Perception:
Optical Flow and RANSAC
Last Time:

Procrustes Problem: \[ \arg \min_{R \in SO(3), T \in \mathbb{R}^3} \sum_i \|RP_i + T - P'_i\|^2 \]

Epipolar Geometry and the Essential Matrix:

\[ x^T_i [T] \times Rx'_i = 0 \]
Now: Point Correspondences

In all the algorithms we have covered thus far we have simply assumed correspondences were given – but how do we compute them?
Image Motion

To try and solve this problem let’s consider two consecutive frames in a video

\[ I_t(x, y) \quad \quad I_{t+1}(x, y) \]
Image Motion: Goal

We want to find correspondences between the two images

\[ I_t(x, y) \rightarrow I_{t+1}(x, y) \]
Optical Flow

Looking the frames overlaid with each other you can better see where the motion is

In this perspective, we see we could move each pixel in the first to match the second. The motion of each pixel in this is called optical flow

Optical flow tells us which pixel in the second image a pixel in second image corresponds to
You can see that some of the patches are easily distinguishable from nearby patches.
Optical Flow - Patches

Some are not…
Optical Flow - Patches

Let’s focus on the ones we can match (we’ll characterize them later)
Optical Flow as Local Search

If the video frames are close enough we can look for these salient points in nearby images!

\[ I_t(x, y) \]

\[ I_{t+1}(x + \delta x, y + \delta y) \]

Which nearby patch in the next image is the closest to the current frame?
Optical Flow as Local Search

Side note: Some people use the reverse convention in terms of which image to search on, but the math is the same.

$$I_t(x - \delta x, y - \delta y)$$

$$I_{t+1}(x, y)$$

Which nearby patch in the next image is the closest to the current frame?
Local Search Algorithm

The function we are trying to optimize is (assuming a continuous image):

\[(\delta x, \delta y) = \arg \min_{\delta x, \delta y} \int \int_{(x,y) \in \mathcal{N}(x_0,y_0)} (I_{t+1}(x, y) - I_{t+1}(x+\delta x, y+\delta y))^2 dx \, dy\]

If we discretize:

\[\sum_{(x,y) \in \mathcal{N}(x_0,y_0)} I_t(x - \delta x, y - \delta y)\]

If time is no issue, we can solve this by exhaustive search.
Local Search Simplification

If we want to speed things up. Assume that the change is small, we can use Taylor expansion:

\[
I_{t+1}(x, y) - I_{t+1}(x + \delta x, y + \delta y) \\
\approx I_t(x, y) - \left( I_{t+1}(x, y) + \nabla I_{t+1}(x, y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} \right)
\]

\[
I_t(x, y) = I_t(x - \delta x, y - \delta y) - \nabla I_{t+1}(x, y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}
\]
Local Search Simplification

\[
\Delta I_t(x, y) - \nabla I_{t+1}(x, y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}
\]

Notice the clear direction where optical flow should go.
Local Search Simplification

\[ \Delta I_t(x, y) = \nabla I_{t+1}(x, y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} \]

\[ \nabla_x I_{t+1}(x, y) \]

\[ \nabla_y I_{t+1}(x, y) \]

\[ \| \nabla I_{t+1}(x, y) \| \]
Local Search Simplification

The new cost function becomes (in discretized form):

\[
(\delta x, \delta y) = \arg \min_{\delta x, \delta y} \sum_{(x,y) \in \mathcal{N}(x_0,y_0)} \left( \Delta I_t(x,y) - \nabla I_{t+1}(x,y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} \right)^2
\]

\[
= \sum_{(x,y)} \Delta I_t(x,y)^2 - 2 \Delta I_t(x,y) \nabla I_{t+1}(x,y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}^T \nabla I_{t+1}(x,y) \nabla I_{t+1}(x,y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}
\]

\[
= \sum_{(x,y)} -2 \Delta I_t(x,y) \nabla I_{t+1}(x,y)^T \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}^T \left( \sum_{(x,y)} \nabla I_{t+1}(x,y) \nabla I_{t+1}(x,y)^T \right) \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}
\]

We can solve this directly!

\[
\begin{pmatrix} \delta x^* \\ \delta y^* \end{pmatrix} = \left( \sum_{(x,y)} \nabla I_{t+1}(x,y) \nabla I_{t+1}(x,y)^T \right)^{-1} \left( \sum_{(x,y)} \Delta I_t(x,y) \nabla I_{t+1}(x,y) \right)
\]
Assumptions We’ve Made

Brightness Constancy: Color doesn’t change between different viewpoints

No occlusions: things stay in sight:

Counterexample
Assumptions We’ve Made

Minimal geometric deformations: No large rotations or scaling

Minimal patch displacement: Taylor expansions only work for small translations
Assumptions We’ve Made

Patch is sufficiently ‘interesting’:

\[
\begin{pmatrix}
\delta x^* \\
\delta y^*
\end{pmatrix} = \left( \sum_{(x,y)} \nabla I_{t+1}(x, y) \nabla I_{t+1}(x, y)^T \right)^{-1} \left( \sum_{(x,y)} \Delta I_t(x, y) \nabla I_{t+1}(x, y) \right)
\]

We assumed we could invert this

When would this fail?
And what does that mean?

\[
det \left( \sum_{(x,y)} \nabla I_{t+1}(x, y) \nabla I_{t+1}(x, y)^T \right) = 0
\]
White Wall Problem

\[
\begin{pmatrix}
\delta x^* \\
\delta y^*
\end{pmatrix}
= \left( \sum_{(x,y)} \nabla I_{t+1}(x, y) \nabla I_{t+1}(x, y)^T \right)^{-1} \left( \sum_{(x,y)} \Delta I_t(x, y) \nabla I_{t+1}(x, y) \right)
\]

When would this fail?

1) \( \nabla I_{t+1}(x, y) = 0 \)  
No gradient = no information. 
This is the ‘White wall problem’

Indistinguishable

In our example, the grass problem
Barber Poll Problem

\[
\begin{pmatrix}
\delta x^* \\
\delta y^*
\end{pmatrix}
= \left( \sum_{(x,y)} \nabla I_{t+1}(x, y) \nabla I_{t+1}(x, y)^T \right)^{-1}
\left( \sum_{(x,y)} \Delta I_t(x, y) \nabla I_{t+1}(x, y) \right)
\]

When would this fail?

2) \( \nabla I_{t+1}(x, y) = \bar{c}, \ \forall x, y \) Constant gradient = insufficient information. No information along a direction

Indistinguishable along this direction

Hence it looks like it moves up

For our example, how high along the tree are we?
Most places in the image are ‘uninteresting’ – we can’t track them – the interesting places are ‘sparse’. So we need to detect interesting ‘features’ (also known as corners) in the image that we can reliably track. How can we find them?
Finding Features/Corners

There is a large literature on this spanning decades, but it typically involves the eigenvalues of that matrix:

$$\left( \sum_{(x,y)} \nabla I_{t+1}(x,y) \nabla I_{t+1}(x,y)^T \right)$$
Extensions

There is a great deal of research trying to avoid these assumptions:

- Solve for the flow iteratively
- Compute flow bi-directionally for better reliability
- Assume affine deformations on the optical flow
- Compute the flow on multiple resolutions of the image and iterate to capture larger flow
Finding Reliable Features

So we’ve found a sparse set of interesting features that we now can track. Even with this, sometimes failures happen. We need to make sure when we fit our models we have good features.
Minimal Example: Line Fitting

The easiest model to fit is a line – we’ll use it as an illustration because the parameter space is only 2D
Hough Transform

One way to try and solve this is to use the Hough Transform. It works well when you have a low dimensional parameter space.

Parameterize the line differently:

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]

Noise model:

\[ p_i \text{ drawn from } \exp \left( -\frac{1}{2\sigma^2} (\cos(\theta)x + \sin(\theta)y - \rho) \right) \]
Hough Transform: Algorithm

Set up an array with all possible parameters you could have.

In the case of a line, a 2D array

$$\cos(\theta)x + \sin(\theta)y - \rho = 0$$

This line corresponds to a point in parameter space.
Hough Transform: Algorithm

Every point has an infinite number of lines that goes through it. Mark all those lines in parameter space

$$\cos(\theta)x + \sin(\theta)y - \rho = 0$$
Hough Transform: Algorithm

Every point has an infinite number of lines that goes through it. Mark all those lines in parameter space

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]
Hough Transform: Algorithm

Every point has an infinite number of lines that goes through it. Mark all those lines in parameter space

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]
Hough Transform: Algorithm

Every point has an infinite number of lines that goes through it. Mark all those lines in parameter space

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]
Hough Transform: Algorithm

Every point has an infinite number of lines that goes through it. Mark all those lines in parameter space

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]

Point off the line intersects at totally different places
Hough Transform: Algorithm

After marking the lines that go through every point, check to see which lines has the most points ‘voting’ for it.

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]

As desired, the most votes are in the right place.
Hough Transform: Multiline Fitting

The advantage of this voting method is that you can even find multiple lines in one image.

\[ \cos(\theta)x + \sin(\theta)y - \rho = 0 \]

Both are in the right place.
Hough Transform: Downsides

While this works great for line fitting (so for instance finding edges in images) it has numerous issues:

• The memory grows exponentially in the parameter space (why?)

• Need bounds on the parameter space

Is there another method that can overcome these problems?
Sample Consensus

Instead of try to store all the parameters, let’s just try to fit all possible minimal models on our data and see which has the most inliers

Inlier Count: 4

See how many points are within the threshold
Sample Consensus

Instead of try to store all the parameters, let’s just try to fit all possible minimal models on our data and see which has the most inliers

Inlier Count: 7

See how many points are within the threshold
Sample Consensus

Instead of try to store all the parameters, let’s just try to fit all possible minimal models on our data and see which has the most inliers.

Inlier Count: 7
Sample Consensus

Instead of trying to store all the parameters, let’s just try to fit all possible minimal models on our data and see which has the most inliers.

Inlier Count: 29

Better than any of the others so we pick this one.

See how many points are within the threshold.
Sample Consensus

Upsides:
• Simple

Downsides
• Computation grows combinatorically with the number of parameters of the minimal model

Line: $\binom{n}{2}$  Homography: $\binom{n}{4}$  Essential Matrix: $\binom{n}{5}$

<table>
<thead>
<tr>
<th>Model</th>
<th>10 points</th>
<th>100 points</th>
<th>1000 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>45</td>
<td>4950</td>
<td>499500</td>
</tr>
<tr>
<td>Homography</td>
<td>210</td>
<td>3921225</td>
<td>$4.14 \times 10^{10}$</td>
</tr>
<tr>
<td>Essential Matrix</td>
<td>252</td>
<td>75287520</td>
<td>$8.25 \times 10^{12}$</td>
</tr>
</tbody>
</table>

Number of models to estimate by number of points
RANSAC: Random Sample Consensus

Popularly used in vision since the 1980s
It is just sample consensus except you pick samples randomly and you stop after $k$ iterations

Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography

Martin A. Fischler and Robert C. Bolles
SRI International
RANSAC: Random Sample Consensus

The algorithm (in pseudocode):

Repeat for $k$ iterations
1. Choose a minimal sample set
2. Count the inliers for this set
3. Keep maximum, if it exceeds a desired number of inliers stop.

This fixes the problem with the number of iterations growing too fast, but how do we know we will actually hit an inlier set? As this is a random algorithm, we will need to do probabilistic analysis to see how it works.
Probability of Hitting Inliers

Say the proportion of inliers is $\epsilon$, and the minimum number of points for your model is $M$. Then the chance of hitting all inliers is $\epsilon^M$.

Thus the chance after $k$ iterations of you NEVER hitting an inlier set is: $\left(1 - \epsilon^M\right)^k$.

Thus the probability of hitting at least one inlier set, i.e. the probability of success, is:

$$p_{success} = 1 - \left(1 - \epsilon^M\right)^k$$

So if you want a given chance of success:

$$k = \frac{\log(1 - p_{success})}{\log(1 - \epsilon^M)}$$
Probability of Hitting Inliers

\[ k = \frac{\log(1 - \rho_{success})}{\log(1 - \epsilon^M)} \]

For \( \rho_{success} = 0.99 \) the \( k \) needed for various models:

<table>
<thead>
<tr>
<th>Model</th>
<th>( \epsilon = 0.8 )</th>
<th>( \epsilon = 0.5 )</th>
<th>( \epsilon = 0.3 )</th>
<th>( \epsilon = 0.1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line (2 points)</td>
<td>5</td>
<td>17</td>
<td>49</td>
<td>459</td>
</tr>
<tr>
<td>Homography (4 points)</td>
<td>9</td>
<td>72</td>
<td>567</td>
<td>46050</td>
</tr>
<tr>
<td>Essential Matrix (5 points)</td>
<td>12</td>
<td>146</td>
<td>1893</td>
<td>460515</td>
</tr>
</tbody>
</table>
Probability of Hitting Inliers

For $p_{\text{success}} = 0.99$ the $k$ needed for various models:

$$k = \frac{\log(1 - p_{\text{success}})}{\log(1 - \epsilon^M)}$$

If you have less than 10% inliers, maybe you should just work on that.
Next Time

In previous lectures we learned how to estimate position and orientation from images. Now we want to be able to measure velocity. We have most of the tools already, we just need to get a model of how motion on the image relates to motion in the 3D world.