Poisson image editing project

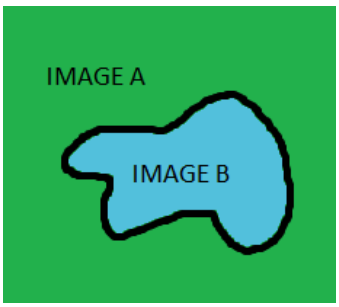
Submitted by:

David Cohn: 039057229

Tal Hakim: 301013439

Poisson Equations Project – explanation of the article

We implemented the math equations explained in the article "Poisson Image Editing" in this link http://www.cs.princeton.edu/courses/archive/fall10/cos526/papers/perez03.pdf.
A good explanation is also written in this link:
http://www.ctralie.com/Teaching/PoissonImageEditing/#source.
We'll write here our own explanation for the article to explain our code.
The following explanations are taken both from the article and from the link above:
Our intention is to copy a patch from a source image and paste it in the target image. Also, we need to blend the pixels in the patch so that it will look as the patch is part of the image (as much as possible).
Let's call the source image B and the target image A. when we copy a patch from image B and paste it in image A it should look like this:



Our goal is to blend image B inside image A so that it will look as a single image and not like someone pasted part of one image in a different image.
To do that we need to do the following:

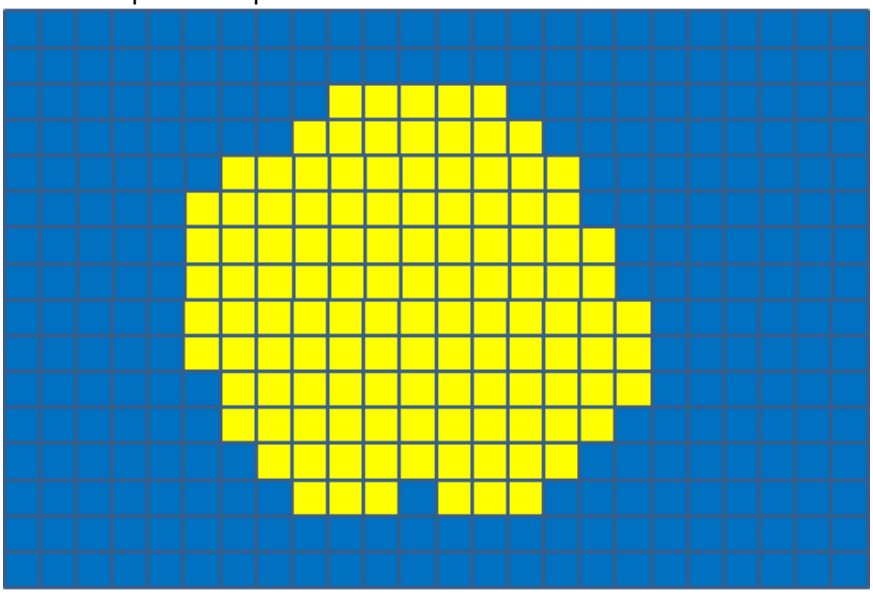
1. All the pixels on the boundary of the patch should have exactly the values of the pixels in image A which are in the same location.
2. The gradient of all the pixels in image B (which is the "details" of the image) should remain the same.

Assume p(x,y) is the grayscale color of a pixel in the patch. The pixel is located in (x,y) coordinate.
Then 4*p(x,y)-p(x-1,y)-p(x+1,y)-p(x,y-1)-p(x,y+1) is the gradient value of p(x,y).
Which means that we subtract 4*p(x,y) with the sum of p(x,y) neighbors (only 4-conn neighbors).

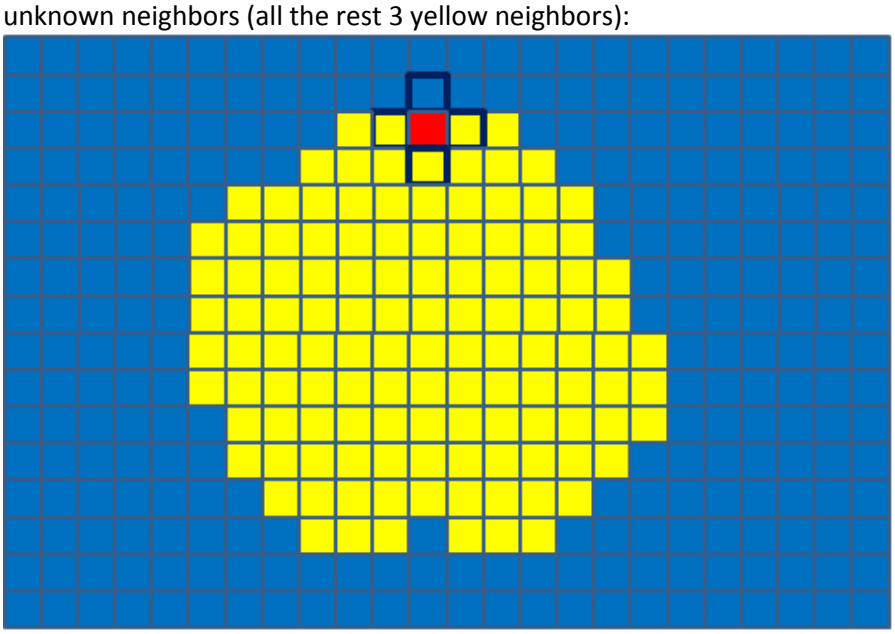
The gradient of the red pixel is 4 times its value minus the sum of the four blue pixels.
If p(x,y) is in the interior of the patch then all its neighbors are also in the patch and we don't know yet their value (p's value and its neighbors).
If p(x,y) is in the boundary of the patch then some of the 4 neighbors can be inside the patch (unknown values) and some can be outside the patch (known value. The pixels are part of image A).

For example if p(x,y) is in the boundary and p(x+1,y) is outside the patch (part of the original A image) and its grayscale value is 180. And all the rest 3 neighbors are inside the patch and therefore unknown, the the equation should be:
Grad(x,y) = 4*p(x,y)-p(x-1,y)-180-p(x,y-1)-p(x,y+1).
Grad(x,y) should be exactly the same gradient of the same pixel in the source image.

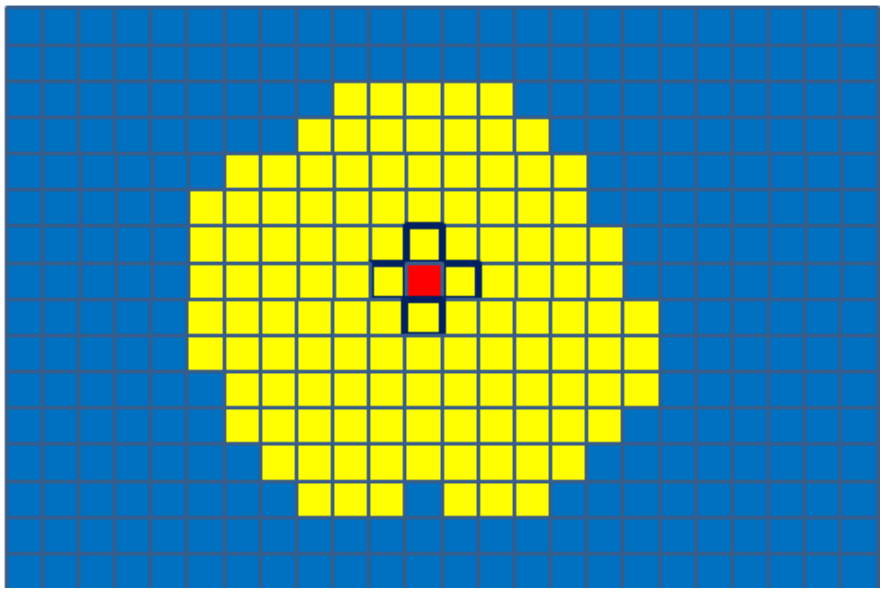
Here's how it looks visually:
The image we're pasting on is in blue and the patch we're pasting is in yellow. All the pixels in blue are known values, and the pixels in yellow are unknown values and these are the values we're supposed to find in the poisson equations:



The red pixel in the boundary of the mask has one known neighbor (the blue pixel above it) and 3 unknown neighbors (all the rest 3 yellow neighbors):



The red pixel in the interior of the mask has only unknown neighbors:



Assume we have N pixels in the patch we paste (boundary + interior of the patch).
Then we need to find the values of N pixels.
For that we need to solve N equations.
If we define a column vector x of length N then we need to find all the values of vector x.
As explained in the article we need to solve the equation Ax=b.
This equation is actually N equations for N pixels. Each equation is what we wrote above.
The A and b we defined in the code is as follows:
A is an NxN matrix (N is the number of pixels in the patch) which is built this way:
A = A4-neighborsMatrix.
A4 is an NxN diagonal matrix of with 4 in the diagonal and 0 everywhere else:

$$A_4 = \begin{pmatrix} 4 & & & & & \\ & 4 & & & & \\ & & 4 & & & \\ & & & 4 & & \\ & & & & \ddots & \\ & & & & & 4 \end{pmatrix}$$

neighborsMatrix is an NxN matrix indicating which of the pixels' neighbors are inside the patch.

If neighborsMatrix(k,m)=1 then k's neighbor m is inside the patch (otherwise the value is 0).

With that logic if neighborsMatrix(k,m)=1 then neighborsMatrix(m,k)=1. So neighborsMatrix is a symmetric matrix.

Row k of neighborsMatrix indicating which of the the 4 of the k'th pixel's neighbors are inside the patch. If any of the 4 neighbors are outside the patch then they are not part of the N pixels and they won't appear in the matrix.

It means that every row in neighborsMatrix has at most four 1's and the rest are 0's.

Most of the rows should have four 1's because most of the pixels of the patch are in the interior (in most cases).

Only the rows representing the pixels of the boundary might have less than four 1's.

Also since N might be a big number (many thousands) then neighborsMatrix is extremely sparse matrix.

b is a column vector of length N defined as follows:

$b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{pmatrix}$ where b_k=grad(k)+sum_of_pixels_outside_boundary.

Grad(k) is the gradient of kth pixel in the source image.

sum_of_pixels_outside_boundary is the sum of the pixels that are not part of the patch (pixels of the original target image).

The reason we add sum_of_pixels_outside_boundary to grad(k) is that in the A matrix we subtracted only the pixels that are inside the patch. We didn't do anything with the pixels that are outside the patch. So, in order to fix that we add them in the b vector.

We'll show in a small example how the equations work:

Assume our patch has N=80 pixels (usually it has many thousands of pixels).

Pixel number 70 is in the boundary that's why it has neighbors inside the patch and neighbors outside the patch.

Suppose pixel number 70 has 2 neighbors outside the patch (known values) with values 180, 95. And 2 neighbors inside the patch (unknown values).

We'll look only in the 70th equation of Ax=b.

A = A4-neighborsMatrix.
neighborsMatrix has only two 1's in the 70th row since pixel 70 has 2 neighbors inside the patch.

This is how row 70 should look like in matrixes A and neighborsMatrix:

$$\text{neighborsMatrix} = \begin{pmatrix} & & & & & & & \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 1 \\ & & & & & & & \end{pmatrix}$$

$$\text{A} = \begin{pmatrix} & & & & & & & \\ 0 & -1 & \dots & 0 & 4 & \dots & 0 & -1 \\ & & & & & & & \end{pmatrix}$$

Also:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{70} \\ \vdots \\ x_{80} \end{pmatrix}$$

$x_1, x_2, .., x_{80}$ are the pixels we want to find their values.

$grad(p70)$ is the gradient of the relevant pixel in the source image.

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{70} \\ \vdots \\ b_{80} \end{pmatrix}$$

Since pixel 70 has also 2 neighbors on the **outside** of the patch and their values are 180, 95 then:

$b_{70} = grad(p70) + 180 + 95$

If we solve Ax=b only for pixel 70 it should look like this:

$$\begin{pmatrix} & & & & & & & \\ 0 & -1 & \dots & 0 & 4 & \dots & 0 & -1 \\ & & & & & & & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{70} \\ \vdots \\ x_{80} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{70} \\ \vdots \\ b_{80} \end{pmatrix}$$

The 70th equation:

$4 * x_{70} - x_2 - x_{80} = b_{70} = \ grad(p70) + 180 + 95$

$\Rightarrow grad(p70) = 4 * x_{70} - x_2 - x_{80} - 180 - 95$

Here we can see that the gradient of pixel 70 is 4 times the value of its own pixel minus the sum of its neighbors.

That's why the equations are correct.

In the code we solve the equations using matlab command cgs.

In the following pages we'll explain the functions in the code:

Explanation of the code:

mainProject.m:

This is the main class to run from.

In this class we define the source and target images to work on. When running this class, the source image appears in a window, the user then creates the mask by clicking as much clicks as he wants with the left click of the mouse. To close the mask the user clicks on the right click and then a line from the last point of the mask will be connected to the first point of the mask. Then the user needs to put the mouse inside the mask and click the right click and choose "Create Mask". If the mask that the user drew is not accurate enough (not exactly surrounding the object) the user can click on the button "Use active contour" to make a new mask inside the mask he drew that will surround the object better. He can click on this better as many times as he want until he's satisfied with the mask. The user can always manually fix the mask.

After the user finishes with the mask and he's ready to paste it in the target image, he clicks on the button "Move to target image". Then the target image appears and the user can choose with the left click of the mouse where to locate the patch. He can do it as many times as he wants and each time he'll see the patch in the different location he clicked (not blended yet). Only when the user is satisfied with the location he then clicks on the right click in order to do the blending equations. And this is it. After that the patch should be blended in the target image.

//

Function blendMaskWithTargetImage(SourceIm, SourceMask, TargIm)

This function blends the patch from the source image into the target image.

Input parameters:

SourceIm - The source RGB image the mask was selected from
SourceMask - A logical matrix indicating the mask ('1' inside the mask and '0' outside the mask)
TargIm - The target RGB image to paste the mask in.

//

function neighborsMatrix = buildNeighborsMatrix(maskMatrix)

This function builds the neighbors matrix as explained in the article explanation at the beginning of this file. This matrix is needed for calculating the A matrix which is used for solving the poisson equation Ax=b (also explained above).

As a reminder from before, neighborsMatrix is an NxN matrix (N is the number of pixels in the patch) indicating which of the pixels' neighbors are inside the patch. A more detailed explanation is in the article explanation above.

Input parameters:

maskMatrix – A binary mask matrix for building the neighbors matrix ('1' inside the mask and '0' outside the mask).

Output parameters:

neighborsMatrix – the NxN neighbors matrix explained above.

//

function MaskTarg = calc_mask_in_targ_image(sourceMask, targImRows, targImCols, shift_in_target_image)

This function calculates the mask image in the target image. This function is actually needed only in cases the user clicked too close to the image edges. Since we don't want the patch to exceed the image boundaries, if the user clicked too close we "fix" the mouse location by moving it to the closest location where we can paste the whole patch.

Input parameters:

SourceMask – the mask image. This is a logical image. '1' inside the mask and '0' outside the mask.

targImRows – number of rows in the target image.

targImcols – number of columns in the target image.

shift_in_target_image – the (x,y) location the user clicked to past the patch. If this location is too close to the image edges it will be fixed so that the whole patch can be seen in the target image.

Output parameters:

MaskTarg – the recalculated mask to be used from now on. This is a logical image (the same sizes as the target image) where '1' is inside the recalculated mask and '0' in outside the recalculated mask.

//

Function [sourceImage, mask] = getUserMask(sourceImagePath)

This is the function in which the user is cutting a mask from the source image.

Input parameters:

sourceImagePath - The path of the source image.

Output parameters:

sourceImage – The source image which is an RGB matrix (meaning 3*M*N for M*N image).

Mask – A logical matrix in which the values '1' are inside the mask the user marked, and the values '0' are outside the mask the user marked.

//

function [targetLocation, button] = getUserTargetLocation(targetImg)

This function is called when the user clicks on the target image to decide where to paste the patch copied from the source image.

Input parameters:

targetImg – the target image (RGB image) the user clicks with the mouse.

Output parameters:

targetLocation – the location (x,y) the user clicked.

button – this is a number indicating which button the user clicked. The button options: 1 for left click, 2 for middle click and 3 for right click

//

function [MaskTargLogical, target_image_with_gradient_source_patch, TargImPaste_Source] = paste_source_into_targ(SourceIm, TargIm, SourceMaskLogical, shift_in_target_image)

This function first calculates the gradient of the source image and then pastes the part of the image that is in the mask into the target image.

Input parameters:

SourceIm – the RGB source image.

TargIm – the RGB target image.

SourceMaskLogical – the mask image from the source image. This image is logical (‘1’ inside the mask and ‘0’ outside the mask).

Shift_in_target_image – the (x,y) location the user clicked with the mouse in the target image (for pasting the patch).

Output parameters:

MaskTargLogical – the recalculated mask to be used from now on (in case the user clicked to close to the image edges).

target_image_with_gradient_source_patch – the target image with the gradient source patch inside the target.

TargImPaste_Source – the real patch of the source image pasted into the target image (no gradient is calculated here).

//

function targetImageWithBlendedPatch = PoissonColorImageBlending(target_image_with_gradient_patch, MaskTarg)

This function splits color image into 3 colors (R,G,B) and pastes source into target image.

Input parameters:

Target_image_with_gradient_patch – target image with the gradient of source image within the masked area.

MaskTarg – mask area for source to be pasted into. This is a logical image with the same sizes of the target image ('1' inside the mask and '0' outside the mask).

Output parameters:

targetImageWithBlendedPatch – target image with the blended pixels (masked areas solved by poisson equations)

//

function target_grayscale_image_blended = PoissonGrayImageBlending(target_image_with_gradient_patch, MaskTarg, neighborsMatrix, boundary)

This function is the heart of the project. Here we actually do the poisson calculations (as explained in the article explanation above) for blending the patch in the target image.

Input parameters:

target_image_with_gradient_patch – target image with the gradient of source image within the patch.

neighborsMatrix – neighbors matrix that was explained above in the article explanation.

Boundary - A matrix with 2 columns containing the (x,y) coordinates of the pixels around the mask (the boundary of the mask) that the user marked.

Output parameters:

target_grayscale_image_blended - the grayscale target image with the blended patch in it.

//

We also have the function **dpsimplify** in the project.
This function was written by Wolfgang Schwanghart and was copied from the link:
http://www.mathworks.com/matlabcentral/fileexchange/21132-line-simplification/content/dpsimplify.m

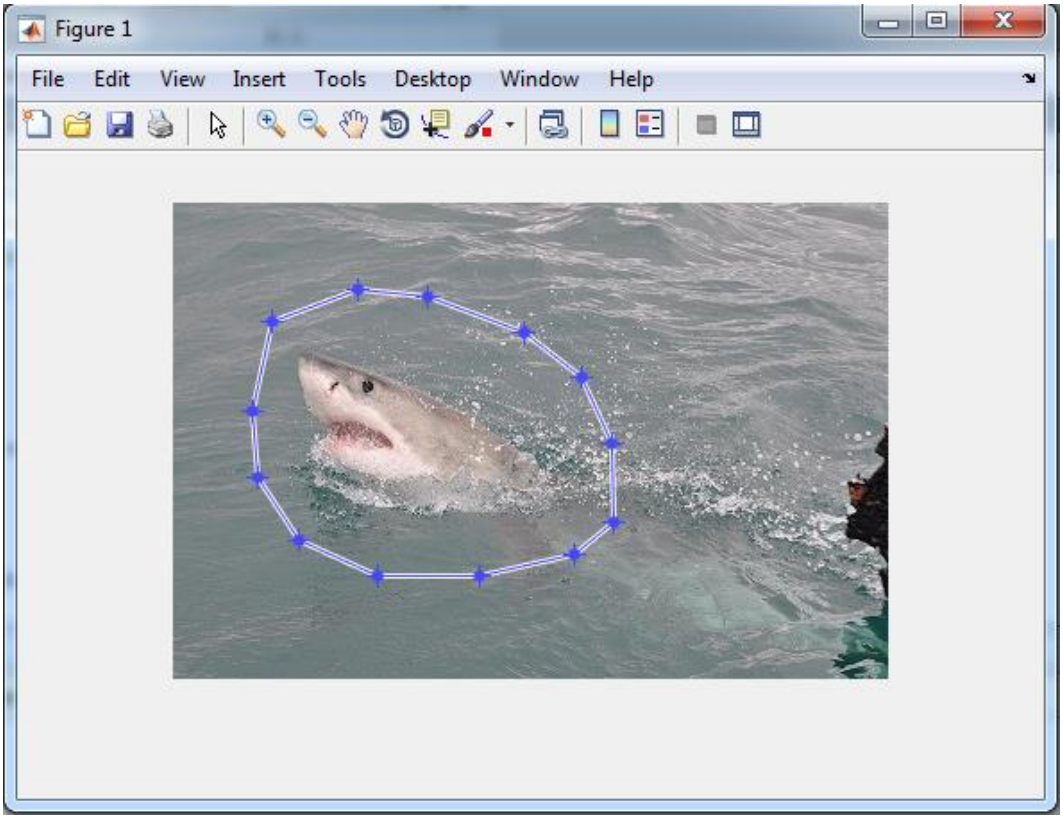
This function simplifies the mask polygon so that it will have less edges. We call it after we call active contour so the mask polygon will surround the object better. But then the polygon has a lot of edges and we want to simplify it in case the user wants to manually adjust the mask to be more accurate.

Results:

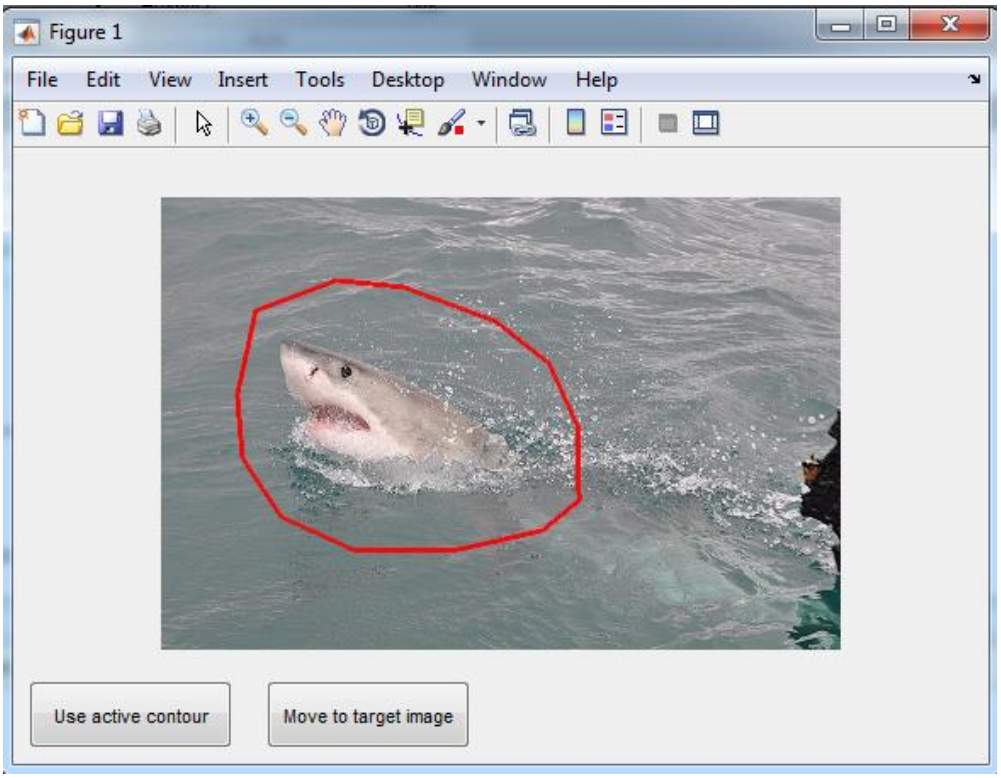
And here are some of the results:

Source image:

Here the user marks the mask around the object he want to copy:



The user then chooses “Create Mask” with the right click of the mouse and then two buttons appear below the image:



For this example, assume we’re already satisfied with the mask and we don’t want to active contour it. So we click on “Move to target image”.

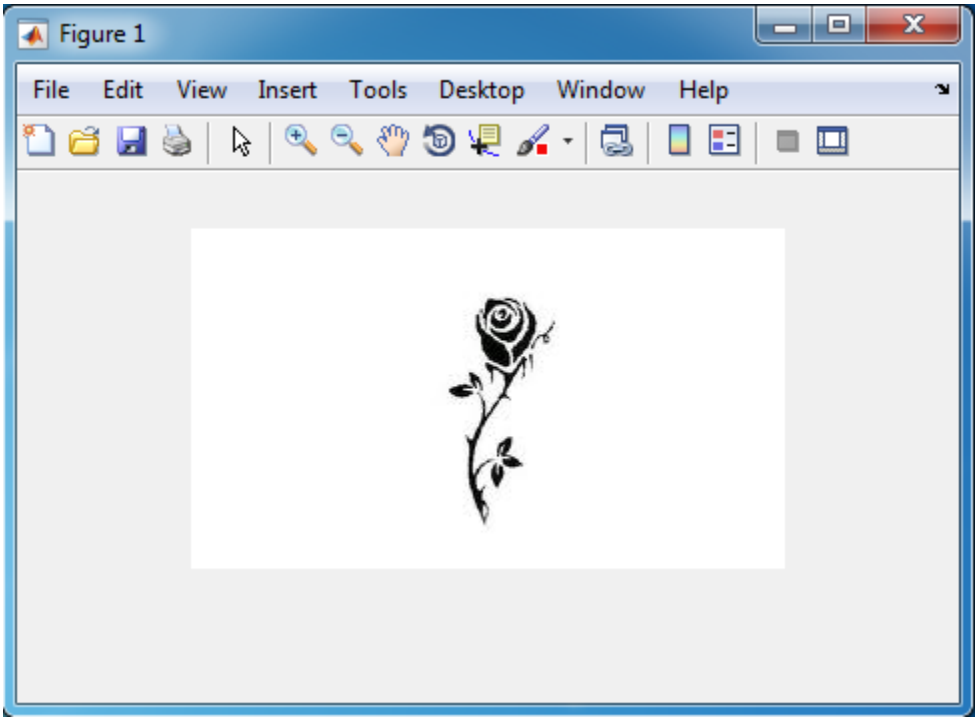
The user can choose as many times as he wants where to locate the patch with the left click of the mouse and each time he'll see the patch in the different locations he clicked (not blended yet). Only when the user is satisfied with the location he then clicks on the right click in order to do the blending calculations.

This is how it looks before the blending:

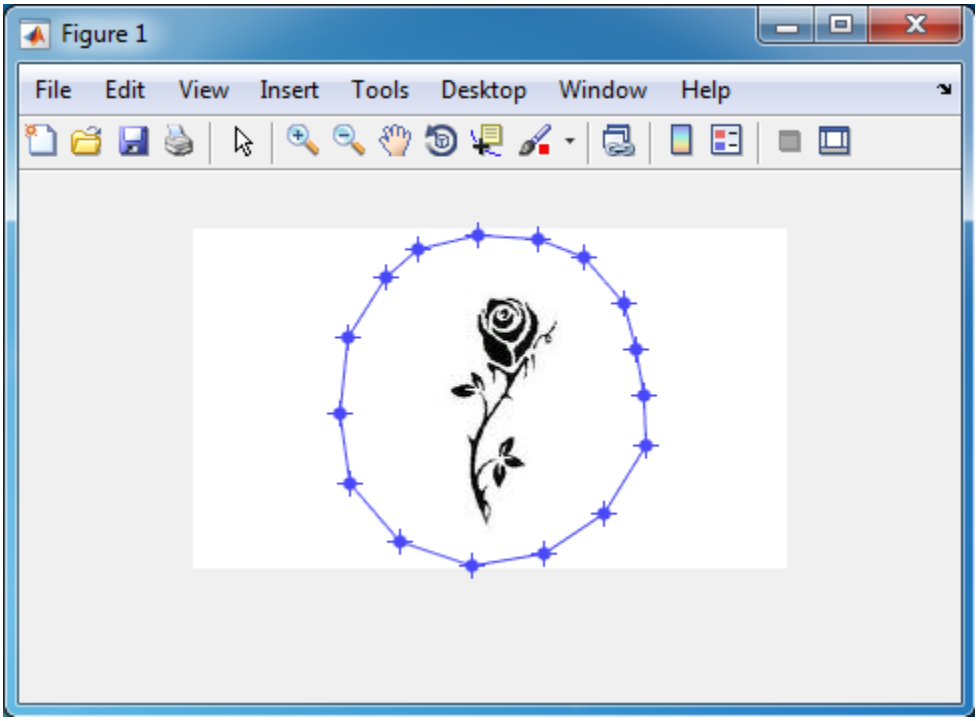
This is how it looks after the blending (the final image):

Another example (using active contour):

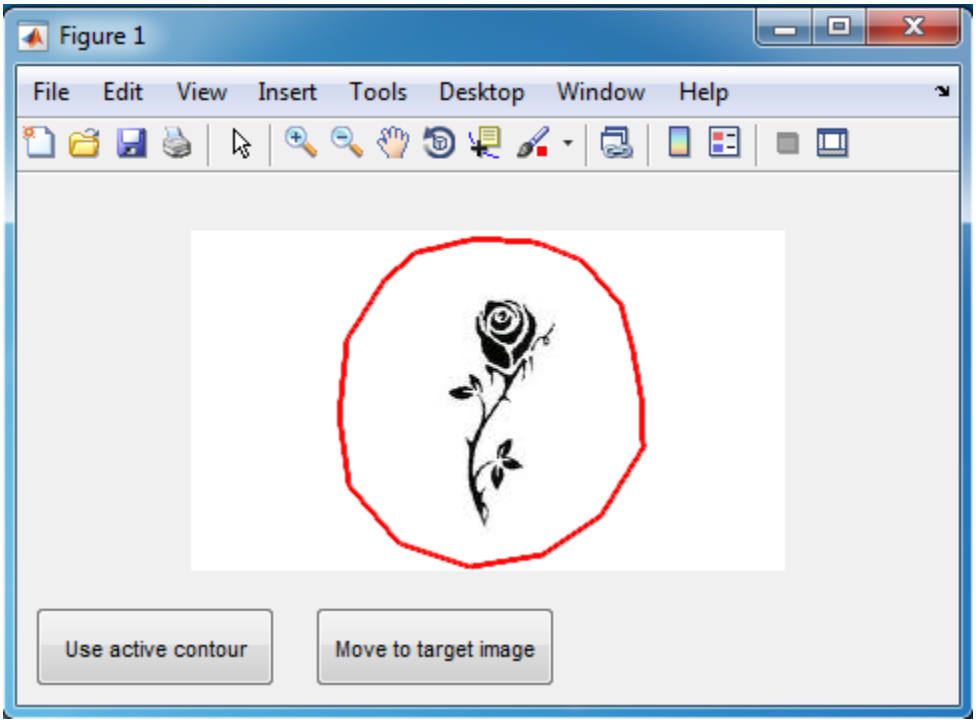
We use the following source image which is a flower tattoo.



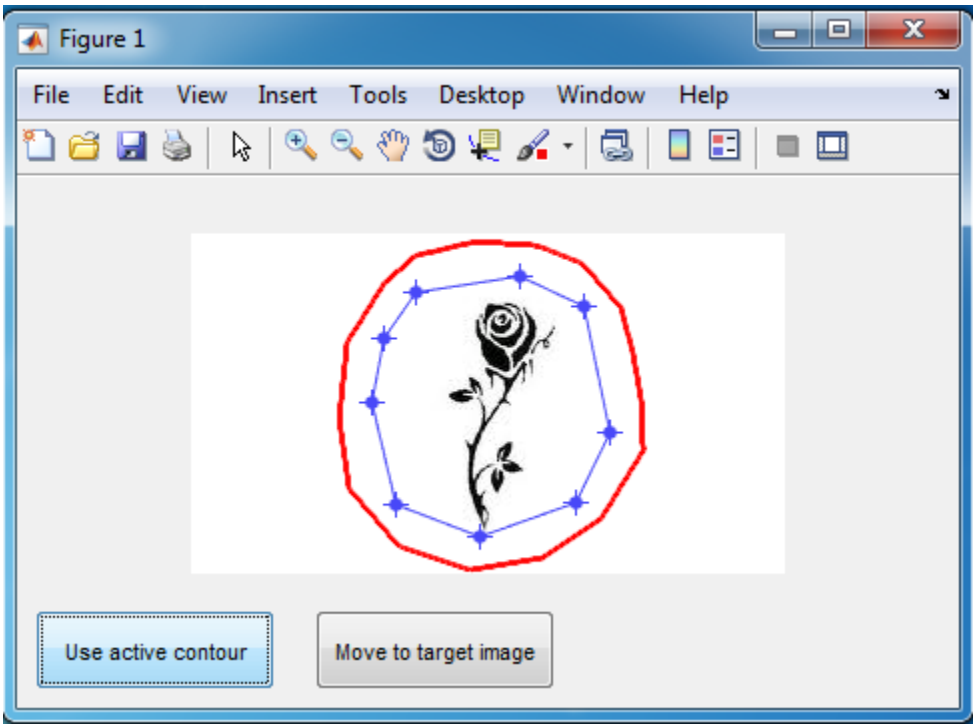
It is hard to mask the tattoo manually but we don't have to. We'll just surround a mask around the whole tattoo.



and then right click inside the mask and choose “Create Mask”. The two buttons bellow appear.



This time we will click the button “Use active contour” to fix the mask:



The mask is more accurate but not accurate enough so we’ll click again at “Use active contour”:

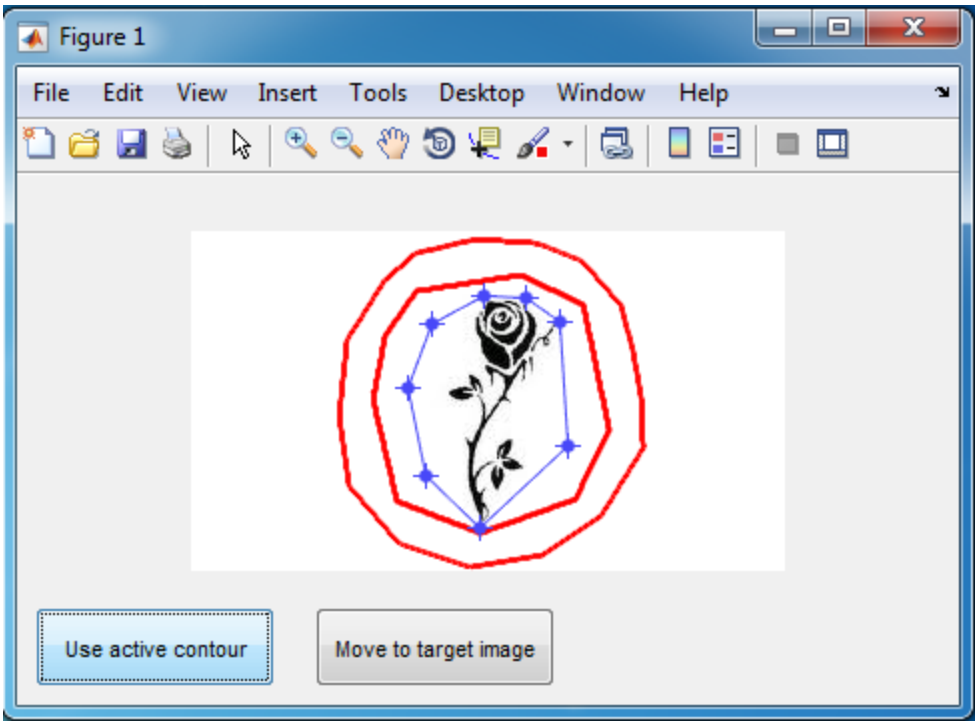


Figure 1
File Edit View Insert Tools Desktop Window Help
Use active contour
Move to target image

And again:

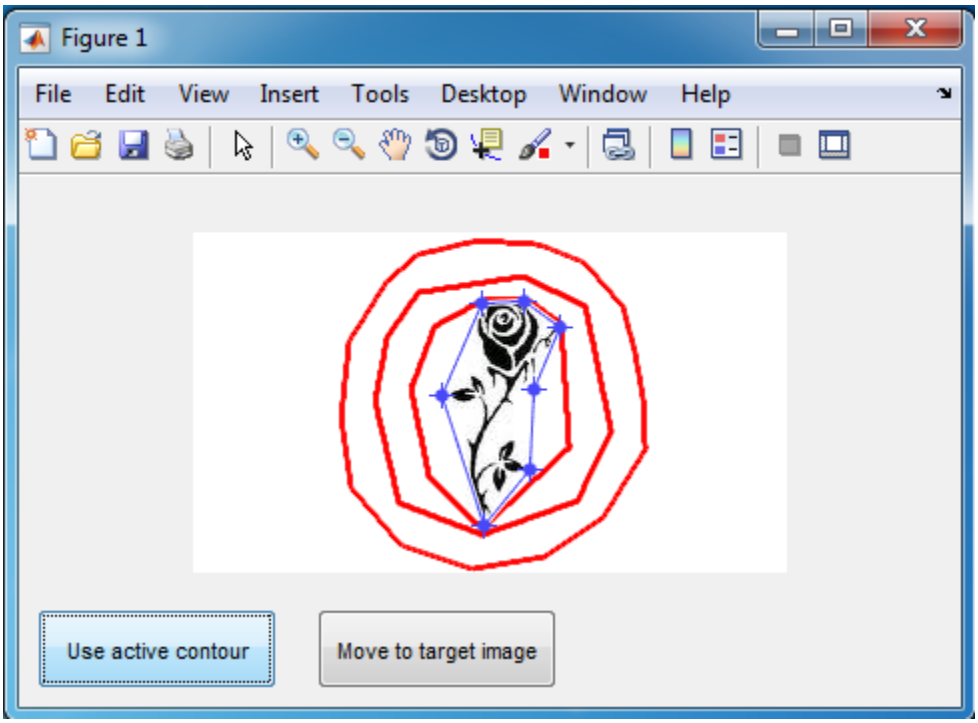
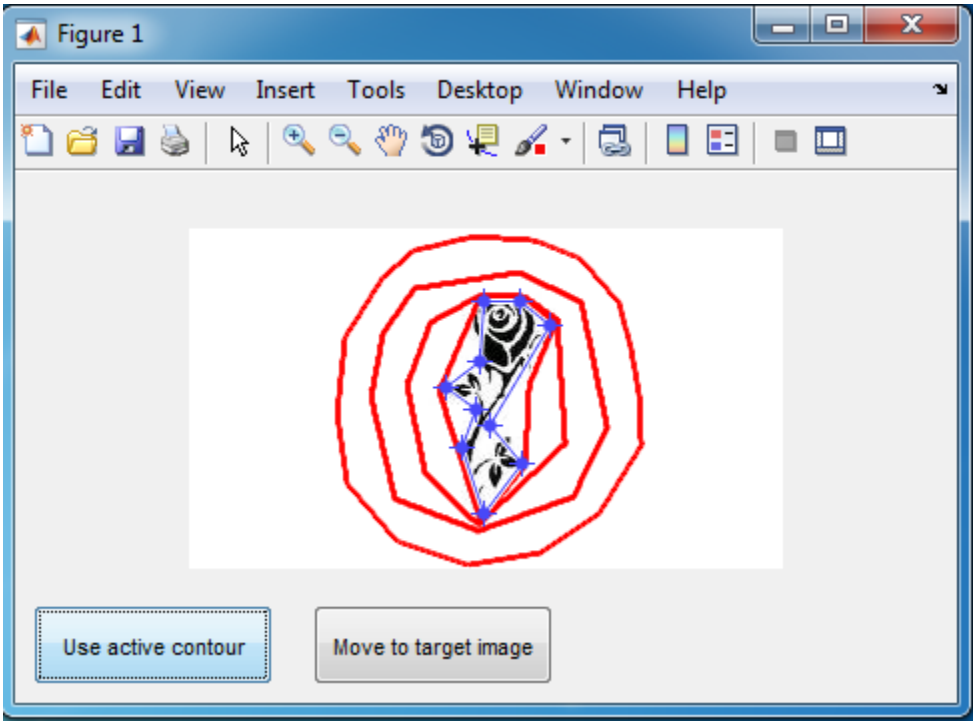
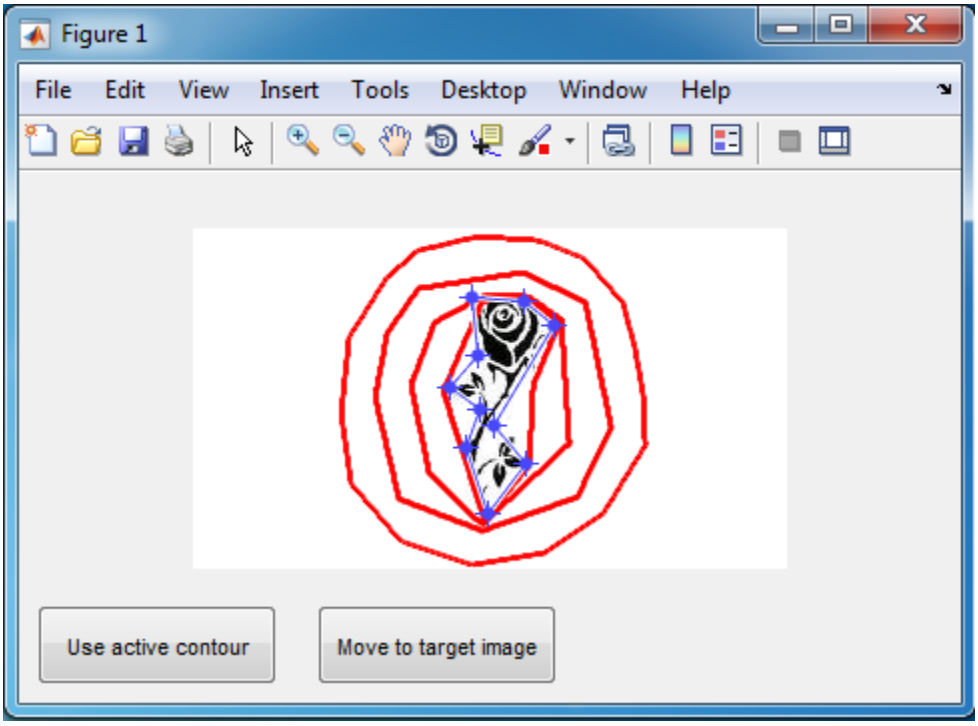


Figure 1
File Edit View Insert Tools Desktop Window Help
Use active contour
Move to target image

And again:



Now it's pretty accurate but we can manually fix the mask by moving the blue dots with the mouse (that's why we called **dpsimplify** function otherwise we had too many dots and it would have been much harder for the user to manually change the mask) :



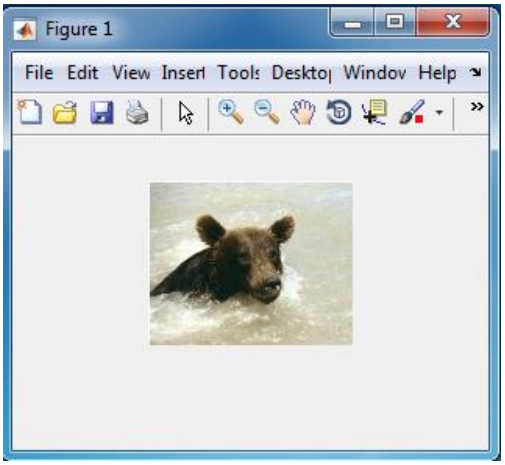
Now that the mask is ready to be pasted we then click on “Move to target image” and the following image appears:

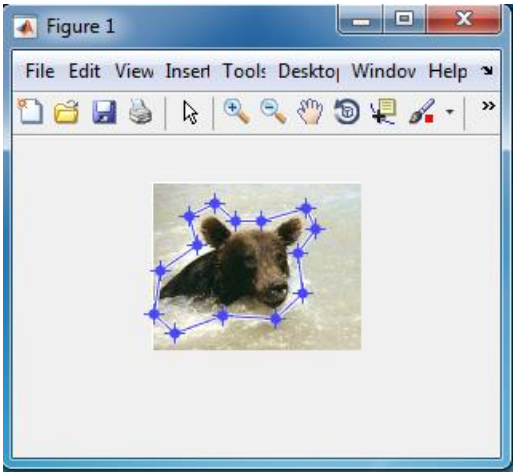
We paste the tattoo on the arm (you can locate the tattoo over and over with the left click of the mouse).

Finally you click on the right click to do the blending calculations. And this is the final image:

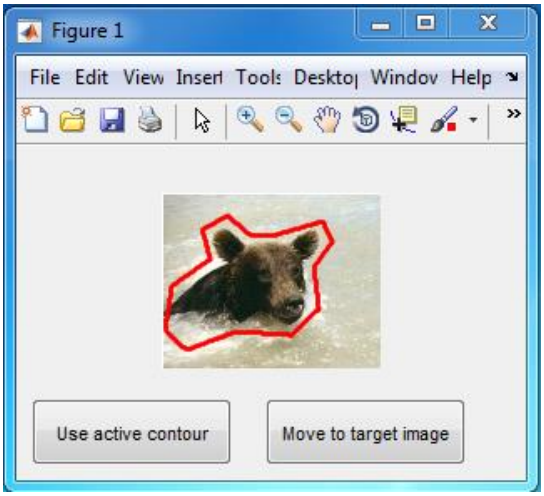
And last example:

Source image:





Right click and pressing on “Create Mask”:



Click “Move to target image” and the target image appears :

Paste the patch (choose where to paste with left click of the mouse):

Then right click to do the blending calculations: