SEAM CARVING

Luca Silva - Luca Benetti
November 19, 2020

1 INTRODUCTION

Seam carving is an image resizing algorithm, its aim is to reduce the dimension of a given image without scaling or cropping the image itself. Thanks to this property the algorithm can be really useful when we want to reduce an image preserving the original dimension of some important objects.
To pursue this goal in each iteration the algorithm identifies a seam, i.e. the less important path of the image, and immediately removes it.
To measure the importance of a pixel, the *Sobel differential operator* is used. This operator returns, for each pixel in the image, the modules of the vectors having as its' components the discrete directional gradients. This quantity is a 3D vector, as every pixel as three components (RGB). To convey the "importance" of a pixel in a single scalar, the algorithm simply takes the norm or the mean of the components of the latter vector.
Once the importance of each pixel is measured, the algorithm identifies the seam with a dynamic programming approach. It then removes it and, after having paste the two divided parts of the image, it starts a new iteration.
In section 2 we will give some general specifications about our implementation, while in section 3 we will describe precisely the different functions contained into the code. In section 4 we will describe an adaptive seam carving procedure. This procedure, at some benchmark iterations, decides if it is better to remove horizontal or vertical seams for the subsequent iterations.

2 GENERAL SPECIFICATIONS

Before deepening the contents of the different functions let's list some important specifications:

- At the beginning of the module we define a Mutable struct "Img_Aux", this struct contains everything needed to perform seam carving, in particular it contains:
  - *img*: the transposed image.
  - *bright*: a matrix with the average intensity on the three RGB components of each pixel.
  - *sobel_x*: the Sobel kernel on the x-axis.
  - *sobel_y*: the Sobel kernel on the y-axis.
  - *intensity_gradient*: a matrix containing the "importance" of each pixel computing as described before.
  - *aux_energy*: a matrix containing in position (i, j) the energy of the least energetic path having with final pixel the one in position (i, j).
  - *moves*: a matrix having in position (i, j), the value -1 if, when you are in position (i, j), the optimal path containing the pixel in position (i, j) contains the pixel in position $(i-1, j-1)$, 0 if it contains the pixel $(i, j-1)$, 1 if it contains the pixel $(i+1, j-1)$.
  - *seam*: an array containing in position i, the index of the row of the optimal path when you are in column i.
  - *iterations*: the number of the iterations that seam carve has already performed on the image.

  The reason for defining such a struct in that, when performing the different seam carving iterations, we can just dynamically modify this structure rather than redefining the objects needed at every iteration in the functions and then returning them; this increases significantly the performance of the code.
- The image we will modify will be not the original one, but its transpose (we will remove horizontal seams). This allows us to have mostly columns wise operations, which as we know in Julia are more efficient due to how arrays are stored in memory.
- In the execution of the code, we distinguish two types of functions: The ones to be executed at the first iteration of seam carving, and the ones to be executed at the other iterations. At the first iteration we have to execute all the operations necessarily, while at other iterations doing so would be very inefficient. We can in fact just update the previous values carefully (we will better explain the update procedure in section 3).
- We do not reduce the dimension of the image at every iteration, as this would be costly. The auxiliary struct has an integer field labelled *iterations*, which keeps track of how many iterations of seam carving were already performed on the image. In this way, when we remove a seam, we can consider as the new image the original one minus the last *iterations* columns, discarding all the pixels in the bottom rows. In those neglected positions there is just garbage, and we can cut them just once at the end of the seam carving procedure.

- To compute the energy of the borders, we add a frame of black pixels (pixel values 0). In this way we do not have to differently treat the explicit calculation of the convolution on the borders. This operation does not have a significant cost, while it gives us a significant gain in compactness and elegance of our code.
- Similarly, when we have to find the path with the least energy, we add to our image an additional row at the bottom and another one at the top with all values equals to infinite. In this way, during the dynamic programming procedure, we do not have to deal with borders in a different way. By putting these equal to infinity, we avoid the possibility of the seam going outside the original image. Again this operation does not have a significant cost, while we have a significant gain in compactness and elegance of our code.

3 DESCRIPTION OF FUNCTIONS

In this section we give a more detailed description of the functions that make up our code:

- **initialize_aux**: This function initializes the *Img_Aux* mutable struct we defined before given an image. Note that if we want to perform seam carving on the transposed image, we have to pass it directly as such to this function.
- **get_energy**: This function is called only at the first iteration, and computes the energy of each pixel of the matrix as described in the introduction.
- **get_seam**: This function also is called only at the first iteration, and it performs the dynamic programming algorithm to find the least energetic horizontal path.
- **remove_seam**: This function removes the found seam and reassembles all the auxiliary matrices the need to be up-shifted: *aux_energy* matrix, *bright*,*moves*, *intensity_gradient* and image. When removing the seam, for every row_index of the seam, we move all the pixels below this index of the corresponing column up by one row, while the ones above remain untouched.
- **update energy**:This function computes the energy matrix (*aux_energy*) after the first iteration. In this context, recalculating the energy of the whole image would be redundant as most pixels would give the same result as before. Infact, given how the seam is removed, at the next iteration, for every row_index of the seam, only the pixels between (*row_index* − 2) and (*row_index+1, j*)) (where j is the corresponding column index) can change their intensity gradient value. The update function precisely uses this observation to recalculate the energy. The following image represents the pixels of which we have to recalculate the energy.

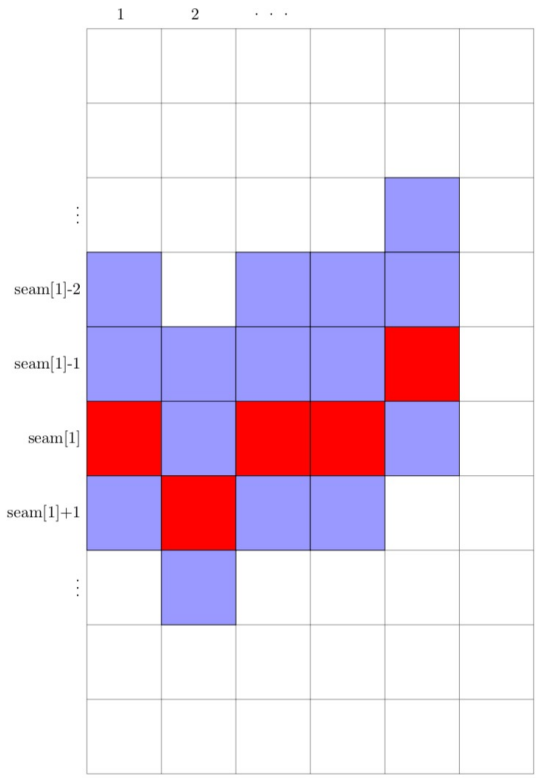

Figure 3.1:

- **update_seam** This function finds the seam after the first iteration. Here one has to be carefull to understand which moves can change their value and which not. Let us start from the beginning of the procedure, from the second column, as the first columns of moves is useless. Here the moves that can change are the ones in which the *aux_energy* of at least one of the elements of which we take the minimum has changed. Now in the update energy explanation we showed that up to four intensity gradients can change in the first column. Hence, in the second column, we are going to have to recalculate the *aux_energy* for the values of the rows corresponding to those four values plus one at the bottom and one at the top. This because, for the top one, when we select the minimum we are going to have that one three candidate values has changed, hence it can be that the *aux_energy* for this value changes also. We can reason anologously for the bottom one. Now, if the *aux_energy* for any of these two external positions remains unchanged from the previous one, then also the

elements coming after in the upper and lower diagonal respectively cannot change. To see this consider, without loss of generality, the upper element in position $(seam[1]-3,2)$. If the *aux_energy* here does not change, then when taking the minimum in the position $(seam[1]-4,3)$, we are going to have the exact same three values to choose from as previously; hence we know that here it is redundant to recalculate the minimum. Once this value cannot change, we can reason in the same way for the element in position $(seam[1]-5,4)$ and so on.
Hence in the updating seam procedure, at every column iteration, we will increase the top/bottom extreme only if the *aux_energy* in such considered extreme has changed from previous one.
One has to be careful that he has to look at change in *aux_energy* and not the *moves* matrices as a permanence in the latter with respect to the previous iteration could still lead to a change in the *aux_energy* value in that position, and hence lead to subsequent changes in the moves matrix down the respective diagonal.

This fact is much easier to grasp graphically:

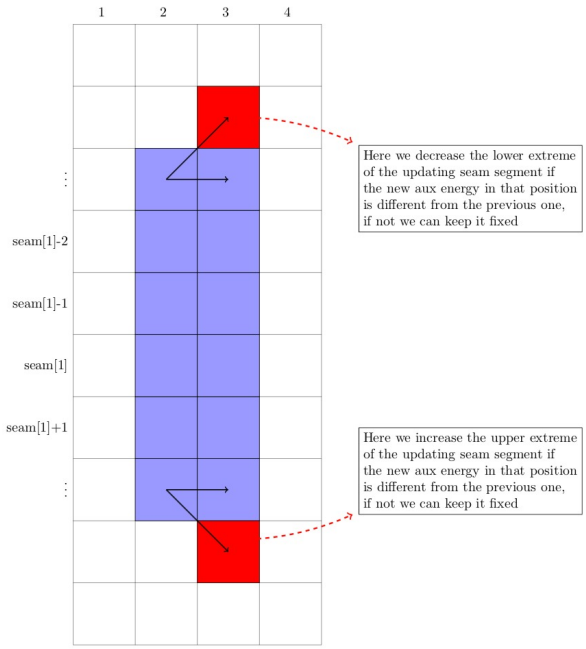

Figure 3.2:

4 Adaptive Seam Carving

In this section we briefly descrive a possible extension of the seam carving procedure. One might want that the algorithm autonomously decides if, at a certain iteration, it is better to remove horizontal seams or vertical ones.
The implementation of this feature is centered around the function *get_direction*, of which we give now a brief description.

- **get_direction**: This function, at a given iteration, performs the first steps on the seam carving procedure both for the image and its' transpose. It then sums the bottom twenty normalized seam costs for both the images, and then assesses which one is smaller. We take the normalized seam costs because we have to account for the different lengths of the horizontal and vertical seams. We choose to sum the bottom twenty because, at the moment, we call this adaptive function once every twenty iterations.

The other functions of this extensions, namely *ada_seam_carve* and *ada_seam_carve_1* have a similar role than their counterparts in the non-adaptive implementation. The former splits the procedure in chunks of size *ada_size*. At the end of every chunk, for the moment, we cut the image and reinitialize the auxiliary structure. In every such chunk, we call latter function which assess if, for the next *ada_size* iterations, it is better to remove horizontal rather then vertical seams. Once it has done this, it removes such *ada_size* seams in the classical way. Notice that regardless of if *get_direction* chooses to remove vertical seams or not, once we remove the seams we always remove horizontally. This because, as we have stated many times, this is more efficient for how arrays are stored in memory in Julia.

We wanted to assess how this adaptive procedure performed on our considered image. We performed the procedure for 180 iterations, and saved the intermediate results as well as the different choices made by our adaptive procedure. As we can see from the image, for the first two chunks, the adaptive procedure actually chooses directly to remove horizontal seams. This makes perfectly sense, as it removes seam that contain only blue sky that have no information. The vertical seams, on the other hand, at a certain point must pass from "sky" to "grass", thus conveying more information. Once all of these seams are terminated, the algorithm decides for the third group of 20 iterations to remove the vertical seams that we removed traditionally. Unfortunately then the adaptive procedure switches back to horizontal seams and seems to make the castle to sink in the grass. This clearly distorts the image, and is not a good resizing of the image. But if we look at the seams that it decides to remove in this setting, we see that they are nothing seams which capture blue sky for the first part, and then more to grass for the second part. There are very similar to the ones we remove vertically in the previous iteration, but this time cause bad results in the reduced image. Hence the criterion that we decided to assess which seems to remove proves to be to vague in this adaptive setting.

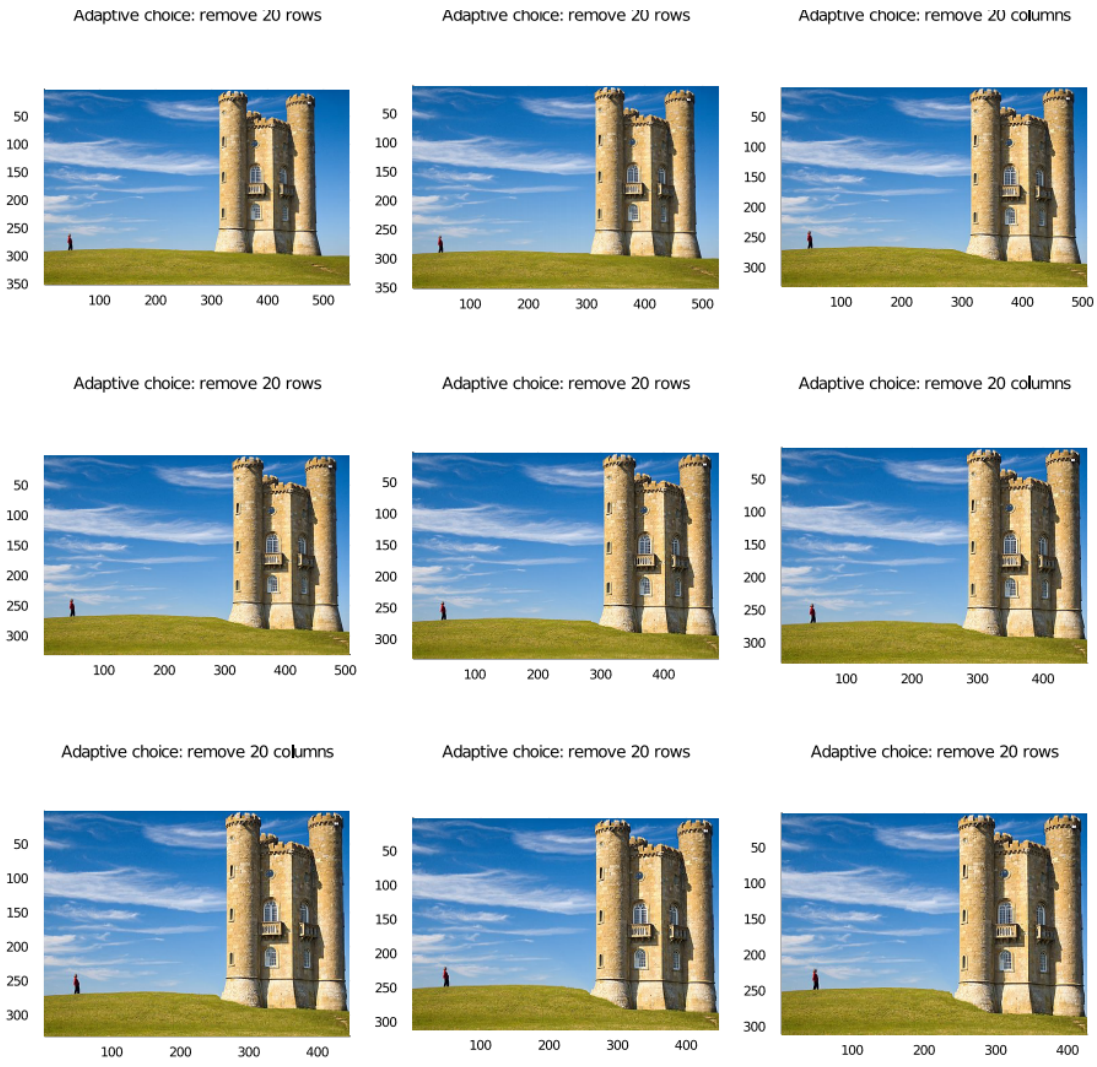
Adaptive choice: remove 20 rows
Adaptive choice: remove 20 rows
Adaptive choice: remove 20 columns
Adaptive choice: remove 20 rows
Adaptive choice: remove 20 rows
Adaptive choice: remove 20 columns
Adaptive choice: remove 20 columns
Adaptive choice: remove 20 rows
Adaptive choice: remove 20 rows

Figure 4.1: