$$
$$y_t = (A + \gamma I)^{-1} (\gamma y_{t-1} - f_y(x_{t-1}, y_{t-1})) \quad (12)$$

Instead a system of linear equations(eqs. (13)–(14)) is solved using the `Eigen` library, because it does not require a dense inverse matrix on which matrix multiplication has to be performed, but instead makes use of LDLT Cholesky decomposition. Note that LDLT Cholesky decomposition is possible, because $A + \gamma I$ is as sparse banded matrix.

$$(A + \gamma I) x_t = \gamma x_{t-1} - \kappa f_x(x_{t-1}, y_{t-1}) \quad (13)$$
$$(A + \gamma I) y_t = \gamma y_{t-1} - \kappa f_y(x_{t-1}, y_{t-1}) \quad (14)$$

Equations (13)–(14) also introduce the factor κ as an external force weight.

The implementation consists of two loops. The inner-loop does the actual advection, while the outer-loop handles resampling after a certain number of iterations. The outer-loop starts by calculating the coefficients for the discrete derivatives. The coefficients $c_1 \dots c_5$ are calculated from the user specified values α(elasticity parameter), β(rigidity parameter) and γ(viscosity parameter).

$$c_1 = \beta \qquad c_2 = -\alpha - 4\beta$$
$$c_3 = (2\alpha + 6\beta) + \gamma \qquad c_4 = -\alpha - 4\beta$$
$$c_5 = \beta$$

From $c_1 \dots c_5$ the $N \times N$ matrix A, with N the

```
Snake snakeAdvection(snake, α, β, γ, κ, gvfx, gvfy, nIter, resample) {
  /* calculate discrete derivatives */
  coefficients[5] = {β, −α − 4β, (2α + 6β) + γ, −α − 4β, β}
  i = 0
  while i < nIter do
    /* create snake.size() by snake.size() banded matrix */
    A = createMatrix(snake.size(), coefficients)
    /* set number of iterations after which we resample */
    if resample <= nIter - i then iIter = resample else iIter = nIter - i
    snake = snakeAdvecInner(snake, A, γ, κ, gvfx, gvfy, iIter)
    snake = resampleSnake(snake, 0.5, 2.0)
    i += iIter
  end
  return snake
}
```

Code listing 8: Snake advection outer-loop.

number of points in the snake, is created as

$$A = \begin{bmatrix} c_3 & c_4 & c_5 & 0 & \cdots & 0 & c_1 & c_2 \\ c_2 & c_3 & c_4 & c_5 & 0 & \cdots & 0 & c_1 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ c_5 & 0 & \cdots & 0 & c_1 & c_2 & c_3 & c_4 \\ c_4 & c_5 & 0 & \cdots & 0 & c_1 & c_2 & c_3 \end{bmatrix}.$$

Then the inner-loop is executed for a number of iterations, followed by the resampling of the snake as shown in **3.8 Snake resampling**. The pseudo-code in **Code listing 8** and **Code listing 9** depict the process. Note that in the code, A is a sparse matrix created using the `Eigen` library.

3.12 Curvature-based Laplacian Smoothing

Curvature-based Laplacian smoothing limits the angle two consecutive points can have. The algorithm is sequential, similar to the snake resampling and thus is implemented in C++. Curvature-based Laplacian smoothing works by deleting points if the angle between two consecutive vectors is too big. The vectors $\vec{v_1}$ and $\vec{v_2}$ are chosen as:

$$\vec{v_1} = \begin{bmatrix} p_i.x - p_{i-1}.x \\ p_i.y - p_{i-1}.y \end{bmatrix}$$

$$\vec{v_2} = \begin{bmatrix} p_{i+1}.x - p_i.x \\ p_{i+1}.y - p_i.y \end{bmatrix}$$

where p_i is the current point being inspected, p_{i+1} is the next point, with $p_{n+1} = p_1$ where n is the number of points on the snake. p_{i-1} is the previous point on the snake that was not deleted, if there is no preserved point yet p_{i-1} is simply the previous point on the snake, with $p_{1-1} = p_n$. The chosen vectors $\vec{v_1}$ and $\vec{v_2}$ are normalized giving $\hat{v_1} = \frac{\vec{v_1}}{\|\vec{v_1}\|}$ and $\hat{v_2} = \frac{\vec{v_2}}{\|\vec{v_2}\|}$. Now the dot product $\hat{v_1} \cdot \hat{v_2}$ is equal to $\cos(\theta)$, where θ is the angle between $\vec{v_1}$ and $\vec{v_2}$.

The algorithm preserves the current point if $\hat{v_1} \cdot \hat{v_2} \geq 0.9$, otherwise the current point is discarded. After going through all the points on the snake, the snake is resampled using the method shown in **3.8 Snake resampling**. Resampling is done with $d_{min} = 0.5$ and $d_{max} = 2$. If points were discarded by the algorithm it is run again, until no points are discarded.

3.13 Contour Closing

Contour closing transforms the points on the snake into actual discrete points and draws a line between them, to make sure the contour is closed. Because the order of points matters when drawing the lines the algorithm is sequential, and thus again implemented in C++. The algorithm loops through he snake coordinates. During the loop the snake points p_i and p_{i+1} are first round to integer values, with again $p_{n+1} = p_1$ where n is the total number of snake points. Using Bresenham's line drawing[2] algorithm all the discrete points that make up the line from round(p_i) to round(p_{i+1}) are added to

```
Snake snakeAdvecInner(snake, A, γ, κ, gvfx, gvfy, iter) {
  bx = snake.x
  by = snake.y
  for i = 0 to iter - 1 do
    for j = 0 to snake.size() - 1 do
      /* 2-D linear interpolation of the GVF for the snake points */
      vfx = interp2(gvfx, snake.x[j], snake.y[j])
      vfy = interp2(gvfy, snake.x[j], snake.y[j])
      snake.x[j] = γ * snake.x[j] + κ * vfx
      snake.y[j] = γ * snake.y[j] + κ * vfy
    end
    /* solve Ax = b */
    snake.x = solve(A, snake.x)
    snake.y = solve(A, snake.y)
  end
  return snake
}
```

Code listing 9: Snake advection inner-loop.

a buffer containing all the pixels to be drawn.

3.14 Hole Filling & Boundary Extraction

Hole filling by reconstruction[4] and boundary extraction[4] have a full implementation in CUDA. Hole filling and boundary both make use of dilation, in both cases a 4-connectivity kernel k is used to perform this dilation. Where k is defined as:

$$k = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

4 Visualization

This section describes the implementation of the different visualization methods that our program provides and also discusses the results that they produce. The different visualizations techniques discussed in this section are: Distance calculation (**4.2.1 Distance calculation**) , Feature vector (**4.2.2 Feature vector**), Double feature vector (**4.2.3 Double feature vector**) and the N segmentations visualization technique (**4.2.4 N segmentations**). But before we start with the visualizations techniques, we will first give a brief overview of some basic methods on which all of the implemented techniques are depending on.

4.1 Overview

Before any of the visualization techniques can be used, the program needs to read in a color image of the skin that needs to be analysed and any possible previously made segmentations of this skin image (in grayscale or binary format). The skin image is saved in its original format but the segmentation images first need to be converted into binary images before they can be used (**4.1.1 Convert to binary**). After this the visualization methods can be performed on the images and the results will be visualized using a rainbow colormap (**4.1.2 Rainbow colormap**).

4.1.1 Convert to binary

Conversion to a binary image is only implemented for grayscale images, conversion is done using CUDA, since data for each pixel is independent. The formula used for the conversion is given by:

$$B_{val} = \begin{cases} 1 & \text{if } G_{val} == 255 \\ 0 & \text{else} \end{cases}$$

where G_{val} is the grayvalue with a integer value in the interval $[0, 255]$ and B_{val} is the resulting

binary value which is either 0 or 1. This conversion works for grayscale segmentation images since it is assumed that the pixels of the segmentations are always white thus have a G_{val} of 255 and the remaining pixels can be considered as background pixels.

4.1.2 Rainbow colormap

In order to visualize the results of the different visualization techniques a rainbow colormap (**Figure 11**) is used. This colormap starts with blue and goes through the colors green and yellow to red. The results are assigned to the colormap in such a way that the lowest possible result values are shown in blue and the maximal possible value gets the color red. All the other result values are scaled between these two colors. This makes it easier for the user to compare the differences between different results by just looking at the colors and see where the differences are big and where they are (almost) equal to each other.

Figure 11: Rainbow colormap

4.1.3 Distance transform

All of the implemented visualization methods need to make use of the distance between pixels in one segmentations and pixels in another segmentation. Therefore we first do a distance transformation on every new added segmentation image using the exact Euclidean Distance Transform (EDT) on the GPU using CUDA as described in [3]. This method only works for binary images and constructs a 2D weighted centroidal Voronoi diagram for every segmentation image. This Voronoi diagram contains information that enables us to find the nearest segmentation pixel for every other given pixel in the image.

4.2 Visualization techniques

4.2.1 Distance calculation

The distance calculation method visualizes the distance between one segmentation and a second segmentation by coloring the pixels of the border using the distance between every pixel on that border and its corresponding closest pixel on the other border using the rainbow colormap as discussed in **4.1.2 Rainbow colormap**. This way the pixels which are close to the other segmentation pixels or even overlapping the other segmentations pixels are colored blue while the pixels which have the biggest distance are colored red.

In order to calculate the distance between both segmentation pixels, we make use of the distance transform discussed in **4.1.3 Distance transform**. With the Voronoi diagrams calculated for the segmentation with which we want to compare our given segmentation, we can easily lookup every pixel of the given segmentation in the voronoi diagram of the other segmentation image. When we know the pixel on the other segmentation closest to the given pixel we can calculate the distance between them using euclidean distance.

$$eu\,(p_1, p_2) = \sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2} \tag{15}$$

where p_1 and p_2 are the two pixels for which we want to calculate the distance. Since this can be done for every pixel independently, we have implemented this in CUDA which improves the performance.

Now that we know for every pixel on one segmentation the distance to the nearest pixel on the other segmentation, we can color every pixel in the segmentation using the rainbow color map. This method also provides the option to let the user color both segmentation borders, in this case the above described procedure is repeated for the other segmentation. This method can be used to find the differences between two segmentations of the same skin image. The user can easily see where the biggest differences between the segmentations are and at which places they are more similar by just looking at the colors.

Besides coloring the selected segmentation in order to display to distance, the program also calculates the distance in a quantitative manner $\text{dist}(s_1, s_2) = \left(\frac{\sum_{p_1 \epsilon s_1} eu(p_1, p_1^*)}{|s_1|^2} + \frac{\sum_{p_2 \epsilon s_2} eu(p_2, p_2^*)}{|s_2|^2} \right) / 2$ where s_1 and s_2 are the two segmentations with p_1 and p_2 pixels on both segmentations, p_1^* and p_2^* the nearest pixel of the other segmentation and $|s_1|$ and $|s_2|$ the total number of pixels in respectively segmentation 1 and segmentation 2. The resulting value indicates the similarity of both borders where a value of 0 means that both segmentations are exactly the same.

In **Figure 12a** an example is shown in which the distance for every pixel on the larger segmentation with their nearest pixel on the smaller segmentation is calculated. The coloring of the segmentation shows clearly that the top part of the segmentation is close to the other segmentation since its color is blue but the lower side of the segmentation is far removed from the other segmentation and therefore colored red. The calculated distance between these two segmentations is 0.0110135. **Figure 12b** shows the same case as in the previous image but this time the option to color both segmentations is used. As can be used in the image, the colors of the segmentation already painted in **Figure 12a** haven't changed but this time the distance for every pixel on the second border with respect to their nearest pixel on the first segmentation is also visualized. **Figure 12c** shows an example in which the two segmentations are more similar with each other. This is also clearly shown by the coloring of the segmentation since the biggest part is painted in blue while only the small parts that differ have different colors. The calculated distance in this case is 0.000375414. Compared with the calculated distance in **Figure 12a**, it can be seen that the calculated distance value is lower when the two compared segmentations are more similar.

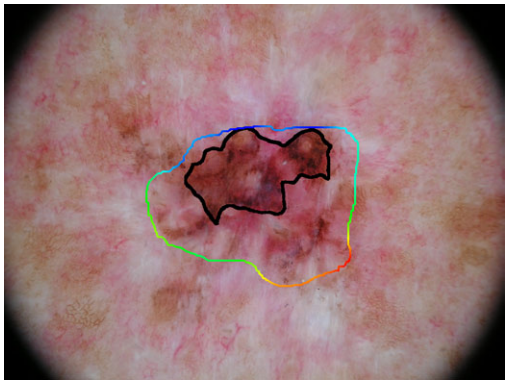
4.2.2 Feature vector

The feature vector is used in combination with the distance calculation and shows for every pixel in one segmentation the nearest pixel in the other segmentation by drawing lines between them. The drawn lines are also colored according to the distance between the two pixels using the rainbow colormap (**4.1.2 Rainbow colormap**). This is especially useful since the Distance calculation method only colors the pixels of a segmentation according to the distance with the nearest pixel in the other segmentation but does not show which pixel on the other segmentation is actually the closest pixel. Since we already have the Voronoi diagrams of the segmentations calculated using the distance transform (**4.1.3 Distance transform**) we already know the coordinates of both the given pixel and the nearest pixel on the other segmentation, therefore we only need to draw a line between them with the proper distance color. This is done by using the Bresenham's line algorithm which can draw a straight line between two points given the coordinates of both points. The pseudo code of this algorithm is shown in **Code listing 10**. We have implemented it in CUDA since we need to draw a line for every pixel in the segmentation to their nearest pixel in the other segmentation in this can be done eassily in parallel once the coordinates of the nearest pixel is known.

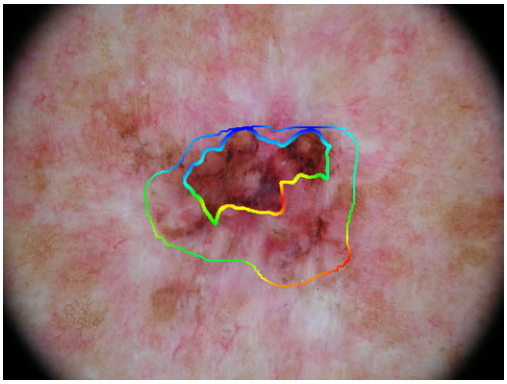
In **Figure 13** we have shown the same segmentations for which we visualized the distance in **Figure 12a** and **Figure 12b** but this time we have drawn the feature vectors for both segmentations. Where the user could only see which parts of the segmentations were similar to / different from each other in the distance calculation method, by using the feature vectors the user can actually also see for every pixel where the nearest pixel on the other segmentation is located.

4.2.3 Double feature vector

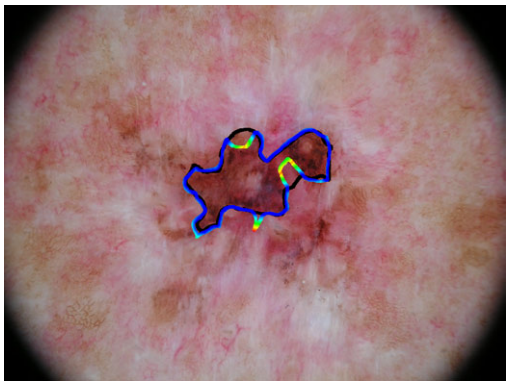
The double feature vector visualizes the same information as the normal feature vector but in this case for both segmentations. For every pixel on one segmentation a line is drawn



(a) Distance between large segmentation and a smaller one.



(b) Distance between both segmentations.



(c) Distance between two similar segmentations.

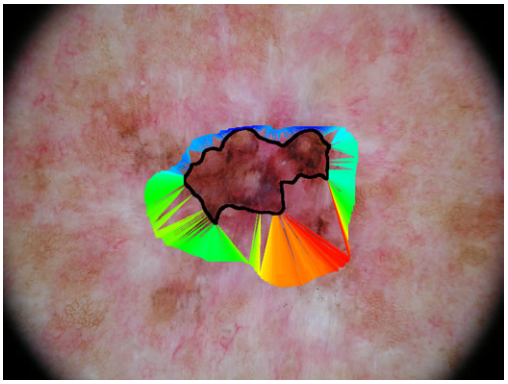
Figure 12: Distance calcualtions

to the nearest pixel on the other segmentation and for every pixel on the other segmentation there is also a line drawn to their nearest pixels on the first segmentation. Again the Bresenham's line algorithm **Code listing 10** is used for drawing the lines between the pixels. The distance however is visualized for both sets together meaning that the maximum distance colored as red (rainbow colormap **4.1.2 Rainbow colormap**) is the maximum distance found in either the first or the second segmentation distances set.

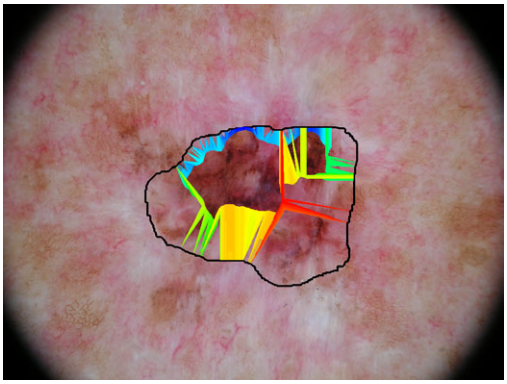
In **Figure 14** we have shown an example of a double feature vector for the segmentations used in **Figure 12a** and **Figure 12b**. Compared to the feature vectors shown in **Figure 13** it can be clearly seen that the double feature vector method draws both the feature vectors from the previous case in the same image.

4.2.4 N segmentations

The N Segmentations functionality can be used to visualize information about a set of segmentations. Every segmentation is compared with the set of the other remaining segmentations in the following way: first an average segmentation is computed by calculating an average point for every set of nearest pixel points of the remaining segmentations. Next the distance between this average segmentation and the given segmentation can be calculated for every pixel in this segmentation using the euclidean distance eq. (15). The pixels will



(a) Feature vector from large segmentation to smaller.



(b) Feature vector from small segmentation to larger.

Figure 13: Feature vectors

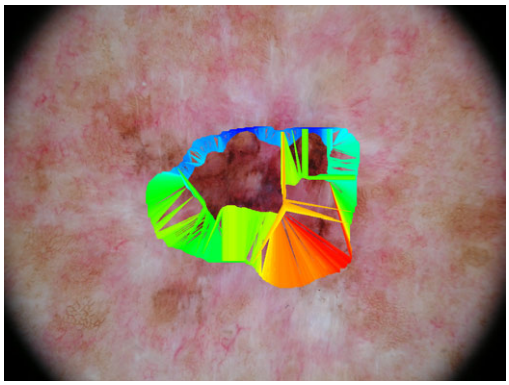


Figure 14: Double feature vector for both segmentations

be colored according to their distance to the their nearest pixel on the average segmentation using the rainbow colormap **4.1.2 Rainbow colormap**. These calculated averages show the user if a segmentation is located near or in the set with other segmentations or that the segmentation can be considered as an outsider in comparison with set of remaining segmentation.

Besides showing how much a segmentation differs from the set of remaining segmentations, this method also shows if the set of other segmentations is coherent or not. In order to visualize this, we use the calculated average segmentation from the previous step and use it to calculate the standard deviation for all the segmentations in this set with the average segmentations. This is visualized with the saturation of the previously calculated pixel color where a low saturation value, the pixel color is (almost) the same as the original assigned average distance color, means that the set of remaining segmentations where this segmentation is compared with is coherent. A high standard deviation value (gray/white) indicates that most segmentations are far away from the calculated average segmentation which makes the set incoherent.

In the end these two combinations can result in four different conclusions about a segmentation with respect to the set of other segmentations:

- The segmentation is not an outlier and the set of remaining segmentations is also co-

```
function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy

  loop
    plot(x0,y0)
    if x0 = x1 and y0 = y1 exit loop
    e2 := 2*err
    if e2 > -dy then
      err := err - dy
      x0 := x0 + sx
    end if
    if x0 = x1 and y0 = y1 then
      plot(x0,y0)
      exit loop
    end if
    if e2 <  dx then
      err := err + dx
      y0 := y0 + sy
    end if
  end loop
}
```

Code listing 10: Bresenham's line algorithm

herent. (Low average distance / low standard deviation)

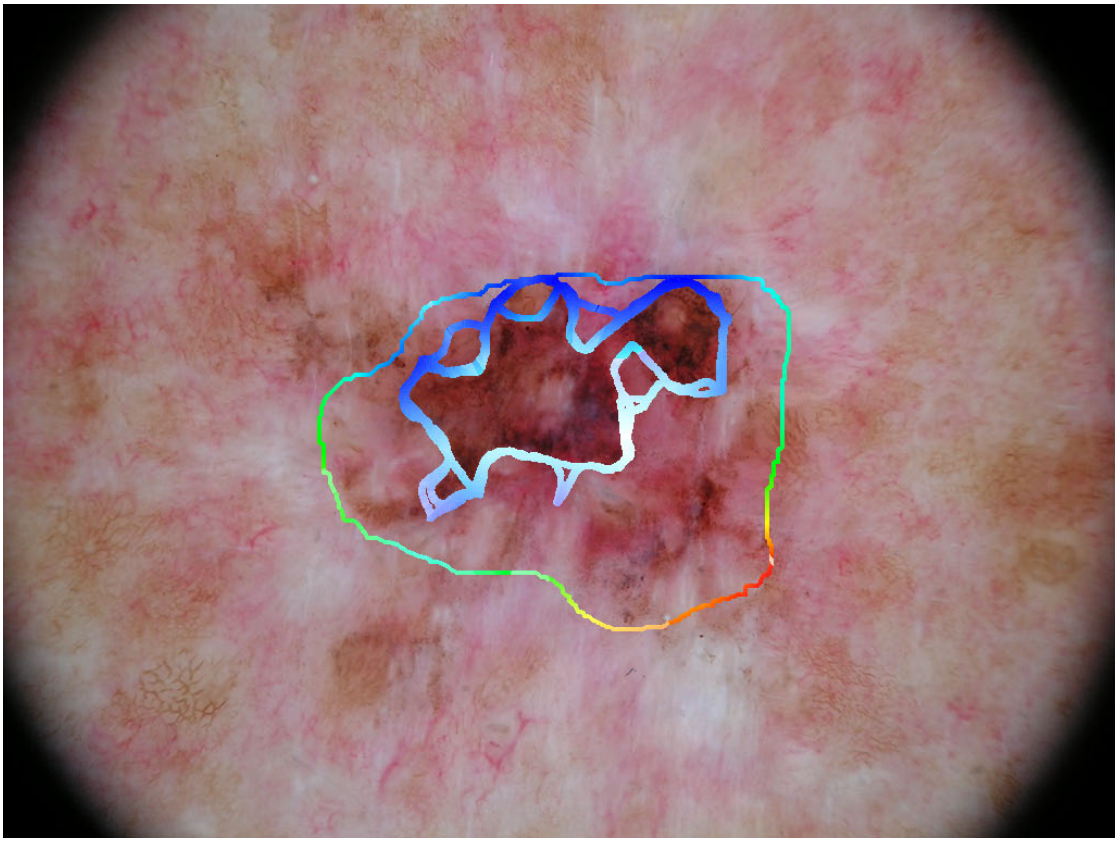
- The segmentation is an outlier but the remaining segmentations are coherent with each other. (high average distance / low standard deviation)
- The segmentation is not an outlier but all the segmentations are not coherent. (Low average distance / high standard deviation)
- The segmentation is a outlier but the remaining segmentations are also not coherent. (high average distance / high standard deviation)

In **Figure 15** we have shown an example of the N segmentations method used on the four different segmentations which were also used in **Figure 12**. **Figure 15a** shows the Nsegmentations with the use of the saturation value for visualizing the standard deviation and **Figure 15b** shows the same Nsegmentation result without using the saturation value also only displaying the average distances. As can be seen when looking at the colors, the inner three segmentations are relatively close to each other in comparison with the outer segmentation. This is easy to imagine since the average segmentation will be located closer to the three inner segmentations than to the only outlying segmentation. When looking at how the saturation values affects the color of the segmentations in **Figure 15b**, it can be seen clearly that it affects the inner segmentations at most. This is due the fact that every segmentation is compared to the set of remaining segmentation also excluding itself. Therefore for every segmentation of the three inner segmentations this sets exists of the 2 other inner segmentation and the only outer segmentation. Since the outer segmentation has a high deviation from the average segmentation, it increases the calculated standard deviation and thus affects the saturation value for the inner segmentations. The outer segmentation itself will reflect the set of three inner borders, which are much closer to the calculated average segmentation and therefore have a lower standard deviation thus a low affection by the saturation value.

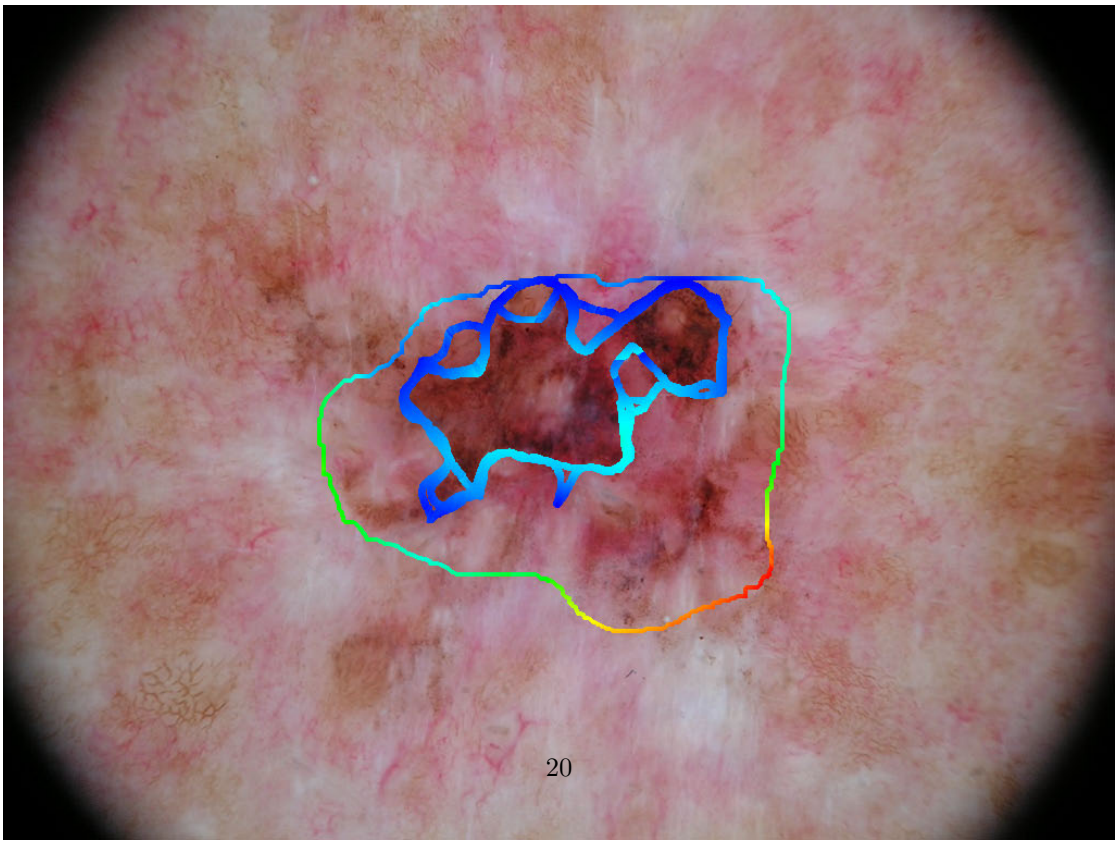
5 Future work and conclusion

We have made a c++/CUDA implementation of the image segmentation method[8] which was already implemented in matlab. As could be seen in section 2 our implementation is significantly faster than the matlab implementation which is easy to explain since our code can be executed in parallel among multiple CUDA threads. Except only porting the matlab code to c++/CUDA, we have also made a graphical user interface around it which makes the program more user friendly and we have implemented multiple visualization techniques which allows the user to compare multiple segmentations with each other as discussed in section 3. However there is still enough room for improvement and extension. We have listed a few possibilities below:

- The implementation of the image resiz-



(a) Nsegmentations with saturation.



(b) Nsegmentations without saturation.

Figure 15: N Segmenations

ing in CUDA should be rewritten to do the resampling using a convolution kernel. Downscaling of images can then combine the resampling kernel with a low-pass filter to reduce the effect of moire patterns.

- Out of the algorithms listed in **3 Algorithms** a couple that are currently implemented in C++ can be converted to CUDA, e.g. connected component labeling.

- The `cufft` library is optimized for problem certain sizes, such as powers of 2 and 3. The current implementation of convolution, **3.3 Convolution** does not take this into account, but can be extended to do so.

- As explained in **4.1.2 Rainbow colormap**, our program currently uses the Rainbow colormap for every visualization method but an option would be to implement another colormap or to implement multiple colormaps and add it as an option to the gui. That way the user can choose which colormap he/she wishes to use.

- Right now it is only possible to select a segmentation by using the drop-down boxes but it would probably more intuitive for the user, to let him select a segmentation by simply clicking on it when it is shown in the image.

- Another option would be to make it possible to let the user hoover over the pixels of a border and let the program display certain information like: pixel coordinates, calculated distance for this point etc.

- At the moment it is also only possible to clear all the segmentations at the same time. It would probably be easier if the user could select a specific segmentation that he/she would like to remove from the painted image.

Acknowledgements

We ackowledge the significant support to our project delivered by dr. Daniel Boda and dr. Adriana Diaconeasa (University of Medicine and Pharmacy "Carol-Davila", Bucharest, Romania) for the acquisition, manual segmentation, and assessment of the skin lesion segmentation tumors used in the design and evaluation of our automatic segmentation software developed in this project. We also acknowledge the financial support delivered by the grant PN-II-RU-TE-2011 -3-0249 provided by UFEFISCDI, Romania.

References

[1] Convert region of interest (roi) polygon to region mask. `http://www.mathworks.nl/help/images/ref/poly2mask.html` . Online accessed: 02-07-2013.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal* , 4(1):25–30, 1965.

[3] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the gpu. *Proc. ACM I3D* pages 83–90, 2010.

[4] R.C. González and R.E. Woods. *Digital Image Processing* . Pearson education. Pearson/Prentice Hall, 2008.

[5] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *IJCV* 1(4):321–331, 1988.

[6] G.A. Moore. Automatic scanning and computer processes for the quantitative analysis of micrographs and equivalent subjects. pages 275–326, 1968.

[7] Nobuyuki Otsu. A Threshold Selection Method from Gray-level Histograms. *IEEE Transactions on Systems, Man and Cybernetics* , 9(1):62–66, 1979.

[8] Alessandro Parolin, Eduardo Herzer, and Claudio R. Jung. Semi-automated diagnosis of melanoma through the analysis of dermatological images. *Proc. SIBGRAPI,* 71–78, 2010.

[9] Chenyang Xu and J.L. Prince. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing* 7(3):359–369, 1998.