Image Mosaicing 3

COSC342

Lecture 8
23 March 2016

Mosaicing So Far

- Mosaicing defined by a homography, $\mathbf{p}' = \mathrm{H}\mathbf{p}$
  - H is a 3×3 matrix, defined up to a scale
  - Can compute a homography from 4 corresponding points
- Features are points which can be accurately located in images
  - Corners – points with high gradient in all directions
  - Blobs – have a location and a characteristic scale
- Feature descriptors are calculated from regions around blobs/corners
  - Want invariance to rotation, scale, brightness, etc.
  - SIFT descriptors are widely used
- Can find matches efficiently with k-d Trees
  - Only approximate matches – some will be wrong
- Need to deal with errors and uncertainty

Errors and Uncertainty

- Any measurement has uncertainty
- In mosaicing we measure the location of feature points
  - We get uncertainty because of pixelisation
  - We may not measure exactly the same feature in two images
  - These errors are often modelled with Gaussian distributions
- We also have some measurements which are just wrong
  - Most ($\sim 70\%$) SIFT matches are wrong
  - Removing ambiguous matches drops this to $\sim 30\%$
  - Also k-d Trees only give approximate nearest matches
- We call these incorrect (not just uncertain) measurements *outliers*

Uncertainty and Least Squares Methods

- Gaussian errors can be accounted for by least-squares methods
  - We have some measurements: $m_i, 1 \leq i \leq n$
  - We have a model which makes an estimate, $\hat{m}_i$ of each measurement
  - We minimise
$$\sum_{i=i}^{n} \|m_i - \hat{m}_i\|^2$$
- If we have more than four matches, then we can easily find an H which minimises
$$\|\mathrm{A}\mathbf{h} - \mathbf{0}\|^2$$
- This is a least-squares solution, but $\|\mathrm{A}\mathbf{h}\|$ doesn't mean much

Uncertainty and Least Squares

- A better error to minimise comes from the original equation

$$\mathbf{p}' \equiv \mathrm{H}\mathbf{p}$$

- Since this is an equivalence we get:

$$k \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \equiv \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which leads to

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}$$
$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

Uncertainty and Least Squares

- We want to find an H which minimises

$$\sum_{i=1}^{n} \left(x_i' - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2 + \left(y_i' - \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2$$

- This is a *non-linear least squares* problem
- These are hard to solve directly
- We make an initial guess (such as from solving $Ah = 0$)
- We can then refine this guess with a *gradient descent* method

Gradient Descent

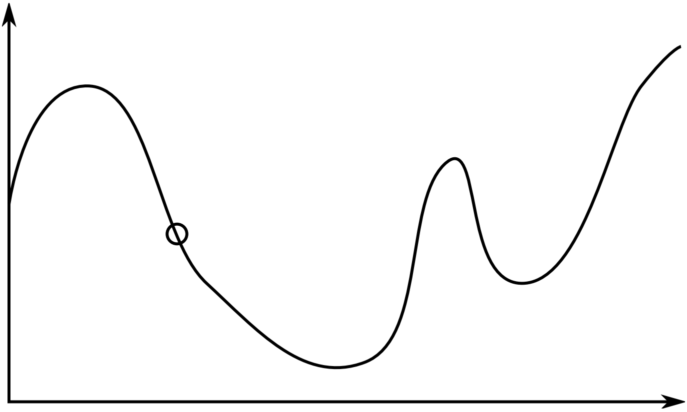
- We want to minimise some function, $f(x)$
- We're given an initial guess, $x = x_0$
- We can approximate $f(x)$ near x_0 with

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

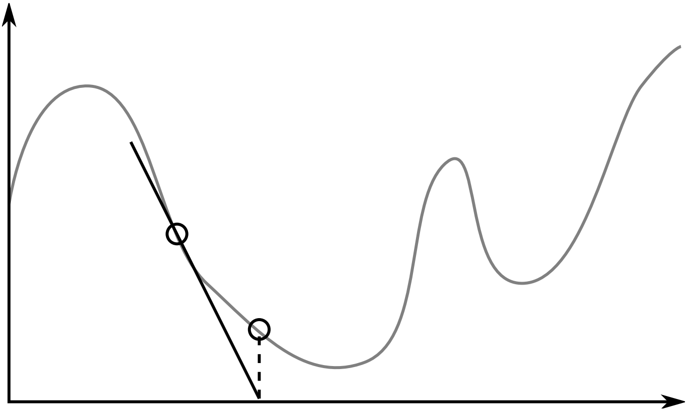
- If we set this equal to 0 (to minimise $f(x)$) we get

$$\begin{aligned} 0 &= f(x_0) + f'(x_0)(x - x_0) \\ x &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

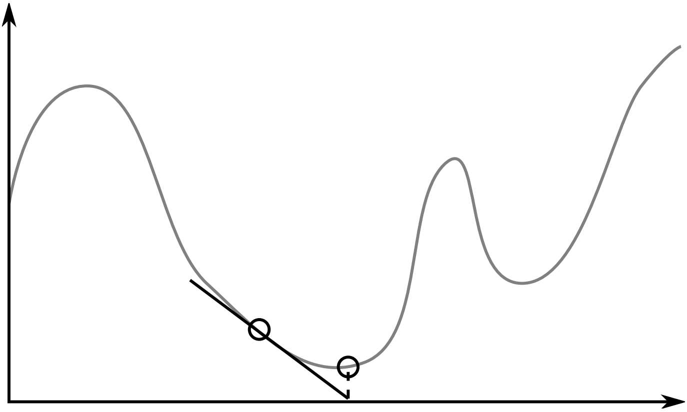
Gradient Descent



Gradient Descent



Gradient Descent

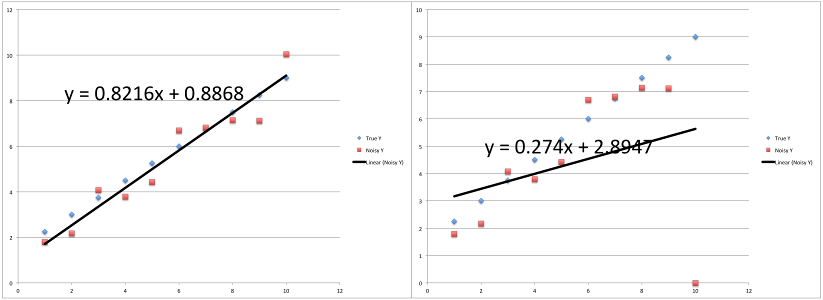


Gradient Descent Problems

- This method can sometimes take too large a step
  - This often happens when the gradient is low
  - If the gradient is not zero, a small enough step always helps
  - Can search for the optimal step to take
- In higher dimensions error functions can get complicated
  - Imagine you're standing on the side of a river valley
  - Which way does the gradient take you?
  - Which way should you go to reach the sea?
- More advanced algorithms account for these problems

Dealing with Outliers

- These methods assume that values are uncertain, but basically correct
- Wildly incorrect values can have a huge effect on the solution



True equation is $y = 0.75x + 1.5$

- We saw in tutorials that RANSAC can help with this

RANSAC and Homography Estimation

- We need four points to estimate a homography
- So the RANSAC version is:
  - Pick four correspondences at random
  - Estimate H from these four points
  - Count how many correspondences agree with H
  - Repeat until you get a large consensus set
  - Fit a least squares solution to the consensus set
- "Agreeing with an estimate of H" means that

$$\left(x_i' - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}}\right)^2 + \left(y_i' - \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}}\right)^2 < d^2,$$

where d is some distance in pixels

How Many RANSAC Trials?

- A common approach is just to do 1000 (or whatever) trials
- In the tutorial we had the formula

$$t = \frac{\log(1-p)}{\log\left(1 - \left(\frac{n}{N}\right)^4\right)},$$

  - t is the number of trials we do
  - p is the chance that we find an outlier-free sample
  - N is the total number of matches to sample from
  - n is the number of correct (inlier) matches
  - 4 is the sample size for each trial.

How Many RANSAC Trials?

- In reality we don't know n
- We can start off with a low estimate of n, say n $= 4$
- Example: $N = 1000, p = 0.99$

$$t = \frac{\log(0.01)}{\log\left(\left(1 - \left(\frac{4}{1000}\right)^4\right)\right.} \approx 18 \text{ billion}$$

- As we find larger consensus sets we update this
- Suppose we find a consensus set with 200 inliers, we get

$$t = \frac{\log(0.01)}{\log\left(\left(1 - \left(\frac{200}{1000}\right)^4\right)\right.} \approx 2,876$$

- As we make more trials, t never increases
- Eventually we've done t trials and can stop

Homography Estimation

- Find feature points in each image
- Compute feature descriptors
- Find (approximate) correspondences
  - Can do this efficiently with a *k*-d Tree
  - Can find two nearest matches and reject ambiguous ones
- Use RANSAC to find an initial guess of H and an consensus set
- Remove the outlier correspondences
- Find a least-squares solution from the inlier set

Mosaicing in OpenCV

- You've already seen a little OpenCV in the labs
- OpenCV has all the things we need for image mosaicing:
  - Reading and writing images in many formats
  - Feature detection and description, including SIFT
  - Feature matching with k-d Trees
  - Homography estimation with RANSAC
  - Image warping and compositing
- There is also a stitching module, but we'll use more generic functions

OpenCV Basics

- OpenCV uses a `cv::Mat` structure for images and matrices
- These can be of different types, some common examples are:
  - `CV_8UC3` – 8 bit unsigned (`8U`), 3 channel (`C3`) images (RGB)
  - `CV_8UC1` – 8 bit unsigned, single channel images (greyscale)
  - `CV_64F` – 64 bit floating point, single channel (assumed) matrices
- Since these can be either images or matrices, there is a choice
  - Index by (x, y) which makes sense for images
  - Index by (row, column), which makes sense for matrices
- Accessing pixel/matrix values is a little awkward

```
image.at<cv::Vec3b>(y,x)[0] = 128; // Set Blue value of a pixel
matrix.at<double>(r,c) = 1.5;      // Set value of a matrix
```

Reading and Displaying Images

```
cv::namedWindow("Display");
cv::Mat image = cv::imread("filename.png");
cv::imshow("Display", image);
cv::waitKey(10);
```

- `cv::imread` by default loads images as colour (`CV_8UC3`) images
- You can pass a second parameter to alter this:
  - `CV_LOAD_IMAGE_GRAYSCALE` Load the image as greyscale
  - `CV_LOAD_IMAGE_UNCHANGED` load the image as stored in the file

- `cv::waitKey` takes a time to wait in milliseconds or a key is pressed
- If you give no time, it waits indefinitely until a key is pressed
- It returns the character code of the key pressed
- It triggers OpenCV's event loop, so refreshes the window display

Feature Detection

- OpenCV has a number of feature detectors implemented
- These produce a list of `cv::KeyPoint` objects
- There are also various descriptors that can be computed
- These include SIFT, which works on greyscale images

```
std::vector<cv::KeyPoint> keypoints;
cv::Mat descriptors;
cv::Mat grey(image.size(), CV_8UC1));
cv::cvtColor(image, grey, CV_BGR2GRAY);
cv::SIFT sift;
sift.detect(grey, keypoints);
sift.compute(grey, keypoints, descriptors);
```

Displaying Keypoints

- The keypoints have positions, orientations, and scales
- We can display them with

```
cv::Mat keypointImage(image.clone());
cv::drawKeypoints(image, keypoints, keypointImage, CV_RGB(0,255,0),
                  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
cv::imshow("Display", keypointImage);
cv::waitKey();
```

Matching Keypoints

- OpenCV supports both brute force and k-d Tree based matching
- The k-d Tree approach lets us find more than one potential match

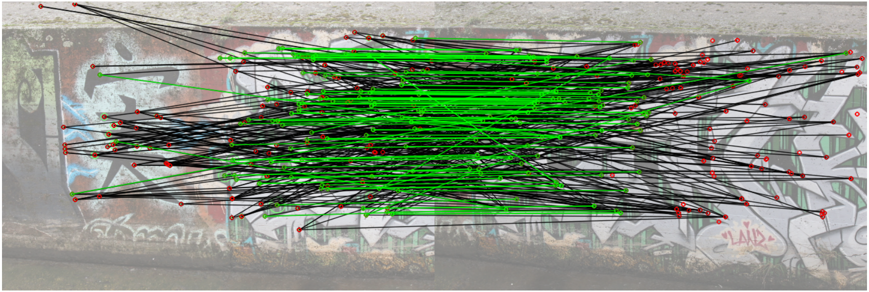
```
cv::FlannBasedMatcher matcher;
std::vector< std::vector< cv::DMatch > > matches;
matcher.knnMatch(descriptors1, descriptors2, matches, 2);
```

- The `cv::DMatch` structure stores the match information
- The index of the first feature is `queryIdx`, and the second is `trainIdx`
- This comes from the use of image matching for search
- The `distance` associated with the match is also returned

Detecting Ambiguous Matches

- We get back two possible matches for each point
- We can check the `distance`s to see if they are ambiguous or not
- Lowe suggests a ratio of 0.8 as a useful threshold

```
std::vector< cv::DMatch > goodMatches;
for (size_t i = 0; i < matches.size(); ++i) {
  if (matches[i][0].distance < 0.8*matches[i][1].distance) {
    goodMatches.push_back(matches[i][0]);
  }
}
```



Homography Estimation

- OpenCV includes a homography estimation routine
- This has a number of approaches, one of which is RANSAC
- We supply it with an error threshold in pixels
- First we need to make lists of corresponding feature locations

```
cv::Mat H(3,3,CV_64F);
std::vector<cv::Point2d> points1;
std::vectro<cv::Point2d> points2;
for (size_t i = 0; i < goodMatches.size(); ++i) {
  points1.push_back( keypoints1[ goodMatches[i].queryIdx ].pt );
  points2.push_back( keypoints2[ goodMatches[i].trainIdx ].pt );
}
H = cv::findHomography(points1, points2, CV_RANSAC, 2.0);
```

Aligning the Images

- The homography can now be used to align the two images
- We make a large image to store the mosaic, and warp each image to it
- The first image can use an identity transform, the second uses `H`

```
cv::Mat mosaic(cv::Size(1000, 1000), cv::8UC3);
cv::Mat I = cv::Mat::eye(3,3,CV_64F);
cv::warpPerspective(image1, mosaic, I, mosaic.size(),
  cv::INTER_CUBIC + cv::WARP_INVERSE_MAP, cv::BORDER_TRANSPARENT);
cv::warpPerspective(image, mosaic, H, mosaic.size(),
  cv::INTER_CUBIC + cv::WARP_INVERSE_MAP, cv::BORDER_TRANSPARENT);
```

Aligning Multiple Images

- If the images come in a sequence we can chain together homographies
- We initialise a transform, T, as the identity matrix for the first image
- For the ith image we compute the homography to the $(i-1)$th image
- We update $\mathrm{T} \leftarrow \mathrm{H}_{(i-1)\rightarrow i}\mathrm{T}$, then use T to warp the ith image

```
cv::Mat T = cv::Mat::eye(3,3,CV_64F);
cv::warpPerspective(image[0], mosaic, T, ...);
for (int i = 1; i < numImages; ++i) {
  // Compute H to align image[i] to image[i-1]
  T = H*T;
  cv::warpPerspective(image[i], mosaic, T, ...);
}
```