

# Fundamentals of Computer Graphics, Vision, and Image Processing

## Exercise 1

Tal Netiv – 207908278 | Maya Carmi – 207435314

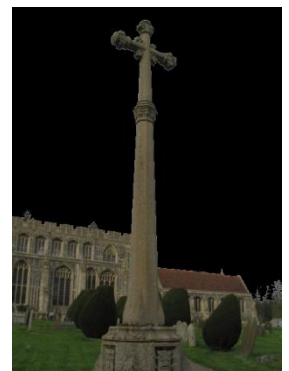
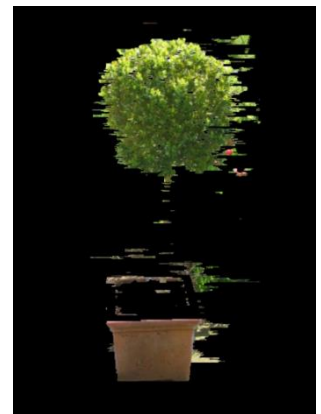
### Summary:

**GrabCut:** In our GrabCut algorithm implementation, we first initialized two GMMs, one to represent the foreground (FG) and one for the background (BG). We also construct the N-links capacities (as described in the article we were referred to) since it's only needed to be done once. We then iterate until energy convergence or up to `n_iter` (set to 5 as a default) iterations. In each iteration, we re-calculate the GMMs according to the latest mask we received. We then construct the graph that contains as many vertices as there are pixels in the image, plus one vertex set to be the source vertex and another one to be the target. We add edges between the source vertex to each pixel, and between each pixel to the target, in addition to edges between neighboring pixels. We then find the minimal cut, update the mask and check for energy convergence.

For most of the images we were able to get decent results, both quality-wise and quantity-wise (accuracy and Jaccard), but we did encounter a few less-satisfying results.

#### Failure Cases:

- **Banana1:** The result we got from running our algorithm with the default variables did not get a clean cut of the banana, unlike the cut of banana2. The great difference between these two images is their background, with banana1 having a background very similar to the object, making it harder to detect the borders between the banana and the surface it's located on. However, when using a smaller number of GMMs, we did manage to get a better cut of the banana (added to the right is the result we got from running the algorithm with 2 GMMs on that picture). Our assumption is that using a larger amount of GMMs caused overfitting and slowed down the convergence, while the use of 2 GMMs forced an earlier hard segmentation. In addition, in the banana's case, running the algorithm for more iterations did manage to improve our results and get to convergence, but it took a lot more time.
- **Bush:** Unfortunately, we did not find a combination of variables that managed to cut the bush out from the image. We think this might be due to the bush's leaves being very similar to their background, and also due to the stem's width (which is very thin and therefore not distinguished from the background). Even after 50 iterations, the background did not get cut out completely, and the case of a smaller amount of GMMs resulted in cutting out too much of the object (the top part of the bush's flower box and most of its stem was deleted in the result, as added to the right).
- **Cross:** The result for the cross image with the default variables did not cut out the cross as desired, leaving most of the sky in the foreground. Even when running it for many iterations, or when trying to set the rectangle to be tighter around the object, the same problem kept occurring. This probably happens because the rectangle we need to get the actual object is quite loose around the object, being that the building's width is significantly larger than the cross itself. However, when we decreased the number of GMMs to be 1, we managed to get a clean cut of the entire object.



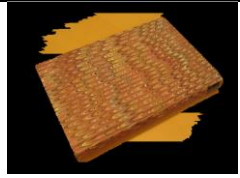








**Poisson Blending:** In our Poisson Blending algorithm implementation, we first had to check the dimensions of the input images, to make sure that they're compatible (meaning the dimensions of the input images do not

exceed the dimensions of the target image) and if they were and it was needed, we padded the input images with zeroes to center them on the background image. We then started the blending process, with first creating the Laplacian operator. We then moved on to construct the matrix A that will be use in the equation; first, the corresponding row in the A matrix of a pixel that was labeled 0 in the mask image, was set to be 1 – meaning the result will use the data from the target image for that specific pixel (since it's considered as background in the mask image). For pixels that are labeled as foreground in the mask image and are border pixels, we remove their entry from A (note that this is the 'difficult approach' from Oren's clarification, explained in more detail at the end of this document). To solve the  $Ax=B$  equation we also construct the vector B; we do so for the different colors (RGB) separately and merge them together. The solution of the equation results in the final blended image.

**Failure Cases:**

- The cases in which we got unsatisfying results were ones that were generated using masks that weren't tight around the object. In those results we got blurred smudges around the object where the excess background was still considered foreground (more about this at the Poisson blending section of this document). We will mention of course that even though the blending was not flawless, it did work nonetheless.

Image Name	Accuracy	Jaccard	Avg. Time	Result
banana1	0.7131998697916667	0.47130126916499143	00:02:21	
banana2	0.98533203125	0.9407323617614564	00:01:45	
book	0.8960514322916666	0.7801605441427548	00:02:34	
bush	0.8387629629629629	0.46681527024213404	00:01:04	
cross	0.6118185185185185	0.4286064756006477	00:02:07	
flower	0.9946296296296296	0.9729472564786656	00:01:10	
fullmoon	0.9956043956043956	0.9320543565147882	00:00:18	
grave	0.9714962962962963	0.798802645682465	00:00:52	
llama	0.9836383411358585	0.9112010950153987	00:00:41	




memorial	0.9789703703703704	0.8904960271542082	00:01:02	
sheep	0.9961074074074074	0.9294630872483222	00:00:44	
stone2	0.996240234375	0.9846538139590503	00:01:10	
teddy	0.9853581286715266	0.9353591376010624	00:00:16	

Note that these results were received either after 5 iterations or when convergence was achieved (set to be energy difference  $< 800$ , as we found it to be a good measure for assuming pixels would not move so much between FG and BG anymore) and were run with the original rectangle we received and 5 GMM components. Furthermore, the runtimes on our computers did vary significantly even when running the same picture with the same arguments, possibly because of our computer's CPU. Note also that most of the time of each iterations is taken up by the built in function of mincut.




To analyze the effect of different variables, we'll run trials on three images: banana1, book and llama.

# Banana1







## Blur

Low	Without (Origin)	High
<p>With blur (5,5):</p>  <p>Accuracy=0.7643326822916666 Jaccard=0.5208098913172979</p>	 <p>Accuracy=0.7131998697916667 Jaccard=0.47130126916499143</p>	<p>With blur (20,20):</p>  <p>Accuracy=0.7600455729166666 Jaccard=0.5158357963875205</p>

## Number of GMMs

3	5 (Origin)	9
 <p>Accuracy=0.8852864583333333 Jaccard=0.6901896313748934</p>	 <p>Accuracy=0.7131998697916667 Jaccard=0.47130126916499143</p>	 <p>Accuracy=0.7374544270833333 Jaccard=0.49307371279163315</p>

## Rectangle Initialization

x=31, y=35, w=603, h=418	x=16, y=20, w=620, h=436 (Origin)	x=0, y=0, w=639, h=479
<p>Rectangle:</p>  <p>Results:</p>  <p>Accuracy=0.771474609375 Jaccard=0.5272843579556932</p>	<p>Rectangle:</p>  <p>Results:</p>  <p>Accuracy=0.7131998697916667 Jaccard=0.47130126916499143</p>	<p>Rectangle:</p>  <p>Results:</p>  <p>Accuracy=0.5886458333333333 Jaccard=0.38324881767556723</p>

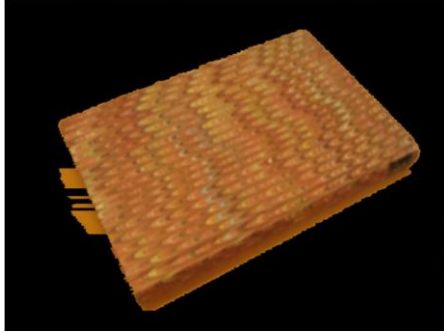


# Book

## Blur

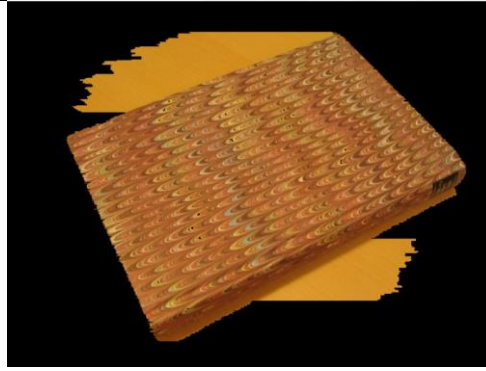
### Low

With blur (5,5):



Accuracy=0.9695279947916666  
Jaccard=0.9238013838013838

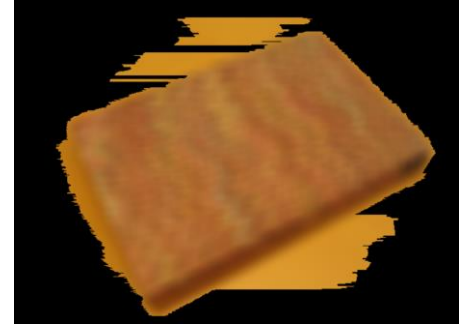
### Without



Accuracy=0.8960514322916666  
Jaccard=0.7801605441427548

### High

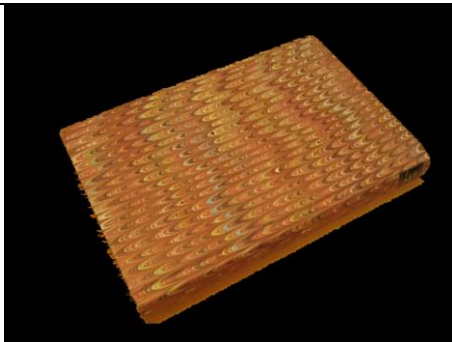
With blur (20,20):



Accuracy=0.86900390625  
Jaccard=0.7381714553404123

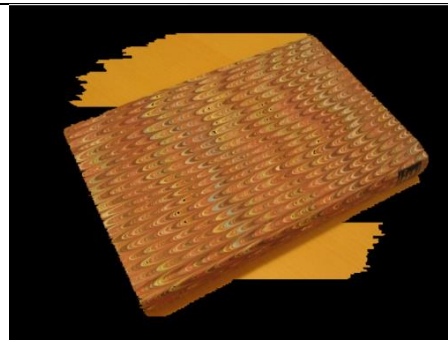
## Number of GMMs

### 3



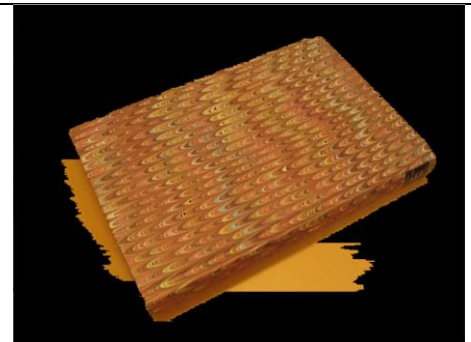
Accuracy=0.9774641927083333  
Jaccard=0.942466072185425

### 5



Accuracy=0.8960514322916666  
Jaccard=0.7801605441427548

### 9



Accuracy=0.9321712239583333  
Jaccard=0.8446935535563886

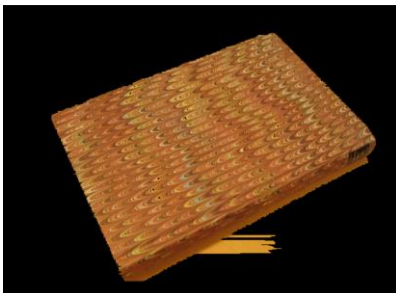
## Rectangle Initialization

x=75, y=40, w=600, h=445

Rectangle:



Results:



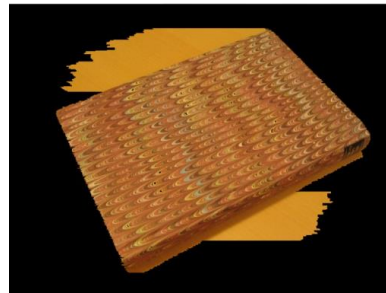
Accuracy=0.9716861979166667  
Jaccard=0.9286411630062925

x=59, y=24, w=618, h=459 (Origin)

Rectangle:



Results:



Accuracy=0.8960514322916666  
Jaccard=0.7801605441427548

x=0, y=0, w=639, h=479

Rectangle:




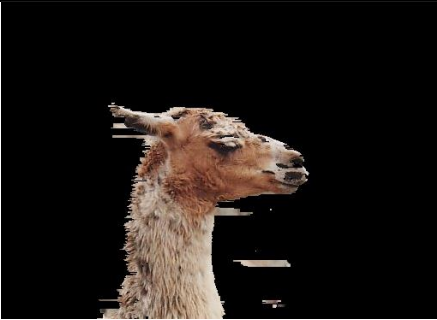

Results:



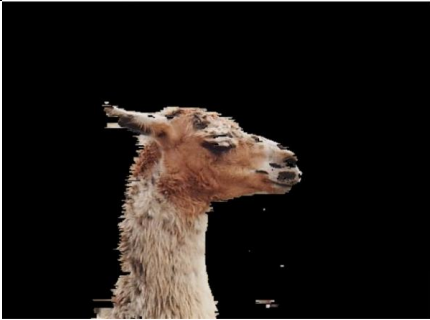


Accuracy=0.6849186197916667  
Jaccard=0.539766918830505

# Llama







## Blur

Low	Without (Original)	High
With blur (5,5):  Accuracy=0.9760880187891111 Jaccard=0.874111366213936	 Accuracy=0.9836383411358585 Jaccard=0.9112010950153987	With blur (20,20):  Accuracy=0.9789147922216442 Jaccard=0.8859651615469865

## Number of GMMs

3	5 (Origin)	9
 Accuracy=0.9774641927083333 Jaccard=0.942466072185425	 Accuracy=0.9836383411358585 Jaccard=0.9112010950153987	 Accuracy= 0.9877839252218596 Jaccard= 0.9317761671410546

## Rectangle Initialization

x=120, y=118, w=360, h=370	x=112, y=106, w=370, h=371 (Origin)	x=0, y=0, w=512, h=370
Rectangle:  Results:  Accuracy=0.9836068157815924 Jaccard=0.9106401260203351	Rectangle:  Results:  Accuracy=0.9836383411358585 Jaccard=0.9112010950153987	Rectangle:  Results:  Accuracy=0.4562191642628584 Jaccard=0.12943195296136473

### Blur:

Blur naturally reduces noise levels, as it smoothens the image, and makes borders more clearly defined. Furthermore, low levels of blur reduce high-frequency details, which might remove small fluctuations in pixel values that may not be meaningful or even visible to human eye. On the other side, there's a tradeoff, since blur of the image costs in data loss, which in times might be significant. For example, in the Llama image, we see that applying a small kernel blur isn't very beneficial, while a big kernel blur clears out unwanted stains, but results in major data loss and a very blurry llama. On the contrary, a low blur on the book image significantly improves both qualitative and quantitative results of the cut (although smoothens the book's pattern), but high blur is still ineffective as it doesn't make any improvement and makes the book object very unclear.

### Number of GMMs:

As the results show, the number of GMMs can significantly affect the algorithm's results. Eventually, we try to cluster every pixel with other pixels which resemble it. Increasing the number of GMMs is considered to capture finer details, and is appropriate when dealing with complex textures or minor color variations which are important to distinguish. It might also overfit noise as FG, such as in our book example, where 5 or 9 GMMs resulted in noise fitting in, while 3 components resulted in a great, clean result. This makes sense as the BG has a great resemblance to the book, and the borders aren't very strict. Same applies to our banana1 example (even better results received with 2GMMs, as noted beforehand). Fewer GMMs might simplify the segmentation in cases where fine details aren't very important. It might also underfit data to be considered as BG.

### Rectangle Initialization:

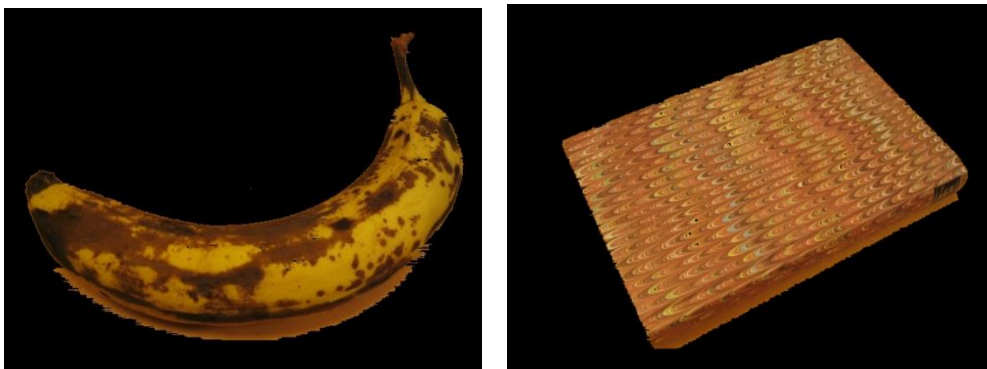
To start with the obvious, tighter rectangle around the object in the image dramatically improves the results of the algorithm, and its runtime as well. Some of the benefits of tightening the rectangle are:

- 'Decide' ahead of time for the algorithm that some pixels are BG, resulting in less pixels which are 'PR' (which reduces noise when clustering).
- Avoid error prone areas in the picture where BG and FG are hard to distinguish.
- Initial estimate of the object boundaries becomes more accurate.

Whatsoever, sometimes, tightening the rectangle might actually backfire, as the remaining BG pixels are somewhat 'isolated', and the absence of their neighboring BG pixels makes it more difficult to categorize them as BG.

### Further Discussion:

In our work, **the maximum number of iterations** was set to 5 as it was a good tradeoff between sufficient results, and reasonable runtime. But, we noticed that for some images significant improvements can happen if we increase the number of iterations that we run. For example, the algorithm results for banana1 and book weren't excellent after 5 iterations, but after 12, 9 iterations (in correspondence) we received these superb results.



More iterations refines the segmentations, as there are more opportunities for every pixel to settle in the component that fits it best. In each iteration, GrabCut updates the probability of a pixel belonging to the foreground or background based on the current mask. The segmentation mask is then refined by assigning pixels with



probabilities above a certain threshold to the foreground and those with probabilities below another threshold to the background. This process is repeated for a specified number of iterations until convergence is reached. By adding more iterations, the algorithm has more opportunities to refine the segmentation mask and improve the accuracy of the final segmentation result. However, adding too many iterations can result in over-segmentation or even convergence to a suboptimal solution. Indeed, we found that the improvement in the results when running more iterations did not show for every single image, some of them got roughly the same results even after 20 more iterations. In general, it seems like when the results aren't good enough it could be helpful to run several more iterations on the images we're trying to cut, and it could result in better masks.

In the original code we received for this project, the mask was initialized so that any pixel outside of the rectangle would be considered as background 100%, and any pixel inside the rectangle as possible foreground. Moreover, **the middle pixel** of the rectangle was considered as 100% foreground because of the heuristic we discussed in class. When we run the algorithm with or without the definition of the center point as FG on an image where the center point isn't an actual part of the object (for example, on the image banana1), we found a 2% improvement in our results.

## Poisson Blending

First, we take the masks we got from running the GrabCut algorithm on each of the images in the dataset, and we use our implementation of Poisson Blending to apply them on one of the backgrounds:

1. banana1 on grass mountains



2. banana2 on wall



3. book on table



4. bush on wall



5. cross on table



6. flower on grass mountains



7. full moon on wall



8. grave on table



9. llama on grass mountains



10. memorial on wall



11. sheep on table



12. stone2 on grass mountains



13. teddy on wall





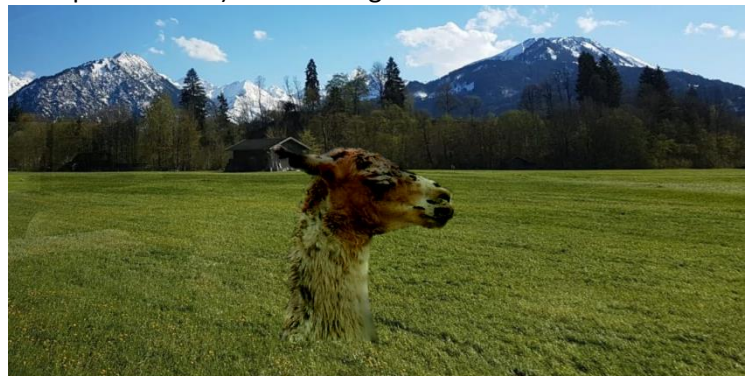
We now want to see the results of applying the Poisson blending on images with an **inaccurate mask**. First, we'll compare between the results of the blending of the mask we created for banana1 on table to the result of the blending of the mask we received for it, and we'll do the same for the cross image.



In Poisson blending, the goal is to seamlessly blend an object from one image into another image by minimizing the difference between the source and target images. The mask we use defines the region of the source image that should be blended into the target image. If the mask is not tight around the object, some parts of the background may be included in the blending process. These parts will then influence the Poisson equation's solution, resulting in blurred or distorted areas around the object, as we can see clearly above the banana and around the cross. This is because the gradient values of the background outside the object's boundary will be different from the gradient values of the object, leading to a discontinuity in the solution of the equations used for the blending itself. Therefore, for a seamless and consistent blending to happen, it is important to have a tight mask around the object to avoid any unwanted influence from the background.

#### Further Discussion:

**Colors** could have a great effect on the result of Poisson blending. For example, we'll look at the llama picture blended into the wall (which is quite close to its color-palette-wise) or into the grass mountains.



In the algorithm, we color the mask's inner border pixels with the colors of the target image pixels. Then, when solving the Poisson equation, the mask's gradients are used to blend the image into the background image, so that the colors will blend smoothly. In this case, it refers to the transition from the llama image to the background image. When we use the wall background image, the colors will blend smoothly with the llama image, creating a warm and harmonious look. The colors of the brown wall will blend gradually with the brown color of the llama, so the result will appear more subtle because the color transition is less noticeable. On the other hand, when we use the grass mountains background image, even though the algorithm will know how to create a subtle gradient effect that makes the final blending look clean, the gradient will be more noticeable and less natural.