Feature Detection

COSC450

Lecture 3

Feature Detection, Tracking, and Matching

Natural feature tracking

- Avoids need for targets like checkerboards or markers
- Feature detection – corners and blobs
- Feature description – SIFT and related methods
- Tracking – KLT tracker for corners
- Matching – Bag-of-Words
- Application – Image Mosaicing

Corners and Tracking

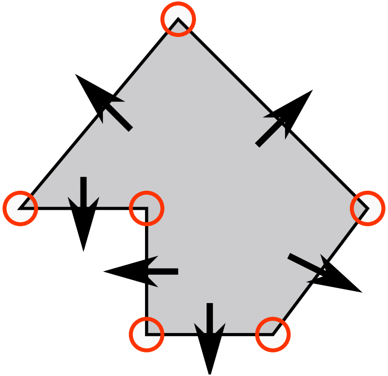
Edges and Corners

Want points that can be accurately located

- Often called *keypoints* or *features*
- Image *corners* can be used
- Formalised using image gradients
  - No gradient – flat region
  - Gradient in one direction – edge
  - Gradient in all directions – corner

How do we compute this?

- Need to estimate image gradients
- Need to find corners



Convolution and Filtering

Convolution common in image processing

- Uses a filter or *kernel*
- Input is image + kernel
- Output is a new image
- Kernel is a small array of numbers

For a kernel, K, of size $((2r+1) \times (2r+1))$

5	6	4	3	4	2
4	8	3	5	6	4
7	7	4	6	8	6
6	9	3	7	9	3
5	2	4	5	6	5
3	1	3	5	8	7

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$[I * K](x, y) = \sum_{d_x=-r}^{r} \sum_{d_y=-r}^{r} I(x+d_x, y+d_y) K(r+d_x, r+d_y)$$

$$[I * K](4, 3) = \frac{1}{16}(7+8+6+18+12+14+2+8+5) = 5$$

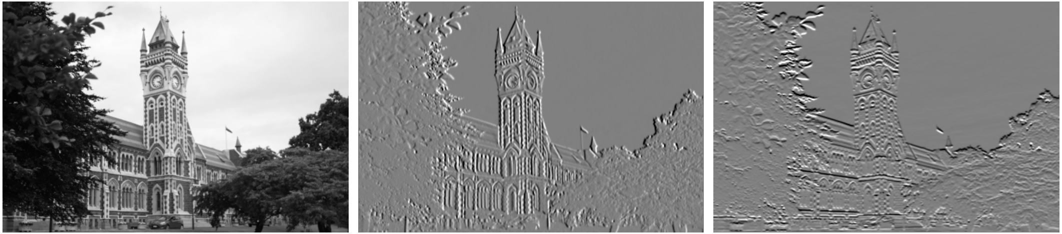
Edge Detection Filters

Sobel filters

- Commonly used for edge detection
- Two filters – compute gradients
- Horizontal (I_x) and vertical (I_y)
- Gradient is a vector, $\mathbf{g} = \begin{bmatrix} I_x & I_y \end{bmatrix}^{\mathsf{T}}$

$$I_x = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$$I_y = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$



Shi-Tomasi Corners

Shift image from (x, y) to $(x + d_x, y + d_y)$

- Make a linerar approximation near (x, y)
- Need gradients to do this (Sobel)
- Look at a region (R) around (x, y)
- Arithmetic gives a *structure matrix*
- Tells us about local area of image
- Corner: high change for all (d_x, d_y)

$$
\begin{aligned}
&\sum_{(x,y)\in R} \left(I(x + d_x, y + d_y) - I(x, y)\right)^2 \\
&\approx \sum \left(I(x, y) + d_x \frac{\partial I}{\partial x} + d_y \frac{\partial I}{\partial y} - I(x, y) \right)^2 \\
&= \sum \left(d_x^2 \left(\frac{\partial I}{\partial x}\right)^2 + 2 d_x d_y \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) + d_y^2 \left(\frac{\partial I}{\partial y}\right)^2 \right) \\
&= d_x^2 \sum \left(\frac{\partial I}{\partial x}\right)^2 + 2 d_x d_y \sum \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) + d_y^2 \sum \left(\frac{\partial I}{\partial y}\right)^2 \\
&= \begin{bmatrix} d_x & d_y \end{bmatrix} \begin{bmatrix} \sum \left(\frac{\partial I}{\partial x}\right)^2 & \sum \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \\ \sum \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) & \sum \left(\frac{\partial I}{\partial y}\right)^2 \end{bmatrix} \begin{bmatrix} d_x \\ d_y \end{bmatrix}
\end{aligned}
$$

Eigenvectors and Eigenvalues

We want high gradient in all directions

- Structure matrix tells us about change in each direction
- Eigenvalues tell us about how a matrix transforms space
- Defined as vectors where $\mathrm{M}\mathbf{v} = \lambda\mathbf{v}$
- $\mathbf{v}$ is an *eigenvector* of M and λ the corresponding *eigenvalue*

Our matrix is 2×2 real symmetric

- So it has 2 eigenvalues/vectors
- Compute *characteristic polynomial*

$$\det(\mathrm{M} - \lambda\mathrm{I}) = 0$$

- Large eigenvalues = large variation
- Want smaller eigenvalue to be large

Eigenvalues for Shi-Tomasi Corners

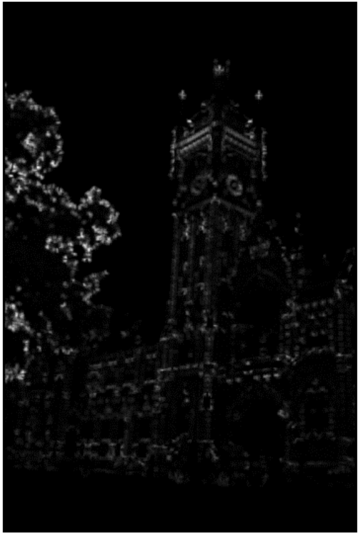
To save space when writing equations we substitute

$$X = \sum \left(\frac{\partial I}{\partial x}\right)^2 \qquad Y = \sum \left(\frac{\partial I}{\partial y}\right)^2 \qquad Z = \sum \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right)$$

so we have

$$\begin{aligned}
\det(\mathrm{M} - \lambda \mathrm{I}) &= 0 \\
\det \begin{bmatrix} X - \lambda & Z \\ Z & Y - \lambda \end{bmatrix} &= 0 \\
(X - \lambda)(Y - \lambda) - Z^2 &= 0 \\
\lambda^2 - \lambda(X + Y) + (XY - Z^2) &= 0 \\
\lambda &= \frac{(X + Y) \pm \sqrt{(X + Y)^2 - 4(XY - Z^2)}}{2}
\end{aligned}$$

Shi-Tomasi Corner Example



KLT Tracking

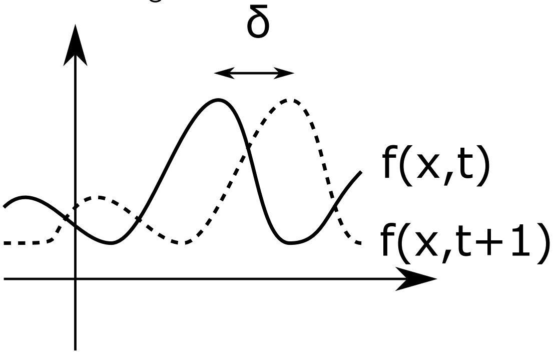
Kanade-Lucas-Tomasi

- Begin with Shi-Tomasi corners
- Suppose we have image sequence $I(x, y, t)$
- Want to find (u, v) so that

$$I(x, y, t) = I(x + u, y + v, t + 1)$$

in some region around the corner

Easier to begin with a 1D case



$$f(x, t) = f(x + \delta, t + 1)$$

1D KLT Formulation

We can try to align a single point using a linear approximation

$$\begin{aligned} 0 &= f(x+\delta, t) - f(x, t+1) \\ &\approx f(x,t) + \frac{\partial f}{\partial x}\delta - f(x, t+1) \\ \delta &\approx \frac{f(x,t+1) - f(x,t)}{\frac{\partial f}{\partial x}} \end{aligned}$$

Over a region, R, we add the squared errors,

$$E = \sum (f(x+\delta, t) - f(x, t+1))^2$$

To minimise, set $\frac{dE}{d\delta} = 0$, giving

$$\delta = \frac{\sum(f(x, t+1) - f(x,t))}{\sum(f'(x,t))}$$

2D KLT Formulation

We consider images, $I(x, y, t)$, over time and we want to minimise

$$E = \sum_{(x,y)\in R} \left(I(x+u, y+v, t) - I(x, y, t+1)\right)^2$$

This leads to the solution:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \left(\sum_{(x,y)\in R} \begin{bmatrix} \left(\frac{\partial I}{\partial x}\right)^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \left(\frac{\partial I}{\partial y}\right)^2 \end{bmatrix} \right)^{-1} \sum_{(x,y)\in R} \begin{bmatrix} \frac{\partial I}{\partial x}\left(I(x, y, t) - I(x, y, t+1)\right) \\ \frac{\partial I}{\partial y}\left(I(x, y, t) - I(x, y, t+1)\right) \end{bmatrix}$$

Note that the matrix we are inverting is the same as in Shi-Tomasi corner detection

Improving tracking

These methods make some assumptions

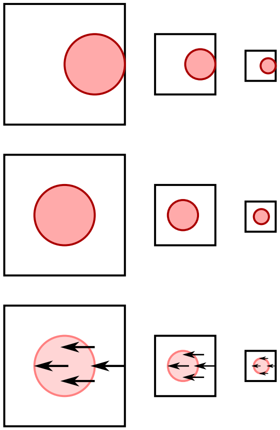
- They work for small motions
- Linear approximations

Pyramid-based optical flow

- Subsample image by half repeatedly
- Compute motion at lowest level
- Double motion to go up one level
- Refine estimate, and repeat

Can also expand on the motion model

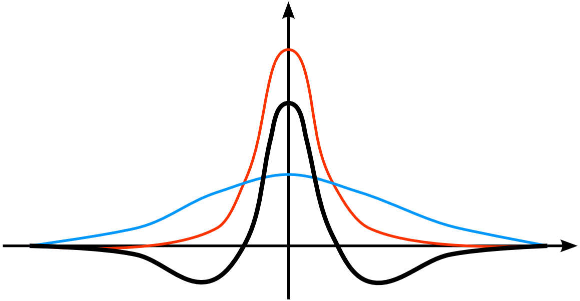
- Affine motion – rotation, scaling, etc.



Blobs and Matching

Blob Features

- More recently, blob features have seen a lot of use
- Blobs are dark regions surrounded by bright regions or vice-versa
- We can find blobs with a *difference of Gaussian* filter



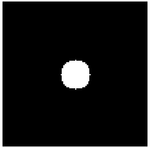
- Blobs have a scale, determined by the variances of the two Gaussians

Blob Detection

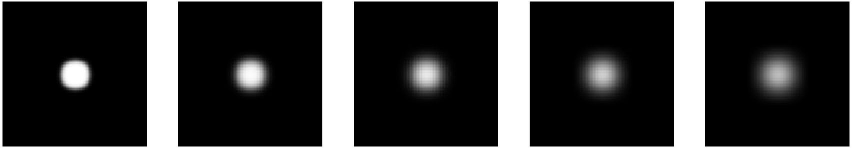
- Blur the image with larger and larger Gaussian kernels
  - You can do this by repeatedly blurring with a small Gaussian kernel
  - For efficiency the image can be halved after every k blurs
- Subtract the adjacent images in the stack from one another
- Blobs are minima and maxima in the stack of difference images
  - Must be locally minimal/maximal in the current difference image
  - Must also be minimal/maximal compared to the two adjacent images

Blob Detection

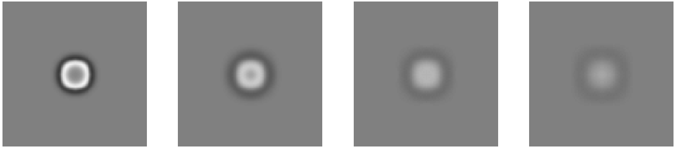
Original Image



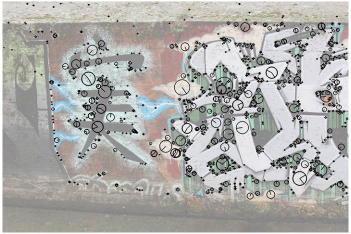
Gaussian blur with $\sigma = 1.5, 3, 4.5, 6, 7.5$



Difference of Gaussians



Corners and Blobs

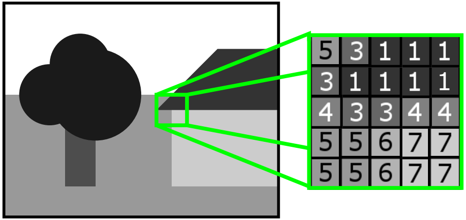


Feature Descriptors

- Features are matched on the basis of some descriptor
- This is a list of numbers, represented as a vector
  - Typically this is a high-dimensional vector
  - SIFT descriptors, for example, have 128-dimensions
- The distance between matching vectors should be small
- The distance should be low regardless of changes in the image
  - Translation and rotation in the image plane
  - Changes in viewing direction
  - Changes in scale
  - Changes in lighting and brightness

A Simple Feature Descriptor

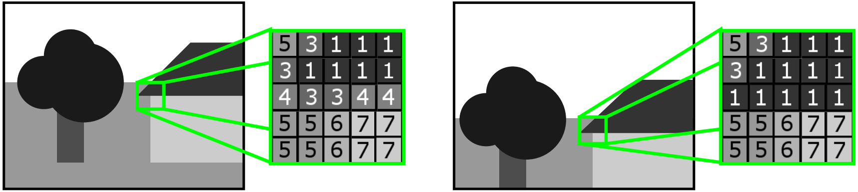
- We could use the pixel values in a window around the feature
  - This is easy to compute, and works well in some cases
  - For simplicity we'll use greyscale images
  - Generalises easily to colour images
- If we take a $n \times n$ window, we get a vector of n^2 values
- We can compare them with the usual (Euclidean) vector distance



$$(5, 3, 1, 1, 1, 3, 1, 1, 1, 1, 4, 3, 3, 4, 4, 5, 5, 6, 7, 7, 5, 5, 6, 7, 7)$$

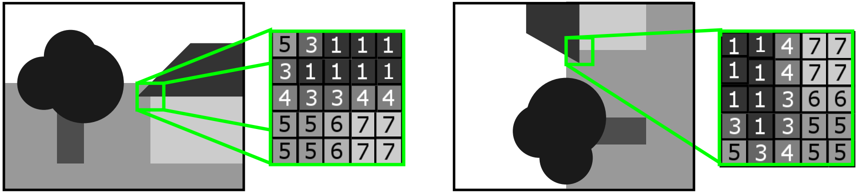
Feature Invariance

- Translation



$$|(0,0,0,0,0,0,0,0,0,0,3,2,2,3,3,0,0,0,0,0,0,0,0,0,0)| = \sqrt{35}$$

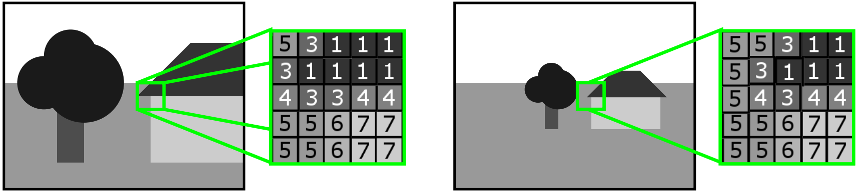
- Rotation



$$|(4,2,-3,-6,-6,-2,0,-3,-6,-6,\ldots,0,2,2,2,2)| = \sqrt{260}$$

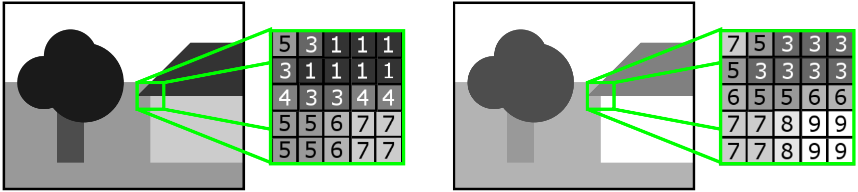
Feature Invariance

- Scale



$$|(0, -2, -2, 0, 0, -2, -2, 0, 0, 0, -1, -1, 0, 0 \ldots, 0, 0)| = \sqrt{18}$$

- Brightness changes



$$|(-2, -2, -2, -2, -2, -2, -2, -2, -2, -2 \ldots, -2, -2)| = \sqrt{100}$$

SIFT Features

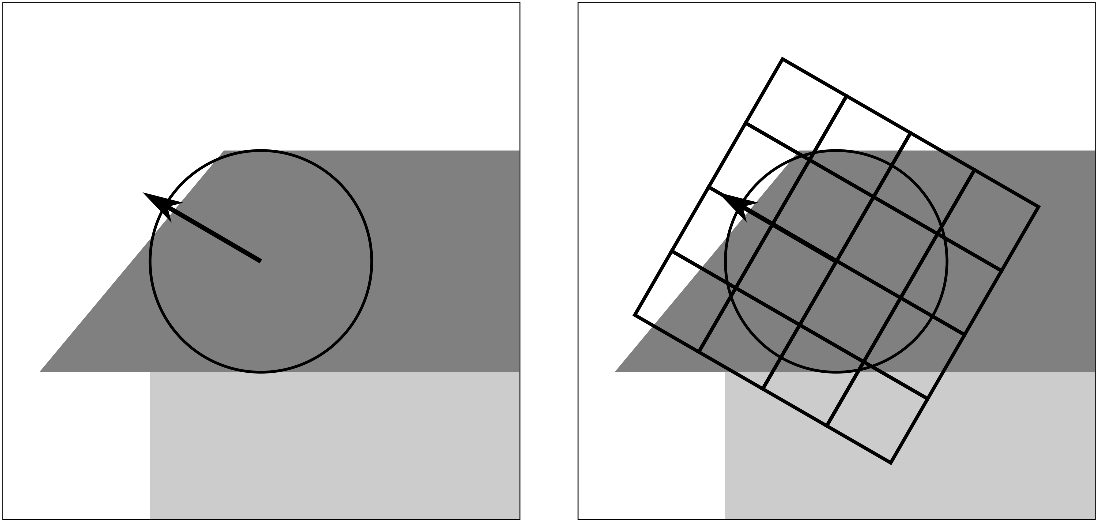
- In 1999 David Lowe proposed an invariant feature detector[1]
- Translation invariance is easy, as we've seen
- Scale invariance comes from using blob features
  - Descriptor is computed from a window around the feature
  - The size of the blob determines the size of the window
- Brightness invariance comes from using image gradients
  - The relative brightness of pixels is fairly constant
  - Gradients do not change much under moderate intensity change
- Rotation invariance comes from finding a dominant gradient direction
  - The window is oriented to the dominant gradient

[1] D. G. Lowe, *Object recognition from local scale-invariant features*, ICCV 1999

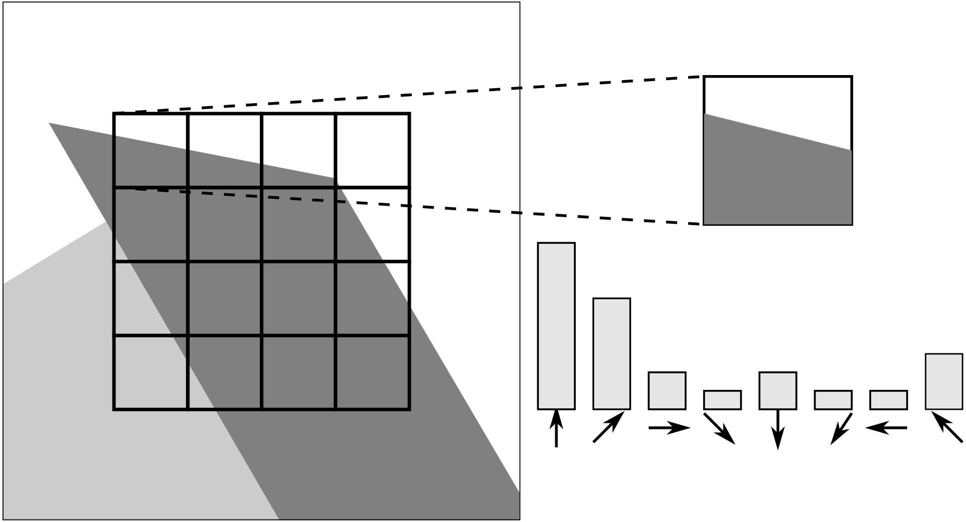
SIFT Features

- Blob features are detected and their scale determined
- A histogram of gradients around the blob are computed
- Peak(s) in the histogram determine the orientation
- A square region is used to compute the descriptor
  - The size of the square comes from the size of the blob
  - The square is aligned to the feature's orientation
- This region is divided into a 4×4 grid of squares
- In each sub-region a gradient histogram is made with 8 bins
- This gives $4 \times 4 \times 8 = 128$ values, which is the descriptor

SIFT Features



SIFT Features

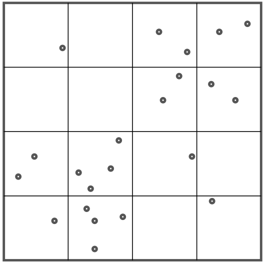


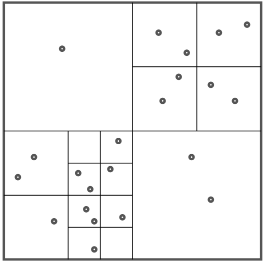
Matching Features

- The final descriptor is 128 values, usually bytes
  - Finding the distance between two descriptors takes 256 operations
  - OK to compute squared difference (no square root needed)
- If we find 10,000 features in each image
  - Matching one feature takes $\sim 2,500,000$ operations
  - Matching all features takes $\sim 25,000,000,000$ operations
- This is often too expensive, so approximate methods are used

Space Subdivision and Approximate Neighbours

- Split space into smaller regions
- 2D examples easier to draw. . .
- Uniform subdivision
  - Division into regular grid
  - Look for neighbours in the same cell as the point we are matching
- Quadtrees, octrees, etc.
  - Recursively split in half
  - Stop splitting when only a few elements in a cell
  - 2D gives a *quadtree*
    3D gives an *octree*



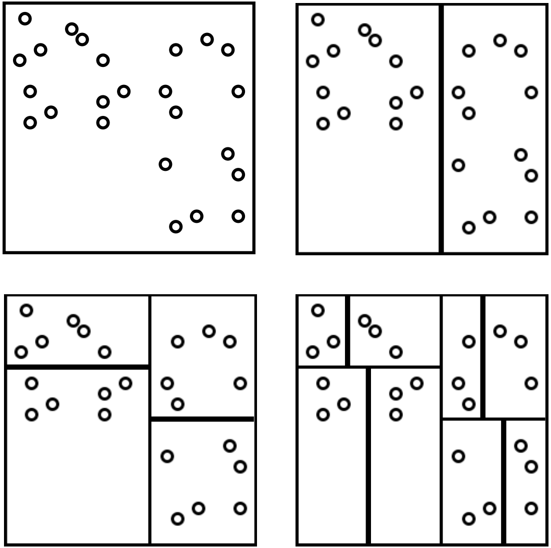


Space Subdivision

- This gets difficult in high dimensions
- Consider uniform subdivision with 8 divisions along each axis
  - In 2D this is $8 \times 8 = 64$ cells
  - In 3D we get $8 \times 8 \times 8 = 512$ cells
  - In nD we get 8^n cells, and $8^{128} \approx 3.9 \times 10^{115}$
- Even if we just have 2 divisions (such as one layer of a generalised quad-/oct-tree), we have $2^{128} \approx 3.4 \times 10^{38}$ cells
- So we can't split along all axes

k-d Trees

- One solution is the use of *k*-d trees
- Choose an axis and split data along it
  - Axis with the greatest spread?
  - The first axis, or a random one?
  - Try to split the data roughly in half
- Then take each half and split again
  - The axis could be chosen as above
  - Try to split each cell's data in half
- Repeat until cells have only a few items



k-d Trees and Feature Matching

- Put all the features in one image into a k-d Tree
- Given a feature from the other image:
  - Find which cell in the k-d Tree it lies in
  - Compute the distance to all features in that cell
  - The nearest one is probably the best match
- For a tree with n layers and 10,000 features this requires:
  - n comparisons to find the appropriate cell
  - $256\frac{10{,}000}{O(2^n)}$ operations in the distance computations
  - If $n = 10$, then $\frac{10{,}000}{O(2^n)} \approx 10$
- This doesn't always find the best match – why not?

Matching SIFT features

- Even if we use brute-force matching most SIFT matches are wrong
  - A lot of blob features don't have much texture detail
  - A lot of scenes have repeating features
  - This leads to ambiguous matches
  - SIFT is often the best we have [2]
- With k-d Trees this gets a little worse, but not much
- Solution: Find the two best matches to check for ambiguity
  - Can use other methods to reject unreliable matches[3]
- Only keep matches if the best distance is much lower than the second
- This makes things better, but still some wrong matches
- Need robust methods (RANSAC)

[2] N. Kahn, B. McCane, S. Mills *Better than SIFT?*, MVA 26(6), 2015

[3] S. Mills, Relative Orientation and Scale for Improved Feature Matching, ICIP, 2013

Application – Image Mosaicing

Basic algorithm:

1. Align features between image pairs
2. Compute Homographies
3. These warp the images to line them up

Details

- Corner tracking or blob matching?
- Incorrect matches cause big problems
- Accumulating transforms over time

$$\mathbf{p}_k = \mathrm{H}_k \mathrm{H}_{k-1} \ldots \mathrm{H}_2 \mathrm{H}_1 \mathbf{p}_0$$

- Blending images together