CS 131 Computer Vision: Foundations and Applications
(Fall 2014)
Problem Set 2

Due Friday, November 14, 5 PM

1 K-Means Clustering

Suppose we want to organize m points $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$ into k clusters. This is equivalent to a label $\ell_1, \ldots, \ell_m$ to each point, where $\ell_i \in \{1, \ldots, k\}$. As discussed in class, the K-Means clustering algorithm solves this problem by minimizing the quantity

$$\sum_{i=1}^{n} d(x^{(i)}, \mu_{\ell_i}), \tag{1}$$

where μ_c is the centroid of all points $x^{(i)}$ with label $\ell_i = c$ and $d(x, y)$ is the distance between points x and y. Usually d is the *Euclidean distance*, given by

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^2 \right)^{1/2}$$

In this problem we will explore the possibility of using other distance functions.

(a) One reason for choosing different distance functions is to control the *shape* of the clusters that are discovered by the K-Means algorithm. K-Means usually tends to find clusters that are roughly *spherical* in shape. A *sphere* with center $c \in \mathbb{R}^n$ and radius $r > 0$ is the set of all points in $\mathbb{R}^n$ whose distance to c is exactly r. For example, when $n = 2$, spheres generated using the usual Euclidean distance function are just circles.

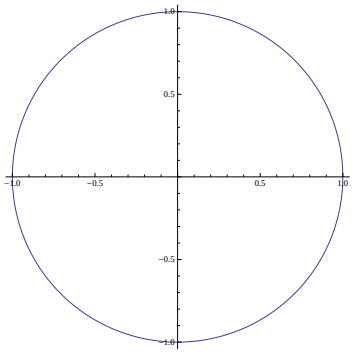

Figure 1: Using Euclidean distance in $\mathbb{R}^2$, spheres are just circles.

For each of the following distance functions on $\mathbb{R}^n$, sketch and describe the spheres that the distance function defines when $n = 2$:

(i) The *Manhattan distance*, also called the L_1 *distance*:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

(ii) The *Chebyshev distance*, also called the L_∞ *distance*:

$$d(x, y) = \max_{i=1,\ldots,n} |x_i - y_i|$$

(iii) The *Cosine distance*:

$$d(x, y) = 1 - \frac{x \cdot y}{|x||y|}$$

, where $x \cdot y$ is the *dot product* given by $x \cdot y = \sum_{i=1}^{n} x_i y_i$ and $|x|$ is the *norm* of x, given by $|x| = \left(\sum_{i=1}^{n} x_i^2\right)^{1/2}$.

(b) Another reason for choosing different distance functions in the K-Means algorithm is to control the *invariances* of the clusters. However, before worrying about invariances of clusters we must think about invariances of distance functions. A distance function d on $\mathbb{R}^n$ is said to be *invariant* under a transformation $\phi : \mathbb{R}^n \to \mathbb{R}^n$ if

$$d(x, y) = d(\phi(x), \phi(y))$$

for all $x, y \in \mathbb{R}^n$.

(i) Consider the case where ϕ is a *scaling* transformation given by $\phi(x) = sx$ for some fixed $s \in \mathbb{R}$ with $s > 0$. Among the Euclidean distance, the Manhattan distance, and the Cosine distance, which are invariant under scaling? For each distance function either prove that it is scale invariant or provide a counterexample to show that it is not.

(ii) Consider the case where ϕ is a *translation* given by $\phi(x) = x + t$ for some fixed $t \in \mathbb{R}^n$. Which of the following are invariant to translation: Euclidean distance, Manhattan distance, Cosine distance? For each distance function either prove that it is invariant under translation or provide a counterexample to show that it is not.

(iii) Let ϕ be a *rotation* given by $\phi(x) = Rx$ where R is a rotation matrix. which of the following are invariant to rotation: Euclidean distance, Manhattan distance, Cosine distance? For each distance function either prove that it is invariant under rotation or provide a counterexample to show that it is not.

(c) The K-Means algorithm with distance function d is said to be *invariant* to a transformation $\phi : \mathbb{R}^n \to \mathbb{R}^n$ if for all sets of points $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$, the cluster assignments that minimize Equation 1 also minimize Equation 1 when each $x^{(i)}$ is replaced by $\phi(x^{(i)})$.

In order to use the K-Means algorithm, we need to know how to compute distances between points and we need to know how to compute the centers of clusters. The distance function d tells us how to compute distances; to compute cluster centers we use a *cluster center function* μ. For any set of points $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$ the center μ_c of these points is given by $\mu_c = \mu(x^{(1)}, \ldots, x^{(2)})$.

A cluster center function μ is *invariant* to a transformation $\phi : \mathbb{R}^n \to \mathbb{R}^n$ if

$$\phi(\mu(x^{(1)}, \ldots, x^{(m)})) = \mu(\phi(x^{(1)}), \ldots, \phi(x^{(m)}))$$

for all sets of points $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$.

(i) Consider the cluster center function

$$\mu(x^{(1)}, \ldots, x^{(m)}) = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{2}$$

Is K-Means with Euclidean distance and the above cluster center function invariant under scaling transformations? If yes, provide a proof; if not, provide a counterexample.

(ii) Consider the cluster center function μ given by

$$\mu(x^{(1)}, \ldots, x^{(m)})_i = median(x_i^{(1)}, \ldots, x_i^{(m)}) \tag{3}$$

In other words, the ith coordinate of the centroid is the median of the ith coordinates of the points $x^{(1)}, \ldots, x^{(m)}$. Is K-Means with Manhattan distance and the above cluster center function invariant under scaling transformations? If yes, provide a proof; if not, provide a counterexample.

(d) Suppose that we want to compute a segmentation for an image I by using K-Means to cluster the color values of its pixels. In other words, for each pixel we will create a vector (r, g, b) of its color values and use K-Means to cluster these color vectors. Each cluster will correspond to a segment in the image; all pixels whose color vectors end up in the same cluster will be assigned to the same segment.

(a) We create a high-contrast image I' by doubling the color values of each pixel in I. Which of the following distance function(s) can we use to ensure that the segmentations for I and I' are the same: Euclidean distance, Manhattan distance, Chebyshev distance, Cosine distance? For each of these distance functions give a short explanation as to why it does or does not produce the same segmentations for images I and I'. For the Manhattan distance, assume that we use the cluster center function from Equation 3; for all other distance functions, assume that we use the cluster center function from Equation 2.

(b) We create a tinted image I' by increasing the red value of each pixel in I by a fixed amount r. Which of the following distance function(s) can we use to ensure that the segmentations for I and I' are the same: Euclidean distance, Manhattan distance, Chebyshev distance, Cosine distance? For each of these distance functions give a short explanation as to why it does or does not produce the same segmentations for images I and I'. For the Manhattan distance, assume that we use the cluster center function from Equation 3; for all other distance functions, assume that we use the cluster center function from Equation 2.

2 Frame Interpolation using Optical Flow

Suppose we are given a pair of frames I_0 and I_1 from a video sequence which occur at times $t = 0$ and $t = 1$, respectively. We wish to generate additional interpolated frames I_t for $0 < t < 1$. This is a common problem in video processing; for example, if we wish to display a video with 24 frames per second on a television with a refresh rate of 60 Hz, then interpolating the frames of the input video can result in smoother playback.

The code and data for this problem are located in the `FrameInterpolationSkeleton` directory. Start by looking at the file `FrameInterpolation.m`; this file sets up the frame interpolation task for you by loading a pair of images and computing the optical flow between them.

NOTE: In the equations below we use Cartesian coordinates (x, y) but the MATLAB code for this problem uses matrix coordinates $(row, column)$. Keep this in mind when implementing your solution.

(a) The simplest way of generating interpolated frames is to use *linear interpolation*, also known as *cross-fading*. In this model, the interpolated images I_t are given by

$$I_t(x, y) = (1 - t)I_0(x, y) + tI_1(x, y)$$

Implement this method in the file `ComputeCrossFadeFrame.m`. After implementing this method, you can view the interpolated video sequence by uncommenting the indicated lines in `FrameInterpolation.m`. Hand in your completed version of the file `ComputeCrossFadeFrames.m` and a printout of the tiled image frames which are written to the file `crossfade.png`.

(b) More sophisticated techniques for frame interpolation take the appearance of the input images into account. One of the most natural ways to incorporate this information is to use optical flow fields. In this problem the horizontal and vertical components of the velocity of the point $I_t(x, y)$ will be denoted $u_t(x, y)$ and $v_t(x, y)$ respectively. As discussed in class, we can compute the optical flow between the images I_0 and I_1 to obtain the velocities $(u_0(x, y), v_0(x, y))$ of every point of I_0.

After computing the optical flow, a simple strategy for computing interpolated frames is to carry the pixels of I_0 forward along their velocity vectors; this algorithm is known as *forward warping*. More concretely, the interpolated images are given by

$$I_t(x + tu_0(x, y), y + tv_0(x, y)) = I_0(x, y)$$

Implement this method in the file `ComputeForwardWarpingFrame.m`. After implementing this method you can can view the interpolated video by uncommenting the indicated lines in `FrameInterpolation.m`. Hand in your completed version of the file `ComputeForwardWarpingFrame.m` and a printout of the tiled image frames which are written to the file `forwardwarped.png`.

(c) Forward warping gives much better results than cross-fading, but it can still lead to significant artifacts in the interpolated images. Give at least one explanation for the types of artifacts that we see when using forward warping to interpolate images.

(d) A more robust strategy is to use *backward warping*. If we knew the velocities $(u_t(x, y), v_t(x, y))$ for all of the points in I_t, then we could compute the pixel values of I_t by setting

$$I_t(x, y) = I_0(x - tu_t(x, y), y - tv_t(x, y))$$

Unfortunately we do not know the velocities at time t. However, we can estimate the velocities at time t by forward-warping the known velocities u_0 and v_0 at time 0. We can increase the robustness of this estimate by using a few heuristics.

For each point (x, y) of I_0, we compute the quantities $x' = x + tu_0(x, y)$ and $y' = y + tv_0(x, y)$. Then for all pairs (x'', y'') with $x'' \in \{floor(x'), ceil(x')\}$ and $y'' \in \{floor(y'), ceil(y')\}$ we set

$$u_t(x'', y'') = u_0(x, y)$$
$$v_t(x'', y'') = v_0(x, y)$$

This can help account for small inaccuracies in the computed optical flow. If this would cause multiple points (x, y) of (u_0, v_0) to be assigned to the point (x'', y'') of (u_t, v_t), then pick the one that minimizes the color difference

$$|I_0(x, y) - I_1(x + u_0(x, y), y + v_0(x, y))|$$

The quantity $I_1(x + u_0(x, y), y + v_0(x, y))$ is appearance of the pixel to which (x, y) would be sent if we forward warped all the way to time 1. The idea behind this heuristic is that if two or more pixels at time 0 collide when forward warped to time t, then we keep only the pixel whose appearance changes the least when forward warped all the way to time 1; hopefully this is the pixel that also changes the least when forward warped to time t.

Additionally, if any points of the interpolated flow (u_t, v_t) remain unassigned after this procedure, we use linear interpolation to fill in the gaps. The function `FillInHoles` can be used for this step. Implement forward warping of the flow field as described above in the function `WarpFlow` of the file `ComputeFlowWarpFrame.m`.

(e) Once we have estimated the velocities (u_t, v_t) we can use backward warping to compute the interpolated image I_t as described above. Implement backward warping in the function `ComputeFlowWarpFrame` of the file `ComputeFlowWarpFrame.m`. After implementing this method you can view the interpolated video by uncommenting the indicated lines in `FrameInterpolation.m`. Hand in the completed version of the file `ComputeFlowWarpFrame.m` and a printout of the tiled image frames which are written to the file `flowwarped.png`.

3 Motion Segmentation

The problem of grouping pixels of an image into regions that each move in a coherent fashion is called *motion segmentation*.

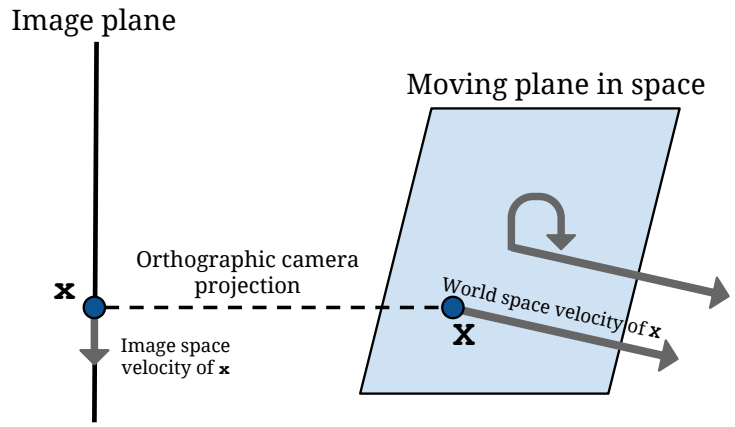

Figure 2: Problem 3a: A point on a moving plane is projected onto the image plane.

(a) Consider a plane in 3D space given by

$$Z = c_x X + c_y Y + c_0.$$

Suppose that this plane is undergoing rigid body motion and that it is being viewed through an orthographic camera as in Figure 2. Show that the velocity $\mathbf{v} = (v_x, v_y)$ of the point $\mathbf{x} = (x, y)$ in image space corresponding to the point $\mathbf{X} = (X, Y, Z)$ on the plane in world space can be written in the form

$$\begin{aligned} v_x &= a_{x,0} + a_{x,x}x + a_{x,y}y \\ v_y &= a_{y,0} + a_{y,x}x + a_{y,y}y, \end{aligned}$$

where the coefficients $a_{x,0}, a_{x,x}, a_{x,y}, a_{y,0}, a_{y,x}$, and $a_{y,y}$ do not depend on x or y. In other words, show that the velocity of all image space points is an affine function of their positions; this is called an *affine motion model*.

HINT: Rigid body motion can be decomposed into a rotation about each axis and a translation. If the plane is rotating with angular velocities $\omega = (\omega_X, \omega_Y, \omega_Z)$ about each axis and is translating with velocity $T = (T_X, T_Y, T_Z)$ along each axis then the velocity of the point $\mathbf{X} = (X, Y, Z)$ is given by

$$\mathbf{V} = \Omega\mathbf{X} + T$$

where

$$\Omega = \begin{bmatrix} 0 & -\omega_Z & \omega_Y \\ \omega_Z & 0 & -\omega_X \\ -\omega_Y & \omega_X & 0 \end{bmatrix}$$

An *orthographic camera* is a simplified camera model whose camera matrix is simply

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In other words, the orthographic camera converts points from world space to image space by simply ignoring the z-coordinate.

(b) If we make the simplifying assumptions that the world is composed of planes undergoing rigid motion and that the camera performs orthographic projection, then by (a) the problem of motion segmentation is reduced to the problem of determining a set of affine motion models for an image and assigning each pixel of the image to one of the motion models.

We can use an optical flow algorithm to estimate the velocity of each point in an image. If the world really obeyed our simplified model and if the optical flow algorithm were perfect, then we could trivially deduce a set of affine motion models using only the optical flow. Since neither of these conditions are likely to hold we must perform some processing on the optical flow in order to estimate the affine motion models.

Given a small group of nearby pixels, it is likely that they belong to the same motion model. Therefore, to generate a set of candidate motion models, we can divide the image into many small regions and use linear regression to fit an affine motion model in each region.

For this problem you may assume that the least squares solution to the problem $Ax = b$ is given by $x = (A^T A)^{-1} A^T b$.

(i) Explain how we can use linear regression to fit an affine motion model to a set of points given their positions and velocities.

(ii) If the entire image is rescaled by a factor $s > 0$ then how do the estimated affine motions models in each region change? You may assume that scaling the image by a factor of s also scales the optical flow of the image by a factor of s.

(iii) If the entire image is rotated, then how do the estimated affine models in each region change? You may assume that rotating the image also rotates the optical flow.

(c) Since we generate one candidate motion model for each small image region, it is likely that the number of candidate motion models is much larger than the number of "true" motion models present in the scene. For example, large objects which could be described by a single motion model may have been split up across many small image patches. In a few sentences, briefly explain how you might use the large set of candidate motion models to generate a smaller set of motion models that more accurately describe the motion present in the scene.

HINT: Think of a motion model as a point in $\mathbb{R}^6$.

4 K-Means Local Minima

Consider the points:

$$x_1 = (0, 16), \quad x_2 = (0, 9), \quad x_3 = (-4, 0), \quad x_4 = (4, 0).$$

Suppose we wish to separate these points into two clusters and we initialize the K-Means algorithm by randomly assigning points to clusters. This random initialization assigns x_1 to cluster 1 and all other points to cluster 2.

(a) With these points and these initial assignments of points to clusters, what will be the final assignment of points to clusters computed by K-Means? Explicitly write out the steps that the K-Means algorithm will take to find the final cluster assignments.

(b) In this case, does K-Means find the assignment of points to clusters that minimizes Equation 1?

5 Affine and Projective Cameras

Recall from lecture that we can transform a point in world space $\mathbf{X} = (X, Y, Z)$ to image coordinates $\mathbf{x} = (x, y)$ by using a 3×4 *camera matrix*. In class we showed how a camera matrix can be factored into a 3×3 matrix of *intrinsic parameters* and a 3×4 matrix of *extrinsic parameters*. In this problem, however, we will consider the case of a single 3×4 camera matrix A which combines both the intrinsic and extrinsic parameters of the camera:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ t' \end{bmatrix} \qquad x = x'/t' \qquad y = y'/t'$$

(a) How many degrees of freedom does the above camera matrix have for expressing transformations from (X, Y, Z) to (x, y)? Explain. How does this compare with the number of degrees of freedom when the camera matrix is factored into a 3×3 matrix of intrinsic parameters and a 3×4 matrix of extrinsic parameters?

(b) In the special case of an *affine camera*, the camera matrix has the following form:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ 0 & 0 & 0 & A_{34} \end{bmatrix}$$

An important property of an affine camera is that it preserves parallel lines. We usually represent a line L in $\mathbb{R}^n$ by choosing a point $p \in \mathbb{R}^n$ on the line and picking some vector $v \in \mathbb{R}^n$ parallel to the line. Every point ℓ on the line can then be written in the form $\ell = p + tv$ for some $t \in \mathbb{R}$. For brevity, we often write this as a set of points: $L = \{p + tv : t \in \mathbb{R}\}$.

Suppose that we have a pair of lines in $\mathbb{R}^n$ given by $L_1 = \{p_1 + tv_1 : t \in \mathbb{R}\}$ and $L_2 = \{p_2 + tv_2 : t \in \mathbb{R}\}$. These lines are *parallel* if the vectors v_1 and v_2 are scalar multiples of each other. Now consider a pair of parallel lines in 3D space given by $L_1 = \{p_1 + tv : t \in \mathbb{R}\}$ and $L_2 = \{p_2 + tv : t \in \mathbb{R}\}$. If the lines L_1 and L_2 are transformed from world space to image space using an affine camera matrix, prove that the resulting lines in image space remain parallel.