Solution 1

(a) If we have two cameras with projection matrices

$$P_1 = K[I|0] \tag{1}$$

$$P_2 = K[I|t] \tag{2}$$

where

$$K = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

$$t = [-t_x, 0, 0]^T \tag{4}$$

we can describe a point in homogeneous coordinates $p' = [X, Y, Z, 1]^T$ in each camera as follows:

$$p_1 = P_1 p' = \left[\frac{fX}{Z}, \frac{fY}{Z}, 1\right]^T \tag{5}$$

$$p_2 = P_2 p' = \left[\frac{fX}{Z} - \frac{ft_x}{Z}, \frac{fY}{Z}, 1\right]^T \tag{6}$$

Then if we let the disparity d be the change in x coordinates between the two cameras,

$$d = x_1 - x_2 = \frac{ft_x}{Z} \tag{7}$$

(b) If we assume the point p' lies on a plane satisfying

$$k = \alpha X + \beta Y + \gamma Z \tag{8}$$

we can solve for a linear equation from the pixel locations of camera 1 relating the disparity and X, Y positions of the point p'. Since equation 7 from part (a) hold for any point p' we can assume it still holds for this case, and we get

$$d = \frac{ft_x}{Z} = ax_1 + by_1 + c \tag{9}$$

But since we know x_1, y_1 from equation 5 and we can multiply by Z, we can rewrite our equation to match the form of our plane in equation 8.

$$ft_x = afX + bfY + cZ \tag{10}$$

We can assume there is some scale difference between the two equations, $\delta = \frac{ft_x}{k}$ we can rewrite our plane as

$$ft_x = \delta\alpha X + \delta\beta Y + \delta\gamma Z \tag{11}$$

and use the equivalency to directly write a, b, c.

$$a = \frac{t_x \alpha}{k} \tag{12}$$

$$b = \frac{t_x \beta}{k} \tag{13}$$

$$c = \frac{ft_x \gamma}{k} \tag{14}$$

(c) If we now consider two cameras with the same form of projection matrices described in equations 1-3, but with $t = [0, 0, -t_z]^T$. We can write a the projection of a point p' (no longer bound to a given plane) as

$$p_1 = P_1 p' = \left[\frac{fX}{Z}, \frac{fY}{Z}, 1\right]^T \tag{15}$$

$$p_2 = P_2 p' = \left[\frac{fX}{Z - t_z}, \frac{fY}{Z - t_z}, 1\right]^T \tag{16}$$

Since the goal of disparity mapping is to find the Z position of an image based on the change of x_1, x_2 and/or y_1, y_2, a point that these two cameras cannot find the distance of is any point where $x_1 = x_2$ and $y_1 = y_2$. The set of points that satisfy this are $X = 0, Y = 0$ with Z as a free variable. This means that this pair of cameras cannot determine the distance of a point directly in front of them. However, They can use points very close to the center, likely a part of the same object, to estimate the distance of the point in the center of their images.

Solution 2

(a) In this problem we build a disparity of a pair of images by building a cost volume and taking the pixel-wise arg min. This method, shown in 1, is pretty good at finding the general outline of objects, but is very noisy.

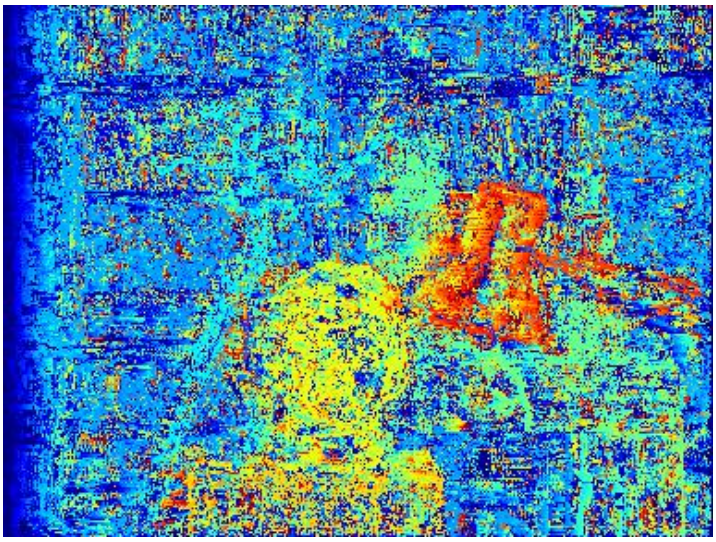

Figure 1: Disparity map from minimizing cost-volume without filtering

(b) To try to eliminate some of the noise, we build a bilateral kernel from the original image to try to preserve outlines of shapes, since we expect objects to have relatively constant disparity. We apply the bilateral kernel to the cost-volume to get the mapping in figure 2. This method is far less noisy, but still is rather splotchy and loses a lot of the fine detail from the original mapping. This makes sense because bilateral filtering blurs details, while retaining edges.

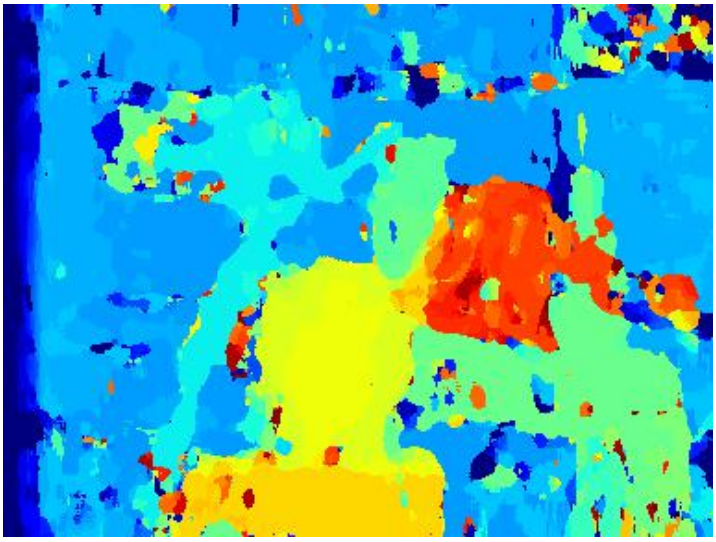

Figure 2: Disparity map from minimizing a bilaterally filtered cost-volume

Solution 3

(a) To implement the Viterbi forward-back algorithm, I used the fact that our smoothness cost function gave only four options for $\arg\min$ of d' for $\tilde{C}[x, d'] + S(d, d')$ to build an options matrix with dimensions $H, D, 4$ where H is the height of the image, D is the depth of the cost volume and 4 comes from the four options. Then, I found the argmin along its third axis to minimize every part of the problem for a given value of x in parallel. Then I just assign $z[x, d]$ as the argmin, and $\bar{C}[x, d]$ as the element of the options matrix that gave the minimum plus $C[x, d]$. The result is shown in Figure 3. There are a lot of streaking artifacts, because each horizontal line was separately minimized.

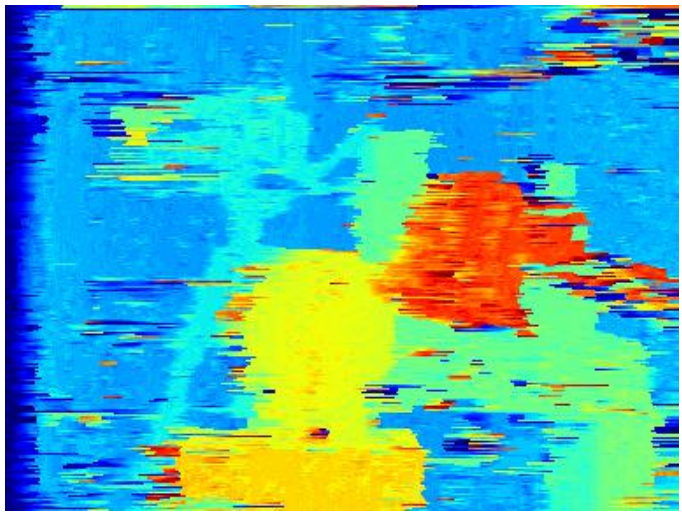

Figure 3: Disparity map from Viterbi left-right algorithm

(b) For this problem I implemented a similar algorithm in SGM. I used the same function to compute the augmented cost volumes as in part (a), without backtracking, just flipped and transposed along the X, Y axes to find augmented cost volumes in left-right, right-left, up-down and down-up directions, then taking the argmin along the D axis. There are less obvious streaking artifacts because the different directions help to smooth them out.

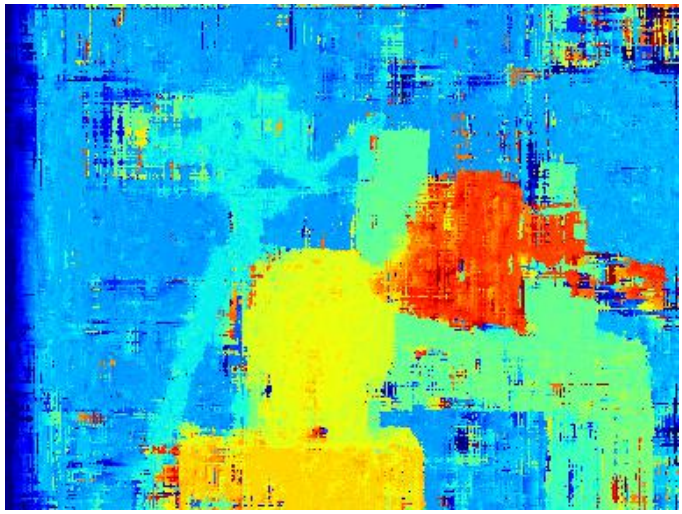

Figure 4: Disparity map from SGM

Solution 4

In this problem we compute an optical flow mapping of a pair of images from consecutive frames from a camera using the Lucas-Kanade Method. A key assumption to this method is that the two images were taken with only a short period of time between them, or that the velocity of objects is slow enough that pixels remain close to where they were located in the previous frame.

To do the computation, we compute a time gradient image, I_t, and $I_{x,y}$ y gradient images for the average of the two frames. The time gradient is simply the second image's intensity subtracting the first image's intensity, and the I_x, I_y gradients are computed using convolution of the average of the two frames' intensities. If we assume that the flow follows the following equation,

$$I_x\left[x', y'\right] u + I_y\left[x', y'\right] v = -I_t\left[x', y'\right] \tag{17}$$

then we can construct a quadratic minimization problem for u, v to solve with least squares:

$$u(x,y), v(x,y) = \arg\min_{u,v} \sum_{(x',y')} \left(I_x\left[x', y'\right] u + I_y\left[x', y'\right] v + I_t\left[x', y'\right]\right)^2 \tag{18}$$

If we take the derivatives in terms of u, v and set them equal to 0, we get

$$\mathbf{0} = \begin{bmatrix} \sum_{(x',y')} 2\left(I_x^2\left[x', y'\right] u + I_x\left[x', y'\right] I_y\left[x', y'\right] v + I_x\left[x', y'\right] I_t\left[x', y'\right]\right) \\ \sum_{(x',y')} 2\left(I_x\left[x', y'\right] I_y\left[x', y'\right] u + I_y^2\left[x', y'\right] v + I_y\left[x', y'\right] I_t\left[x', y'\right]\right) \end{bmatrix} \tag{19}$$

We can then formulate this as a matrix equation to solve in least squares,

$$\begin{bmatrix} \sum I_x^2 + \epsilon & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 + \epsilon \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{20}$$

where the sums are local summations of pixels in a range of $\pm W$ in the x, y directions, and ϵ is a small number added to the diagonal elements to make inverting the matrix more stable. The local summations can be computed for each pixel in parallel using a convolution with a kernel of all ones. To make this solution possible with elementwise operations, we can then find the 2×2 inverse by hand to solve for u, v,

$$u = \frac{\left(\sum I_x I_y\right)\left(\sum I_y I_t\right) - \left(\sum I_y^2\right)\left(\sum I_x I_t\right)}{\left(\sum I_x^2\right)\left(\sum I_y^2\right) - \left(\sum I_x I_y\right)\left(\sum I_x I_y\right)} \tag{21}$$

$$v = \frac{\left(\sum I_x I_y\right)\left(\sum I_x I_t\right) - \left(\sum I_x^2\right)\left(\sum I_y I_t\right)}{\left(\sum I_x^2\right)\left(\sum I_y^2\right) - \left(\sum I_x I_y\right)\left(\sum I_x I_y\right)} \tag{22}$$

The result of the algorithm is shown in figure 5.

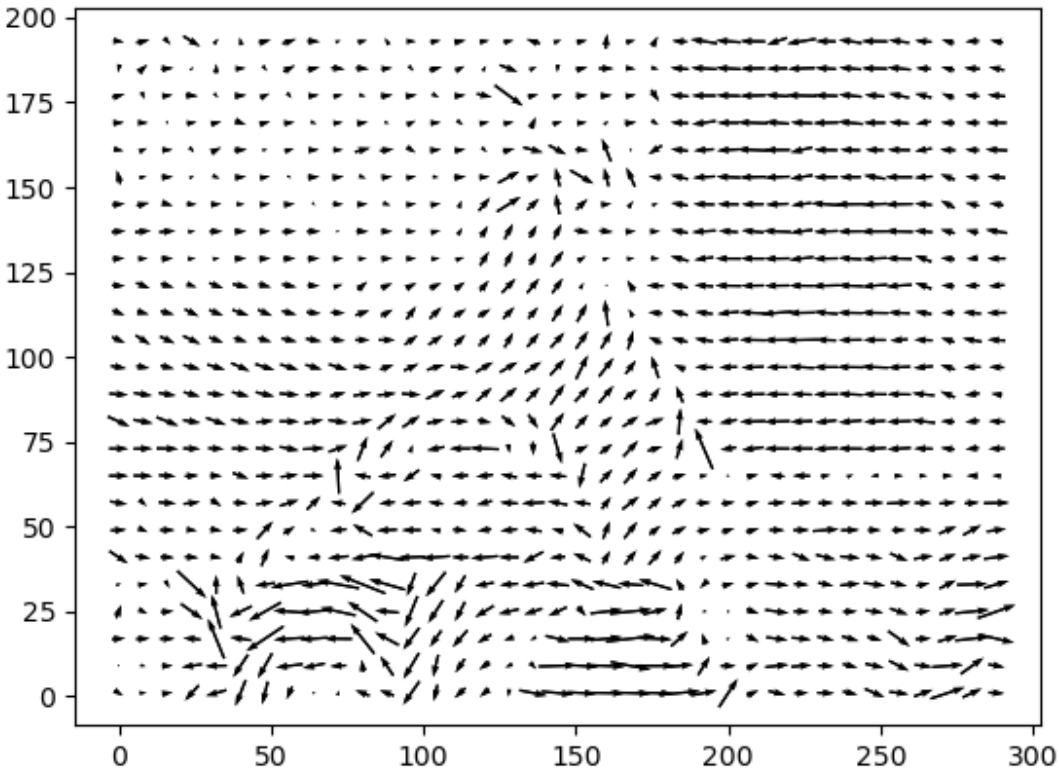

Figure 5: Optical Flow using Lucas-Kanade Method

Information

This problem set took approximately 25 hours of effort (a very large part of which was dedicated to bugfixing when trying to parallelize problem 3a)

I discussed this problem set with no one.

I also got hints from the following sources:

- numpy.where documentation (p3a), https://numpy.org/doc/stable/reference/generated/numpy.where.html
- numpy.transpose documentation (p3b), https://numpy.org/doc/stable/reference/generated/numpy.transpose.html