Advanced Topics in Computer Vision and Deep Learning (EEEM071)

Week 3: Digital Image Warping – part 2/2

Dr Xiatian Zhu

Introduction

In this lab you will experiment with digital image warping techniques. You should first have completed the Week 2 lab sheet ("Digital Image Warping, part 1"). During that lab you will have unzipped the labcode **EEEM071_labcode_weeks2-4.zip** into your home space and set the MATLAB path appropriately. If not please do that first.

1. Bilinear Interpolation

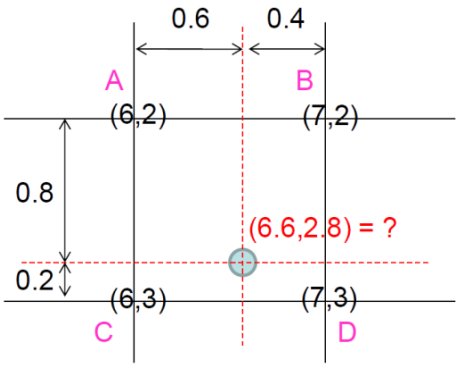
We will start working from the solution to the previous labsheet. Ensure that the 'backward mapping with bounds correction' solution runs correctly by typing:

ipv_warp_backward_with_bounds

Note that the warp is correct but the image quality is not great. The hard edges in the image (e.g. around the stag feet and neck) are quite jagged. This is because the 'interpolation' method used is nearest neighbour. Recall that when the backward warping code runs it is iterating through every pixel in the output image and, for each, calculating the coordinates of the pixel to sample in the input image. When those coordinates are non-integer they are simply rounded (lines 68-69).

We will now do better, by replacing those two lines with several lines that perform bilinear interpolation. You should ensure that you attending/watching both warping lectures in order to make sense of this labseet.

Now open up **ipv_warp_backward_with_bounds** in the editor and save it with a new filename, like **ipv_warp_backward_with_bounds_modified.** Recall the lecture example for bilinear interpolation:



Here the blue dot indicates the position we are trying to sample in the source image, it has coordinates p=(u,v) per lines 61-62 in the code. We cannot directly sample this location in the source image because pixels have only integer coordinates. Therefore we have to figure out the colours at locations A,B,C,D and blend them according to their distances from p.

We can compute A,B,C,D by performing 'ceil' (round up) and 'floor' (round down) operations on (u,v)

```
A=img_in(floor(v),floor(u),:);
B=img_in(ceil(v),floor(u),:);
C=img_in(floor(v),ceil(u),:);
D=img_in(ceil(v),ceil(u),:);
```

You should add these inside the bounds checking if statement in the code (above line 68)

Now below those lines add

```
vertblend=u-floor(u);
horizblend=v-floor(v);
```

Here vertblend tells us how far we are from A to C (and similarly B to D). In other words how much we need to weight toward C when blending A and C together to get the colour at E. (and similarly how much to weight toward D vs. B when blending B and D together to get the colour at F).

So add two lines to do that

```
E=A*(1-vertblend) + C*vertblend;
F=B*(1-vertblend) + D*vertblend;
```

Now finally blend E and F together via horizblend (which tells us how much to weight toward F when blending E and F).

```
G=(1-horizblend)*E + horizblend*F;
```

Here G refers to the colour at (u,v) i.e. the solution. So finally replace the existing line setting the pixel in img_out with a line that simply uses the colour computed for G

```
img_out(y,x,:)=G;
```

Run the code and satisfy yourself that a correct warp has been produced. You should compare the output to that of the original code **ipv_warp_backward_with_bounds.** Hopefully you followed the advice in this sheet and saved your current file with a new filename and didn't directly edit that file – otherwise you can download it again from the labcode zip.

Notice how the edges in the bilinear interpolation and less jagged and the warp is generally of a higher aesthetic quality.

If you did not get the correct result then take a look at the solution pre-supplied for you in ipv_warp_backward_with_bounds_bilinear however you should take the time to work through this sheet and understand why the bilinear interpolation works the way it does.

2. Aliasing

Aliasing occurs when a signal is sampled at too low a sampling rate. Nyquist's theorem tells us that we should sample a signal at a rate at least twice the highest frequency component in it. That is the 'Nyquist limit'. If we don't and the rate is too low, then those frequencies above the Nyquist limit will 'alias' as low frequencies in the sampled signal.

In our case the signal is the source image. When our backward mapping loop samples the source image it is subject to Nyquist's limit.

Ensure the solution to part 1 runs correctly

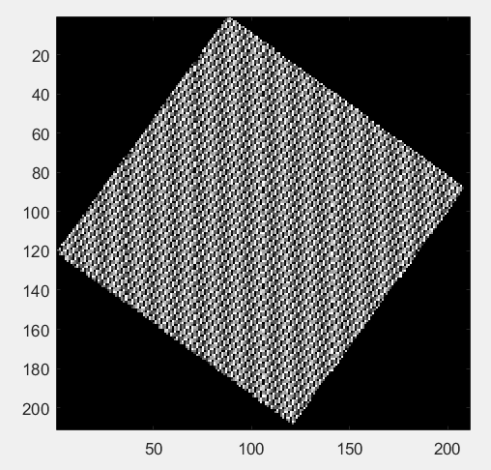
ipv_warp_backward_with_bounds_bilinear

Now uncomment line 22 (which loads the Surrey stag) and uncomment line 23 (which loads up a fine chequerboard pattern) as the input image instead. Note the comment character in MATLAB is %. Run it again.

Now edit line 33 and change 'sf' which refers to scale factor, to 0.3.

Run the code again. It will warp a fine-grain chequerboard pattern by rotating it pi/5 radians and scaling down by a factor of around 3 i.e. scale factor (sf) 0.3

You should see an output resembling:



Note the black vertical wavy lines – these are aliases of the high frequencies in the source image masquerading as low frequencies in the warped output. Why? Because the transformation we performed caused the backward warp loop to sample the source image too sparsely.

The original chequerboard image was generated by the line

```
img_in=ipv_cheqpattern(500,500,3,3);
```

which generated a pattern of squares i.e. step edges every 3 pixels. In order to avoid aliasing we should have sampled the signal at least twice per 3 pixel interval i.e. every 1.5 pixels. However due

to the scaling operation (by factor of 0.3) we will have sampled the source image every approximately 3 pixels – so around half the Nyquist limit. This is why we see aliasing.

We can avoid aliasing by removing high frequencies in the source image. There are two ways to do this:

1) Generate a coarser chequerboard pattern (change the 3 in the ipv_cheqpattern call to a larger number like 10+).

2) Blur i.e. low-pass filter the source image. Try adding the following immediately after generating the chequerboard image. The aliasing will vanish.

```
img_in=imgaussfilt(img_in,1);
```

3. Quantifying performance of interpolation methods

In part 1 of this sheet we implemented bilinear interpolation and visually verified that it performed a higher quality warp on the image of the Surrey Stag.

However this was a subjective judgement – how can we quantify the improvement?

One solution is to warp the image and then perform the inverse warp to recover the original image. Due to the lossy nature of the interpolation process, the image will undergo degradation and the exact original image will not be recovered.

We can therefore calculate the difference between pixel values in the original and the recovered (unwarped) image. One common way to compute this difference ('error') is to subtract pixels in corresponding positions and sum up those differences (errors) squared. If we then divide my the number of pixels in the image, we get the Mean Squared Error or MSE score.

Run the code to perform this process using a nearest neighbour interpolation.

ipv_warp_evaluate

The resulting image shows on the left hand side the original image, and on the right the unwarped image. The MSE score is approximately 0.46.

Now edit the code to comment line 25 and uncomment 26 – this will perform a bilinear interpolation. What is the MSE for bilinear interpolation – by how many times is the interpolation improving the accuracy of the output in terms of MSE?

For the remainder of this lab, try out different images from the internet that contain varying amounts of high frequency detail (e.g. small textures like gravel, hair, brickwork). How does the MSE gain on such content different from warping content containing large expanses of colour like cartoons? Why do you think that there is such a different in the MSE gain of interpolation vs. nearest neighbour across these high and low frequency content classes?

END