COMP3005/Computer Vision

8

Matching consistency

Summary

When comparing two images by matching local feature, we need to eliminate mismatches. By applying geometric constraints to the problem of finding corresponding interest points between pairs of images, it is possible to both reduce the number of mismatches and potentially learn a geometric mapping between the two images. Amongst many other applications, these transforms can be used to build panoramas from multiple images. The presence of such a transform is a good indicator for a match between the to images, and is commonly used in object recognition and image retrieval applications.

Key points

Applying constraints to matching

- Even the most advanced local feature can be prone to being mismatched.
    - There is always a tradeoff in feature distinctiveness.
        - If it's too distinctive will not match subtle variations due to noise of imaging conditions.
        - If it's not distinctive enough it will match everything.
- Given a number of correspondences between the interest points in a pair of images, it is potentially possible to estimate (under a number of assumptions!) which of those correspondences are *inliers* (i.e. correct) or *outliers* (i.e. incorrect/mismatches).

Geometric mappings

- One of the simplest possible ways of constraining a set of matches between an object depicted in a pair of images is to assume that the object is largely flat (it could be actually be a flat surface, or it could just be really far away and not change so much in appearance between possible views). With this assumption in mind, it is possible to search for a geometric mapping that is satisfied by the *correctly* corresponding points.
- In general, a geometric mapping can be thought of as a transform function that maps the x, y coordinates of points in one image to another.
    - We're specifically going to consider transforms that can be written in matrix form as follows: $\vec{y} = \mathbf{T}\vec{x}$, where $\mathbf{T}$ is the transform matrix and $\vec{x}$ and $\vec{y}$ are column vectors representing the original and transformed coordinates respectively.
    - For the 2D case, the matrix $\mathbf{T}$ will be 3x3 (see below for why).
- We're going to focus on two specific transforms: the **Affine transform** and the **Planar Homography**:
    - An **Affine transform** is a type of transform that allows for (a combination of) translation, scaling, aspect ration, rotation and skewing. Affine transforms always **preserve parallel lines** however.
        - The standard way of writing a 2D affine transform is as follows: $\vec{y} = \mathbf{A}\vec{x} + \vec{b}$, where $\mathbf{A}$ is a 2x2 transform matrix which encodes scale, rotation and skew, and $\vec{b}$ is a vector encoding the translation.
        - We can rewrite this as a single matrix multiplication by adding an extra dimension to the vectors with a value fixed at 1 as follows:
        $$\begin{bmatrix}\vec{y}\\1\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \vec{b}\\0\ldots0 & 1\end{bmatrix}\begin{bmatrix}\vec{x}\\1\end{bmatrix}$$

        - So, in the case of a 2D affine, transform, the transform matrix will be 3x3:

$$\mathbf{T} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix}$$

        - A 2D affine transform is said to have **6 degrees of freedom**.
    - Because the affine transform preserves parallel lines, it doesn't allow perspective effects such as objects getting smaller as they move further away.
        - The 2D projective transform or (Planar) Homography takes the following form:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, x' = \frac{u}{w}, y' = \frac{v}{w}$$

        - The normalization by *w* is required because the transformation is actually non-linear.
            - The vector $[wx', wy', w]^T$ is called a **homogeneous coordinate**.
            - Homogeneous coordinates allow us to deal with the transforms as a matrix.
        - Values *a-f* in the Homography are the affine parameters (they just define an affine transform assuming *g* and *h* are zero). The *g* and *h* parameters define keystone distortions, which make lines that were originally parallel come together at one end in the transformed space.
- Recovering the transform matrix **T** is a matter of solving a set of homogeneous linear equations with a number of pairs of matching points (at least 4 pairs for the Homography, and 3 for affine) – the detailed specifics of how this is done are not important for this course, but it is useful to appreciate the concept of least-squares estimation:
    - The actual numeric method for estimating the transform must be able to withstand some error in the positions of the matched points.
    - It is common to compute the *least-squares* estimate of the transform matrix.
        - The least-squares estimate minimizes the sum-squared error in the prediction.
            - The error of a single point is often called the **residual**, and is the difference (or distance) between the predicted value and the observed value.

Robust estimation

- Least-squares estimation of a transform matrix (or any other model) has a big problem in the presence of noise.
    - For example, in computing the transform matrix, mismatches can drastically throw off the estimated transform.
- This brings us back to the original problem: how can we determine the inliers?
    - Using only the inliers we could generate a much better transform estimate!
- There are a number of **robust estimation** algorithms that can be applied to robustly estimate the inliers under the assumption that the inliers fit some model with a given accuracy.
    - We're going to look at an algorithm called Random Sample Consensus, more commonly known as **RANSAC**.
    - The RANSAC algorithm is as follows:

```
Assume:
M data items required to estimate model T
N data items in total

Algorithm:
1.) select M data items at random
2.) estimate model T
3.) find how many of the N data items fit T within tolerance tol, call
this K (i.e. compute how many times the absolute residual is less than
tol). The points that have an absolute residual less than tol are the
inliers; the other points are the outliers.
```

```
4.) if K is large enough, either accept T, or compute the least-squares
estimate using all inliers, and exit with success.
5.) repeat steps 1..4 nIterations times
6.) fail - no good T fit of data
```

- In terms of matching points between two images, RANSAC picks some pairs of points randomly and estimates the transform (i.e. affine or Homography), and then measures how many of the remaining point pairs fit that transform within some tolerance; all the pairs that fit are the inliers. If there are enough inliers (usually in proportion to the total number of pairs) then the algorithm stops, and the transform is re-estimated using least-squares with all the inliers. Otherwise, the algorithm goes back to the start, and picks a new random set of pairs.

Improving matching speed

- The biggest problem with the interest-point descriptor matching techniques we've discussed is *speed*.
  - Brute-force matching of 128-dimensional SIFT vectors takes a long time
    - Any given image (say of 800x600 pixels) might have ~2000 interest points
    - If we want to compare an images SIFT features against a large set of other images' features, it's going to take a very long time!
  - How can we speed it up? There are a number of possible techniques for speeding-up nearest-neighbour search in high dimensional spaces:
    - Tree structures: K-D Trees
      - A k-d tree is a binary tree structure
      - Each node splits a specific dimension of the space in two
      - The leaf nodes store a number of points corresponding to the points that have made it down the tree to that point
      - Fast nearest-neighbour search can be achieved by walking down the tree until a leaf is hit.
        - Brute force search can be used to select the neighbour from the points in the leaf; unfortunately, this isn't guaranteed to actually be the closest, so you have to back-track up the tree looking in down the other paths to different leaf nodes until you can be assured that you've checked all the leaves that can possibly contain the neighbour (this is still just a small subset of all the leaves in the tree)
    - Hashing and sketching
      - Some recently introduced techniques allow feature vectors to be hashed with special hashing schemes that allow vectors that are spatially similar (i.e. in the Euclidean sense) to have similar hash codes!
        - These are called Locality Sensitive Hashing Functions
      - A common technique called **sketching** produces binary string encodings for features that allows them to be compared efficiently
        - Often produced so that the Hamming distance between two features encoded as binary strings is proportional to the Euclidean distance between the two original vectors!
          - Sketches are much smaller than the original features (typically for a SIFT feature of 128 single byte elements, the sketch might only be 128 bits)
          - Hamming distance can be efficiently computed using lookup tables and bitwise operations (much cheaper than computing the Euclidean distance in the original space)
    - Vector quantisation and indexing – more on this next time

Further reading

- The original RANSAC paper: http://www.ai.sri.com/pubs/files/836.pdf
- Wikipedia page on k-d trees and locality sensitive hashing are quite good:
  - http://en.wikipedia.org/wiki/K-d_tree
  - http://en.wikipedia.org/wiki/Locality-sensitive_hashing

Practical exercises

- Create some 2D data points and try using the OpenIMAJ `org.openimaj.math.model.LeastSquaresLinearModel` class to fit a straight line. Then adapt your code to use the `org.openimaj.math.model.fit.RANSAC` class to try fitting your model robustly.
- If you haven't already tried it, Chapter 5 of the OpenIMAJ tutorial covers fitting an affine transform to a set of point correspondences using RANSAC.
  - Can you modify the code to fit a Homography instead?