Visual SLAM Overview

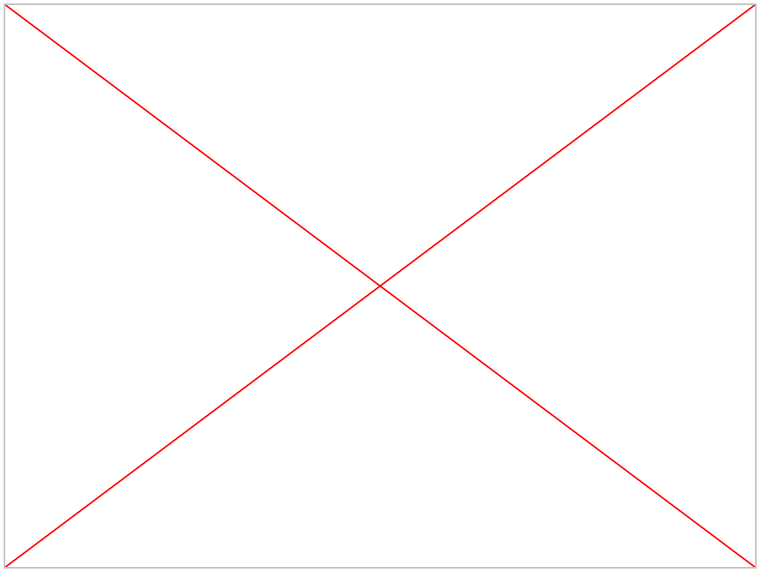
By: Zara Coakley, Owen Himsworth, Bhargavi Deshpande

What is Visual SLAM?

Visual Simultaneous Localization And Mapping

Using visual data, we can apply this Visual SLAM algorithm to:

- drive a robot into an unknown area
- create a map of the area
- while determining the robots position within the map



Structure Overview

Front End Algorithms: Responsible for estimating camera pose and 3D map points

- Feature Extraction
- Feature Matching
- Pose Estimation
- Triangulation
- Local Mapping

Back-End Algorithms: Responsible for optimizing camera pose and map

- Bundle Adjustment
- Loop Closure Detection
- Pose Graph Optimization

Demo Images

Load two consecutive frames and apply grayscale

1. import cv2 as cv, numpy as np
2. I0 = cv.imread("frame0.jpg"); I1 = cv.imread("frame1.jpg")
3. g0 = cv.cvtColor(I0, cv.COLOR_BGR2GRAY); g1 = cv.cvtColor(I1, cv.COLOR_BGR2GRAY)

Feature Extraction

Identifying **Feature Points:**

Unique points within the Image

Composed of **2** parts:

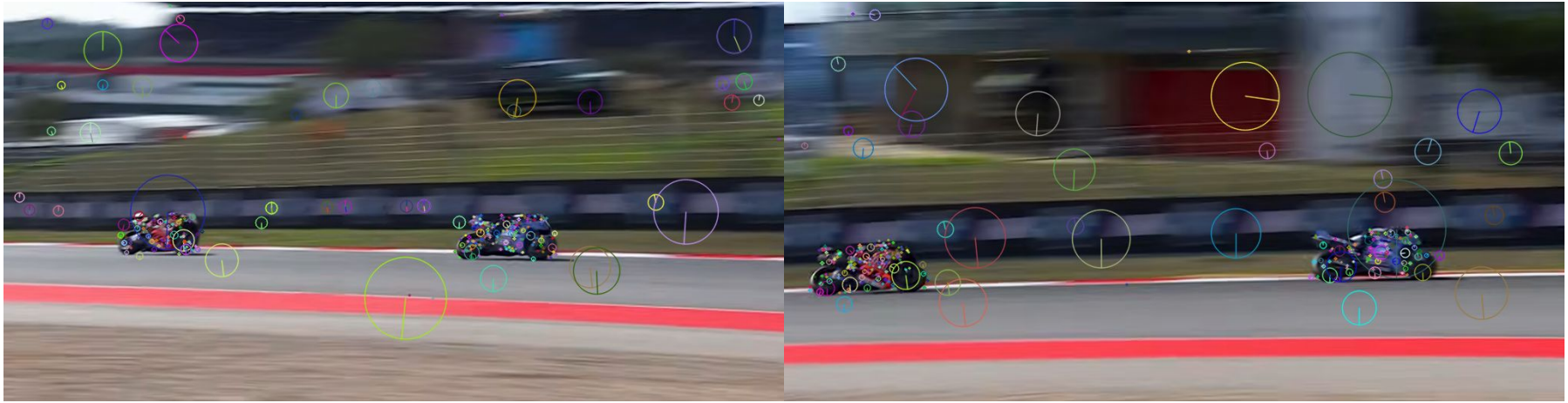
- Key Points
  - 2D Image position on image
- Descriptors
  - Information vector
  - Describes pixels around key point
  - Features with similar descriptors can be lumped to one feature

Feature Extraction Algorithms

- Scale Invariant Feature Transform(SIFT)
  - Images are scaled using Gaussian scaling
  - Features are chosen as dark points across all scaled images (scaled invariant points)
  - Robust but computational heavy
- Features from Accelerated Segment Test(FAST)
  - Detects corners in a image using high contrast surrounding pixels
  - Fastest on this list, but not scale-invariant and sensitive to noise
- Binary Robust Independent Elementary Features(BRIEF)
  - Uses a short binary descriptor, scale-invariant but performs poorly with rotations
  - Extremely fast point recognition rate and lower memory requirements compared to SIFT and SURF
- Oriented Fast and Rotated BRIEF(ORB)
  - Uses FAST algorithm for feature detection and modified BRIEF descriptors (better with rotation handling)
  - Fast, good for low-power devices, and real-time applications
  - Does poorly with high scale, rotation, or lighting
- & many more!

SIFT

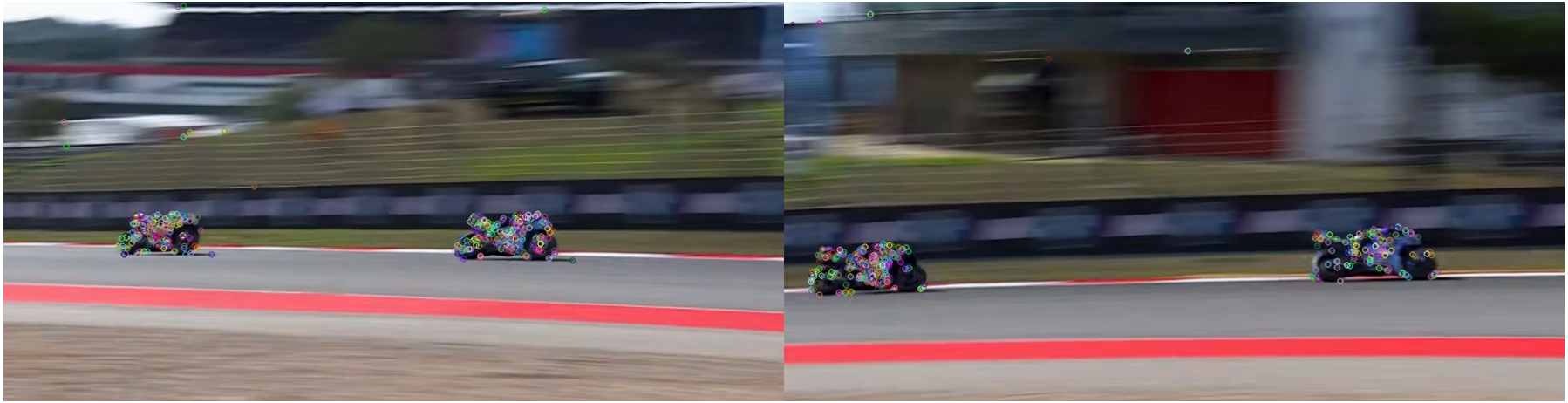
1. sift = cv.SIFT_create(nfeatures=1200)
2. k0, d0 = sift.detectAndCompute(g0, None)
3. k1, d1 = sift.detectAndCompute(g1, None)



FAST

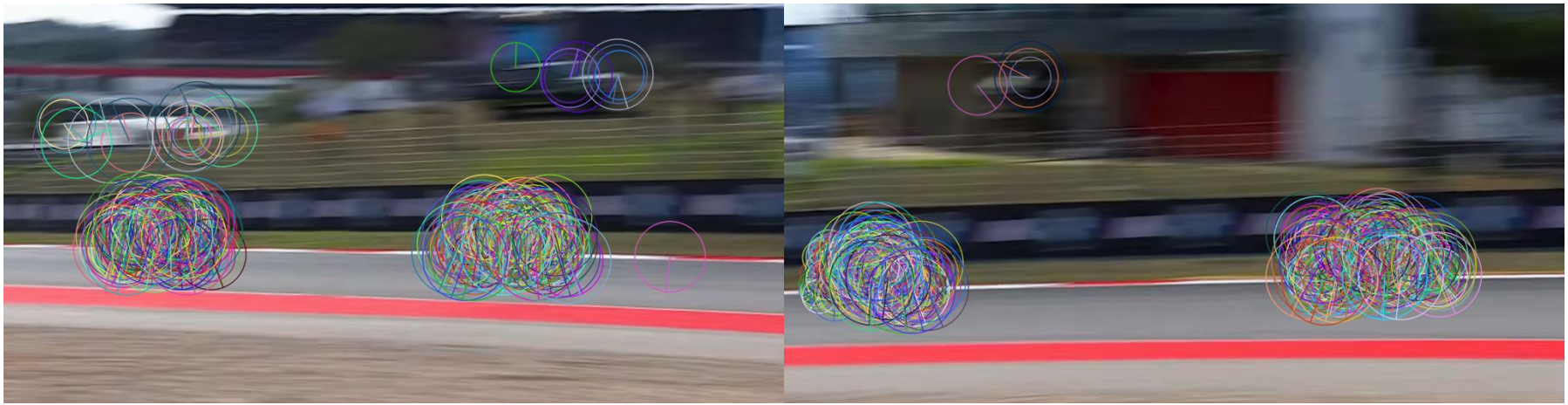
1. fast = cv.FastFeatureDetector_create(threshold=20, nonmaxSuppression=True)
2. k0f = fast.detect(g0, None)
3. k1f = fast.detect(g1, None)
4. k0f = sorted(k0f, key=lambda k: -k.response)[:1200]
5. k1f = sorted(k1f, key=lambda k: -k.response)[:1200]

FAST



ORB

1. orb = cv.ORB_create(nfeatures=1000)
2. k0, d0 = orb.detectAndCompute(g0, None)
3. k1, d1 = orb.detectAndCompute(g1, None)



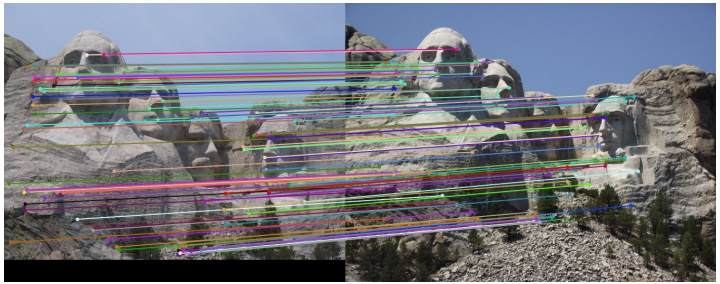
Feature Matching

Between two consecutive frames - find the matching features

F1: Set of identified features in frame 1

F2: Set of identified features in frame 2 (consecutive from frame 1)

1. For each feature in F1 compute *sum of squared differences* with each F2
2. Best Match (smallest dist. difference) / Second Best Match
3. If ratio is above threshold, keep the matched pair (confidence)



SIFT

1. bf = cv.BFMatcher(cv.NORM_L2)
2. raw = bf.knnMatch(d0, d1, k=2)
3. good = [m for m, n in raw if m.distance < 0.75 * n.distance]

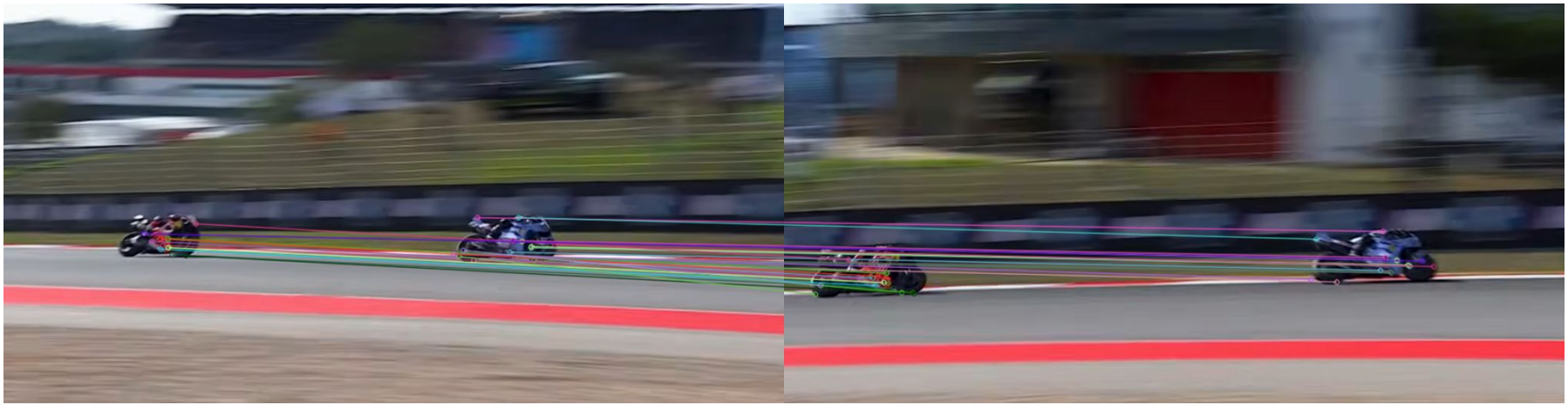
ORB

```
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=False)
raw = bf.knnMatch(d0, d1, k=2)

good = []
for m,n in raw:
   if m.distance < 0.75*n.distance:  # Lowe ratio
       good.append(m)

pts0 = np.float32([k0[m.queryIdx].pt for m in good])
pts1 = np.float32([k1[m.trainIdx].pt for m in good])
```

ORB



Feature Tracking

Used as a replacement for Feature Matching

Follows kepoints from frame to frame as opposed to re-identifying key points between frames

More computationally efficient, but performs poorly with large motion, drift, and missing correspondence

Example Workflow:

1. Pick good corners (Shi–Tomasi/FAST)
2. Track with pyramidal LK (forward)
3. Forward–backward check ➔ keep reliable tracks

Lucas-Kanade(LK) Feature Tracking

- Tracks selected points from frame *t* → *t+1* by finding the small motion that best preserves local appearance.
- Assume **brightness constancy** & **small motion**. Solve for displacement *(u,v)* that minimizes

$$\sum_{\mathbf{x}\in\mathcal{N}} \left(I_{t+1}\left(\mathbf{x}+\mathbf{d}\right)-I_t\right)\mathbf{x}^2\right)$$

over a small window $\mathcal{N}$ around each point. Do this across a pyramid to handle larger motion.

LK Code

```
1.  p1, st, err = cv.calcOpticalFlowPyrLK(
2.      g0, g1, p0, None,
3.      winSize=(21,21), maxLevel=3,
4.      criteria=(cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT,30, 0.01)
5.  )
6.  p0f = p0[st[:,0]==1]; p1f = p1[st[:,0]==1]
```

Forward Backward(FB) Checking

Forward–backward (FB) checking is a simple consistency test for feature tracking.

Acts as validation of motion on top of LK Tracking

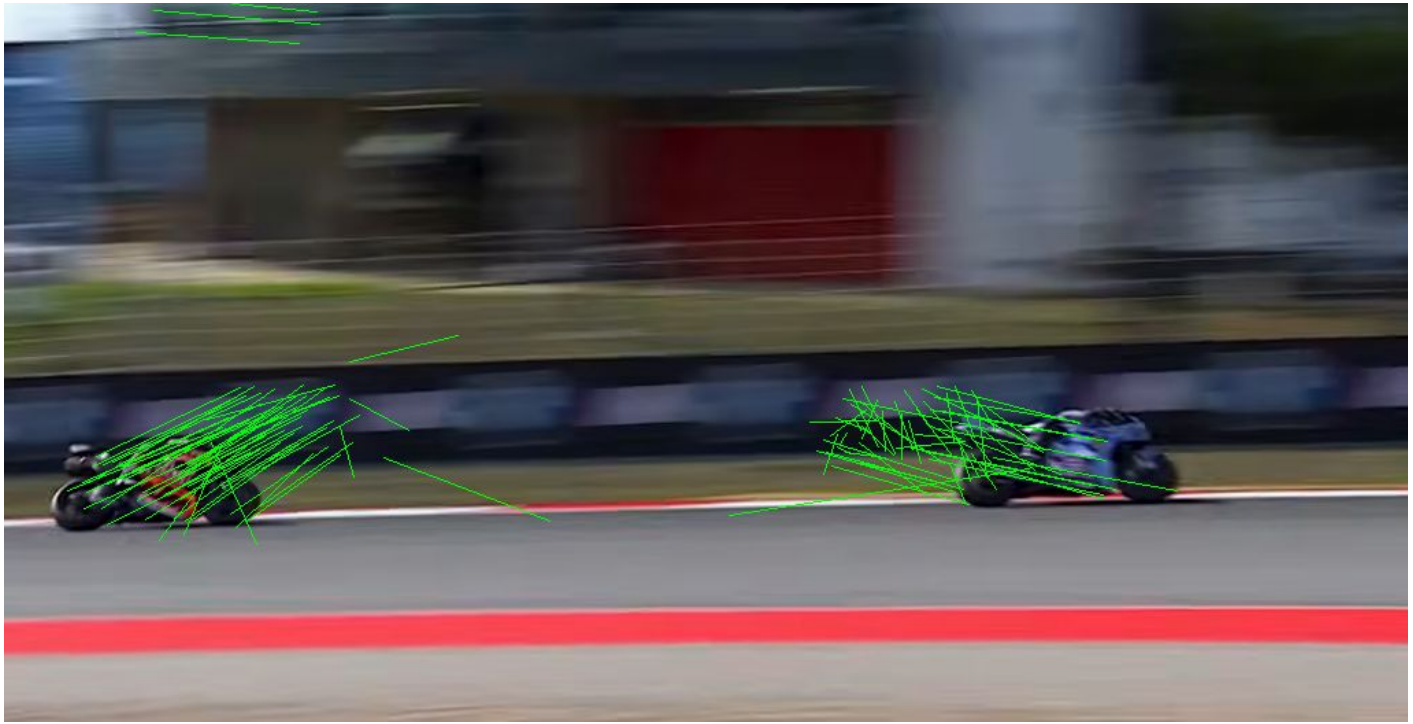
1. Track points **forward** from frame *t* ➔ *t+1* to get p_1.
2. Track those results **backward** from *t+1* ➔ *t* to get $\hat{p}_0$.
3. If the round-trip returns close to the original ($\|p_0 - \hat{p}_0\|$ is small), the track is likely reliable. If not, discard it (it probably drifted, hit an occlusion, or was ambiguous).

- Basically LK in reverse

FB Code

```
1.  p0b, stb, _ = cv.calcOpticalFlowPyrLK(g1, g0, p1f,None,
2.      winSize=(21,21), maxLevel=3
3.  )
4.  good = (stb[:,0]==1) & (np.linalg.norm(p0f - p0b, axis=2).ravel() <
5.      1.0
6.  )
7.  p0g, p1g = p0f[good], p1f[good]
```

Without Forward-Backward Check



With Forward-Backward Check

Camera Pose Estimation:

We have: 2D pixel correspondences between two frames

We want to know: Rotation and Translation of camera between frames (localization)

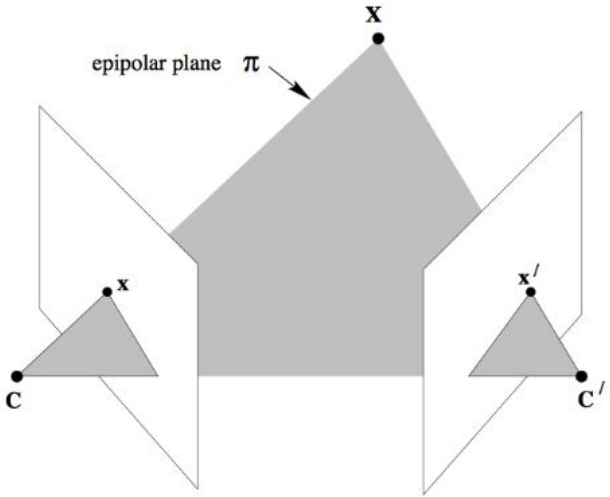
We can use epipolar geometry to relate the corresponding points between frames and understand how the cameras moved between them

Epipolar Geometry

Through this geometry: x, x', and X lie on the same plane (coplanar)

Epipolar Geometry: Tells us that the project of both image points must lie at the same 3D point

Essential Matrix E:
Encodes this epipolar geometry up to a scale



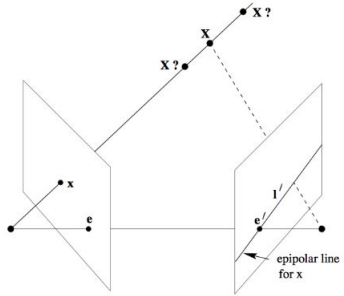
C: Camera Position Frame 1

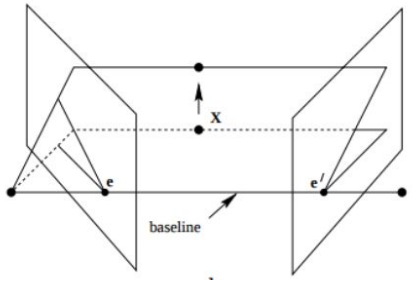
C': Camera Position Frame 2

x: Pixel in Frame 1

x': Corresponding pixel in frame 2

X: 3D point (x and x') in space





Camera Pose Estimation

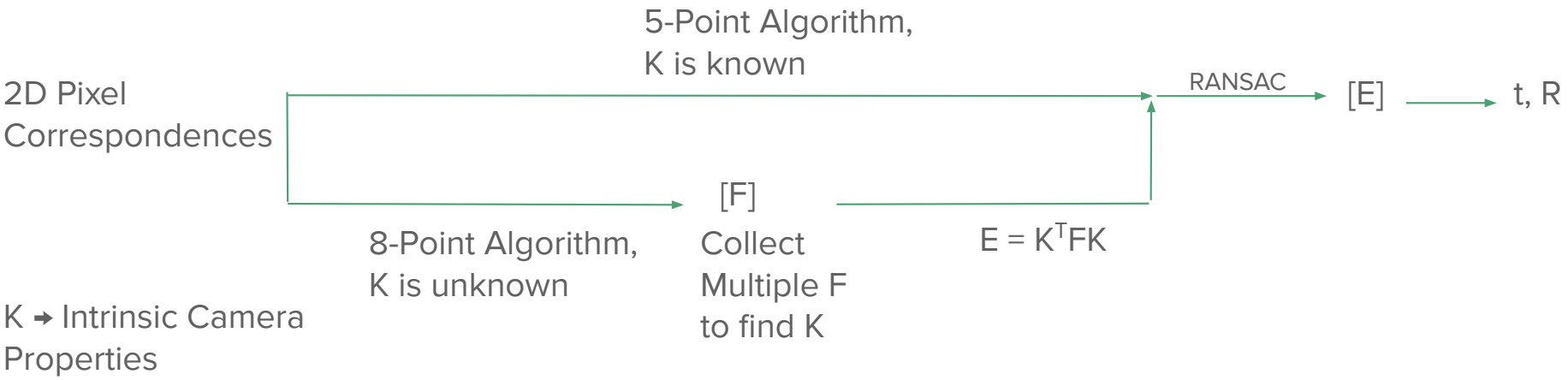
$E = [t]_x R$

where
E: Essential Matrix
t: unit vector of translation (bc no scale)
R: how image 1 is rotated relative to image 2

We want to find [E] so that we can decompose to get R and t !!

RANSAC: RANdom SAmple Correspondence
Not all 2D correspondences will be correct

- Algorithm to determine outliers v.s. Inliers
- Removes points outside of error threshold



Epipolar Triangulation

Now we have: 2D Points, Camera rotation and translation between image 1 and 2

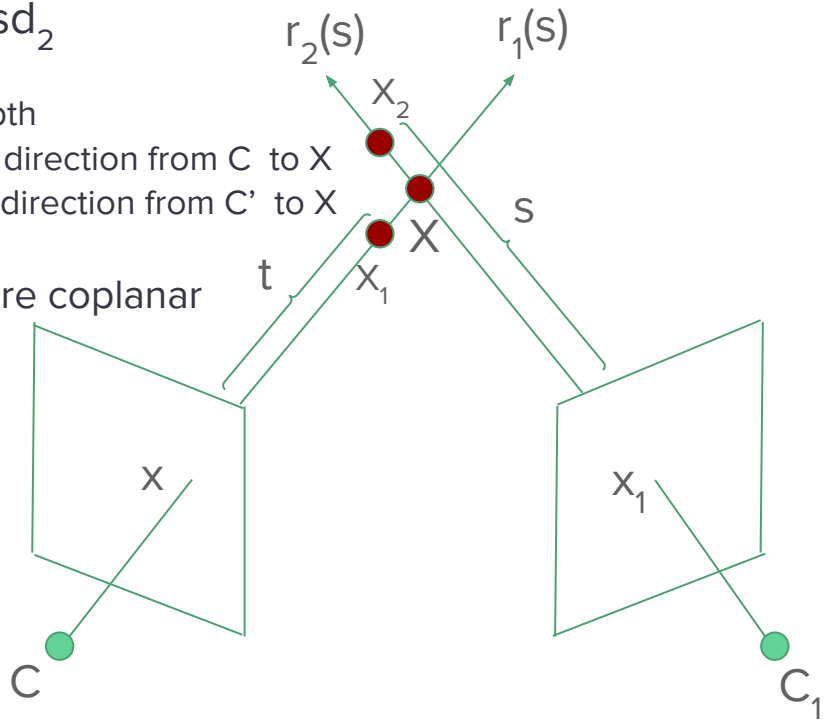
Want to know: 3D Point in space (depth!)

Epipolar Triangulation Cont.

$r_1(s) = C + td_1$
$r_2(s) = C' + sd_2$

t, s : scalar depth
d_1 : unit vector direction from C to X
d_2: unit vector direction from C' to X

x, x_1, and X are coplanar



$r_1(s) = r_2(s)$
$X_1 = X_2$
$C + td_1 = C' + sd_2$

$$[d_1 \; -d_2] \begin{bmatrix} s \\ t \end{bmatrix} = C_1 - C$$

Using Least Squares ➔ solve for s and t

But because noise:
Plug in s and t into $r_1(s)$ and $r_2(s)$ to find X_1 and X_2

$\mathbf{X} = ½(X_1 + X_2)$

Demo - Estimating Camera Pose of a Car

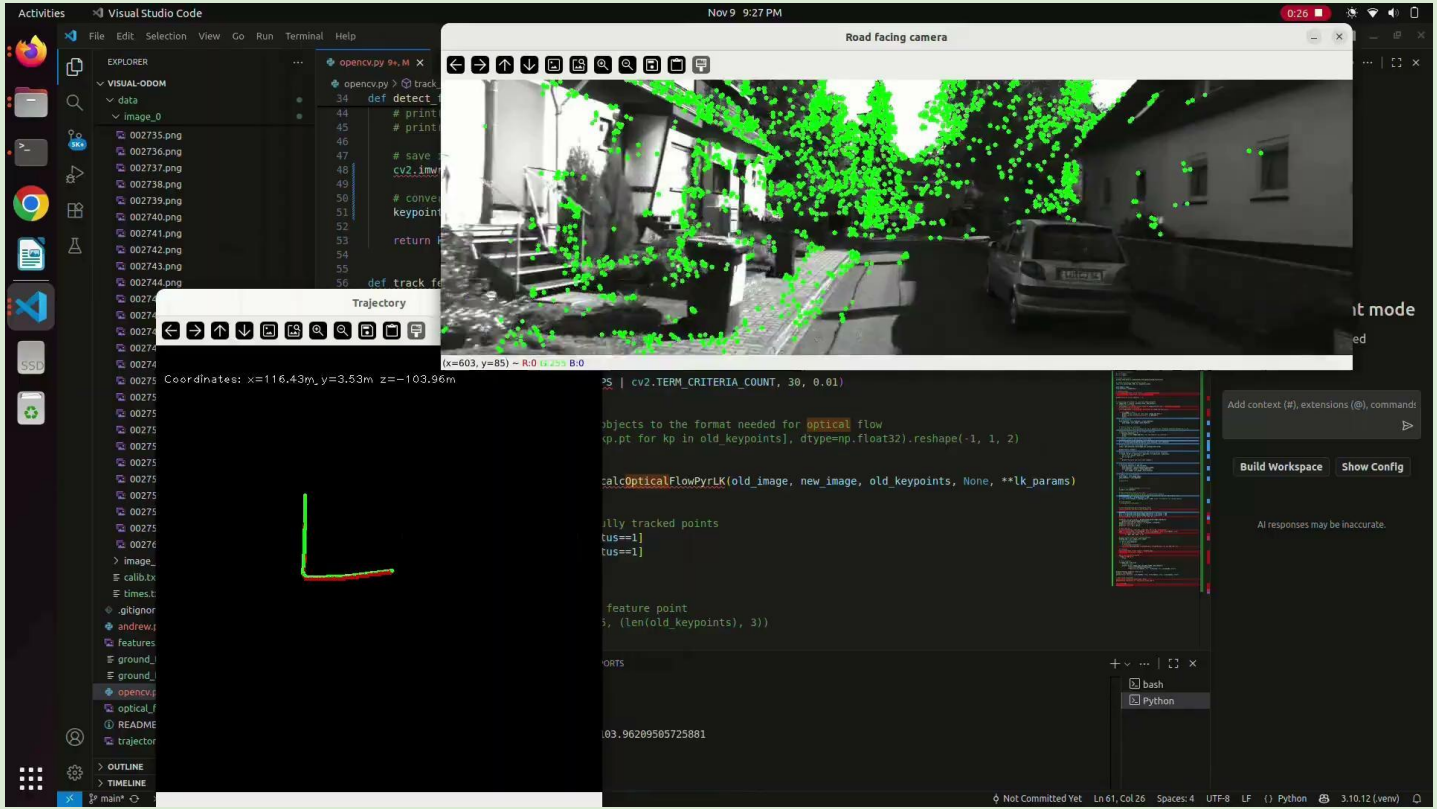
Elements:

1. Feature extraction using FAST
2. Feature tracking between video frames
3. Camera pose estimation using RANSAC (to get the essential matrix E)
4. Decompose the essential matrix into R and t
5. t is your xyz position, which you can graph!

OpenCV Functions:

1. cv.FastFeatureDetector_create()
2. cv.calcOpticalFlowPyrLK()
3. cv.findEssentialMat()
4. cv.recoverPose()

Demo - Estimating Camera Pose of a Car



Local Mapping

The step that actually places the 2D points onto the newly created 3D local map

Updates the 3D depth with new keypoint/observations as they occur

Many algorithms for this ➔ core step of SLAM

Kalman Filters, Particle Filters, Sparse Maps, Dense Maps, etc...

- Predict and Update position of each feature in 3D
- Predict and Update position of camera in 3D

Keyframes are identified in the map

- Contains high density of features ➔ used for mapping and pose estimation
- Image at specific time + corresponding camera pos. and orientation

Bundle Adjustment + Local Optimization

Over the course of the process, error accumulation could occur

Minimizes the difference between where the points are in the image and where they should be according to the 3D map ➔ called reprojection error

Bundle adjustment is similar to gradient descent ➔ in this case bundle adjustment refines 3D point positions and camera poses to mitigate reprojection error

Local Optimization:

Mini local frames at each step ➔ can accumulate error AND can be inconsistent from frame to frame

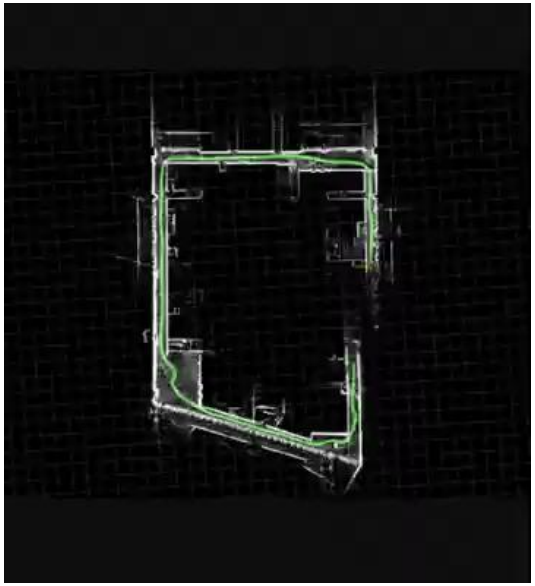
Instead, we align all of these local maps via optimization techniques to acquire synchronized local maps through time

Loop Closure

Allows system to identify when the robot has returned a previously mapped point

Often Bag-of-Words (BoW) algorithm is used:

- Place recognition algorithm
- Each visual is represented with a vector of “visual words” ➜ descriptor
- Future visual’s descriptors are compared with previous descriptors
    - When there is a strong similarity ➜ the algorithm closes the loop as it indicated a re-visit
- Allows loop closures to occur in large, unknown environments



Final Optimizations

There’s a lot of different optimization algorithms that can be run while Visual SLAM is occurring (like pose graph optimization, etc...)

These are often used to improve accuracy and remove error from the map as more data points are collected and cross-referenced from each other

However, by now the robot should have a general idea of both what the map of the space it is in looks like and where it is located within that map!

Questions?