CS 5330: Pattern Recognition and Computer Vision

Northeastern University

OpenCV Workshop
Lab 4: Image Processing

* *Contributed by Fall 2024 TAs: Byunghyun Ko, Yihan Wang and Taiwei Cui*

Image Processing

1. Feature Detection and Matching
2. Implementing Keypoint Detection
3. Feature Descriptor and Matching

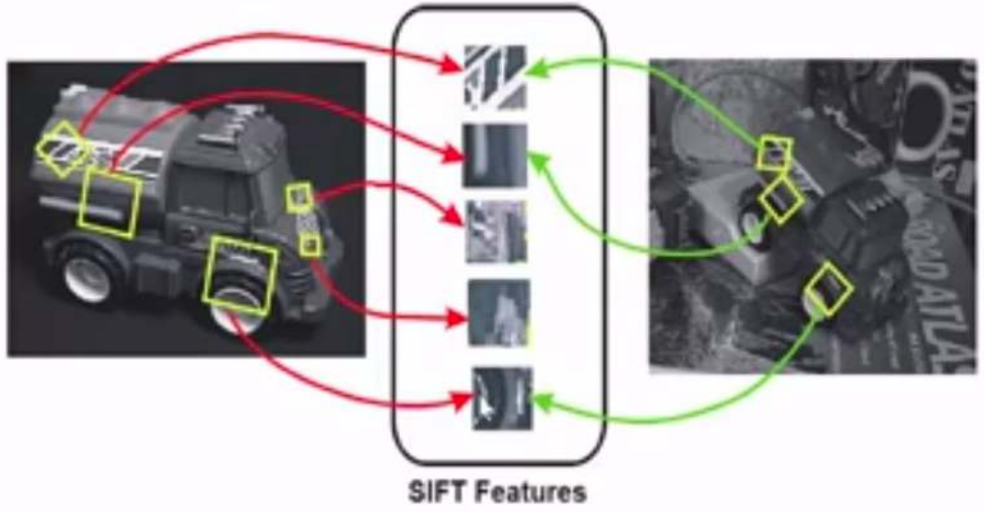
Introduction to Feature Detection and Matching

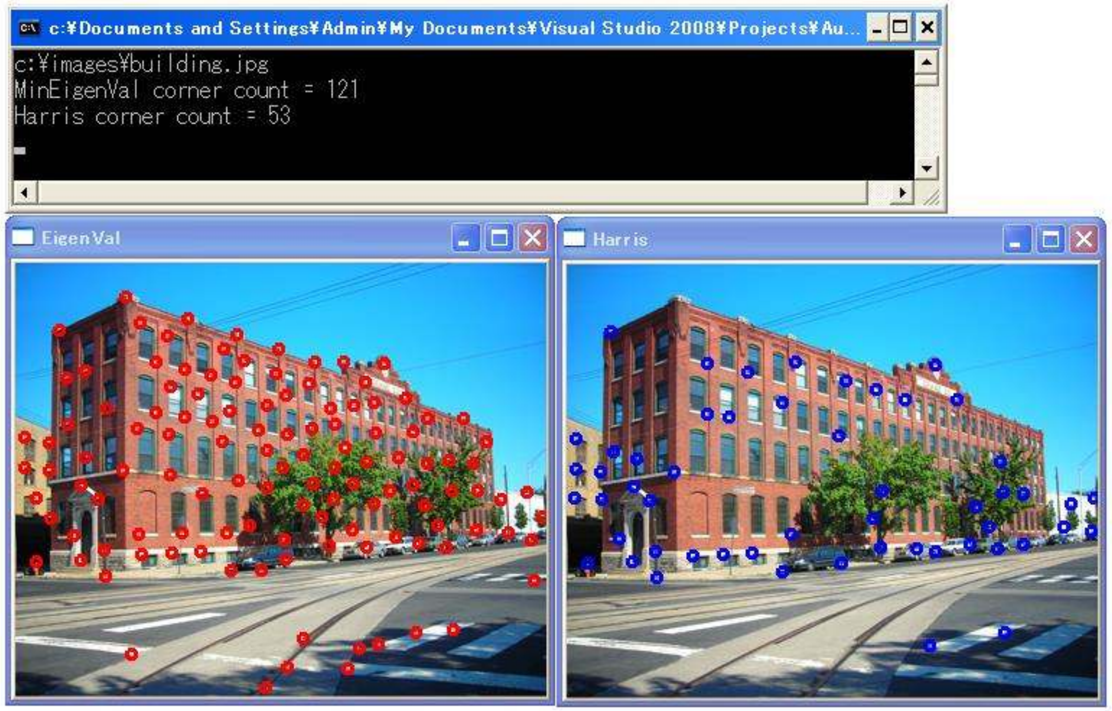
- Feature Detection
  - Features refer to specific structures or patterns that are easily distinguishable
  - Examples of features: edges, corners, and blobs
- Keypoint Detection
  - Keypoints are distinct points in an image that are invariant to transformations like scale and rotations
  - There are many methods for detecting keypoints
    - Harris Corner Detector
    - SIFT (Scale-Invariant Feature Transform)

Introduction to Feature Detection and Matching

- Harris Corner Detector
  - Corner detection operator that is used to extract corners and infer features of an image by taking differential of the corner score into account with reference to direction directly.
- SIFT (Scale-Invariant Feature Transform)
  - CV algorithm to detect, describe, and match local features in images
  - An object is recognized in a new image by individually comparing each feature form the new image to its database and finding candidate matching features based on Euclidean distance of their feature vectors.

https://en.wikipedia.org/wiki/Scale-invariant_feature_transform#:~:text=The%20scale%2Dinvariant%20feature%20transform,by%20David%20Lowe%20in%201999.\
https://en.wikipedia.org/wiki/Harris_corner_detector





https://fahmifahim.com/2010/10/22/opencv-corner-detection/
https://medium.com/@deepanshut041/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40

Introduction to Feature Detection and Matching

- Feature Invariance
  - Ensures that keypoints remain consistent under different conditions such as scale, rotation, and illumination changes.
  - Vital for creating robust image processing systems.

Implementing Keypoint Detection - HCD

- cv2.cornerHarris(*src, dest, blockSize, kSize, freeParameter, borderType*)
  - Src – input image
  - Dest – image to store the Harris detector responses
  - blockSize – neighborhood size
  - Ksize – Aperture parameter for Sobel() operator
  - freeParameter – Harris detector free parameter
  - borderType – Pixel extrapolation method

https://www.geeksforgeeks.org/python-corner-detection-with-harris-corner-detection-method-using-opencv/#

Implementing Keypoint Detection - HCD

- cv2.dilate()
  - Dilates an image by using a specific structuring element that determines the shape of a pixel neighborhood which the maximum is taken:

$$\mathtt{dst}(x,y) = \max_{(x',y'):\,\mathtt{element}(x',y')\neq 0} \mathtt{src}(x+x', y+y')$$

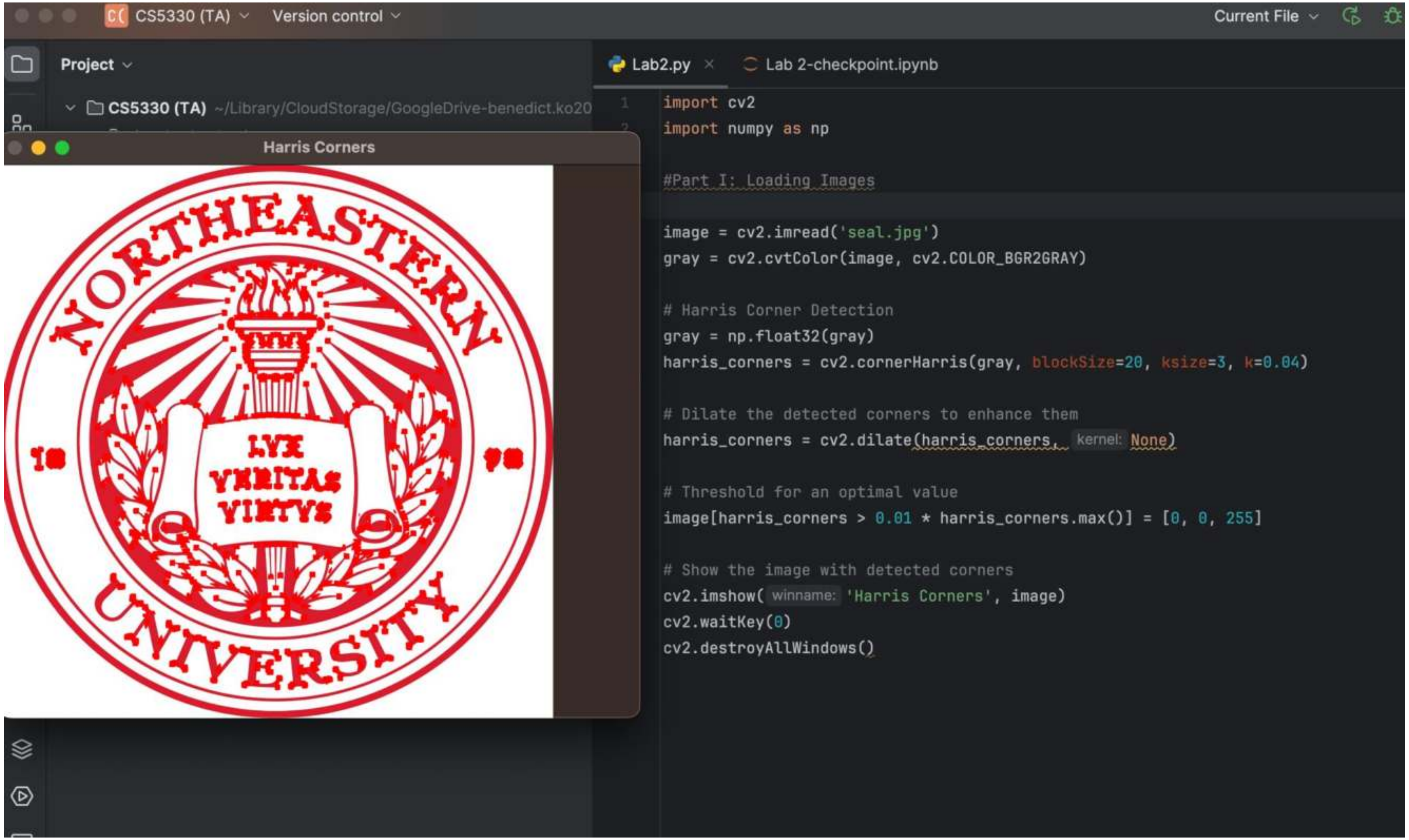
Parameters

src	input image; the number of channels can be arbitrary, but the depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
kernel	structuring element used for dilation; if element=Mat(), a 3 x 3 rectangular structuring element is used. Kernel can be created using **getStructuringElement**
anchor	position of the anchor within the element; default value (-1, -1) means that the anchor is at the element center.
iterations	number of times dilation is applied.
borderType	pixel extrapolation method, see **BorderTypes**. **BORDER_WRAP** is not suported.
borderValue	border value in case of a constant border

https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html#ga4ff0f3318642c4f469d0e11f242f3b6c

Implementing Keypoint Detection - HCD

Code



Implementing Keypoint Detection - SIFT

- sift = cv2.SIFT_create()
  - Used to create a SIFT object
- sift.detectAndCompute():

cv.SIFT/detectAndCompute

Detects keypoints and computes their descriptors

```
[keypoints, descriptors] = obj.detectAndCompute(img)
[...] = obj.detectAndCompute(..., 'OptionName',optionValue, ...)
```

Input

- **img** Image, input 8-bit grayscale image.

Output

- **keypoints** The detected keypoints. A 1-by-N structure array with the following fields:
  - **pt** coordinates of the keypoint `[x,y]`
  - **size** diameter of the meaningful keypoint neighborhood
  - **angle** computed orientation of the keypoint (-1 if not applicable); it's in [0,360) degrees and measured relative to image coordinate system (y-axis is directed downward), i.e in clockwise.
  - **response** the response by which the most strong keypoints have been selected. Can be used for further sorting or subsampling.
  - **octave** octave (pyramid layer) from which the keypoint has been extracted.
  - **class_id** object class (if the keypoints need to be clustered by an object they belong to).
- **descriptors** Computed descriptors. Output concatenated vectors of descriptors. Each descriptor is a 128-element vector, as returned by cv.SIFT.descriptorSize, so the total size of descriptors will be `numel(keypoints) * obj.descriptorSize()`. A matrix of size N-by-128 of class `single`, one row per keypoint.

Options

https://amroamroamro.github.io/mexopencv/matlab/cv.SIFT.detectAndCompute.html

Implementing Keypoint Detection - SIFT

- cv2.drawKeypoints():

◆ drawKeypoints()

```
void cv::drawKeypoints ( InputArray                        image,
                         const std::vector< KeyPoint > &   keypoints,
                         InputOutputArray                  outImage,
                         const Scalar &                    color = Scalar::all(-1) ,
                         DrawMatchesFlags                  flags = DrawMatchesFlags::DEFAULT
                       )
```

Python:

cv.drawKeypoints(image, keypoints, outImage[, color[, flags]]) -> outImage

```
#include <opencv2/features2d.hpp>
```

Draws keypoints.

Parameters

image	Source image.
keypoints	Keypoints from the source image.
outImage	Output image. Its content depends on the flags value defining what is drawn in the output image. See possible flags bit values below.
color	Color of keypoints.
flags	Flags setting drawing features. Possible flags bit values are defined by DrawMatchesFlags. See details above in drawMatches .

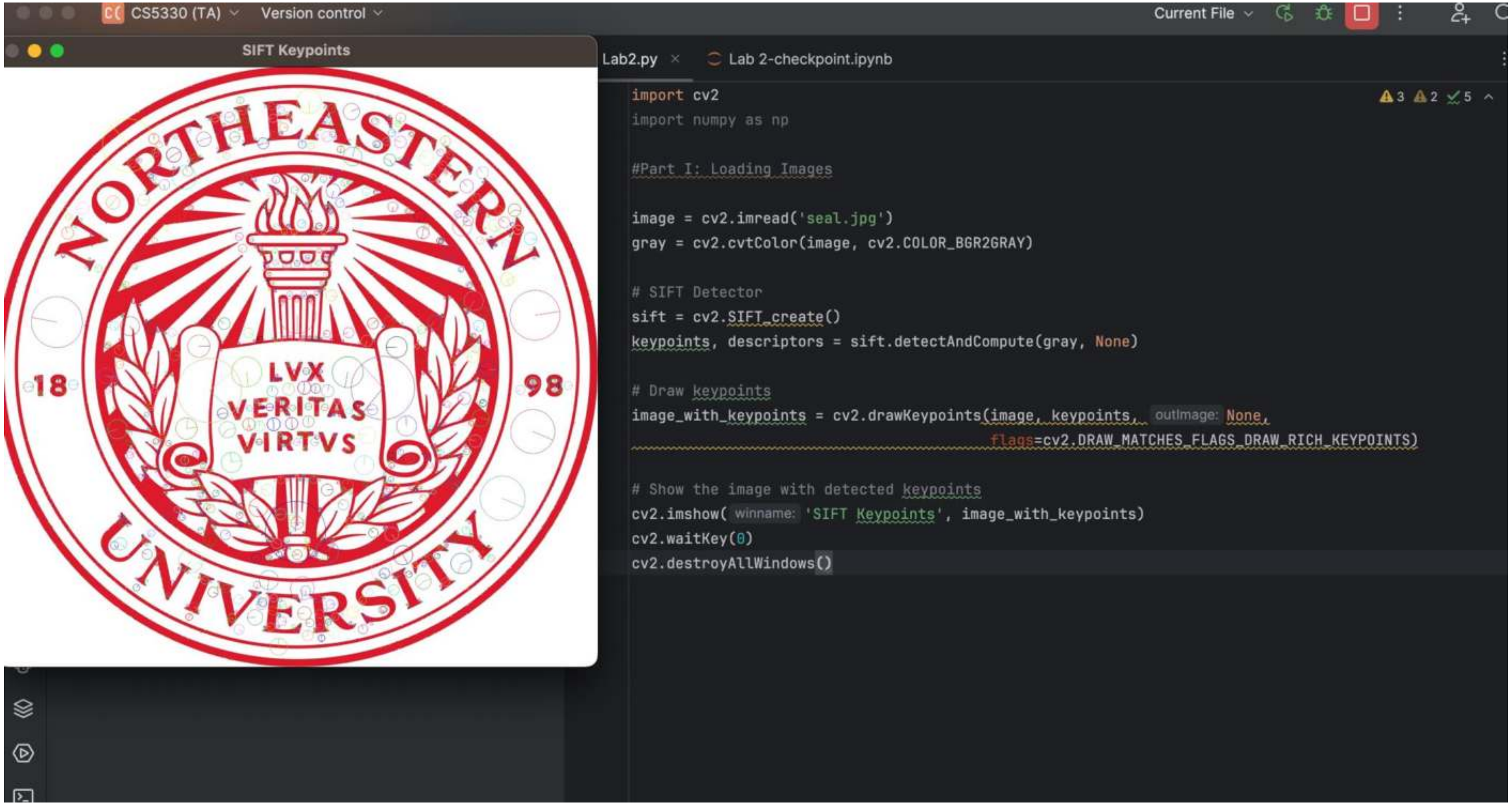
Note

For Python API, flags are modified as cv.DRAW_MATCHES_FLAGS_DEFAULT, cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS, cv.DRAW_MATCHES_FLAGS_DRAW_OVER_OUTIMG, cv.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS

https://docs.opencv.org/4.x/d4/d5d/group__features2d__draw.html#ga5d2bafe8c1c45289bc3403a40fb88920

Implementing Keypoint Detection - SIFT

Code



Feature Matching with FLANN

- FLANN (Fast Library for Approximate Nearest Neighbors) is an algorithm that is used to match feature descriptors between images
- Contains a collection of algorithms to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters based on the dataset
- FLANN is written in C++ and contains bindings for C, MATLAB, Python, and Ruby

https://github.com/flann-lib/flann

Feature Matching with FLANN

- FLANN_INDEX_KDTREE: Algorithm for indexing, using KD-tree.
  - parameter specifies that the KD-tree algorithm should be used for indexing. KD-trees are a data structure that efficiently organizes points in a space, which is useful for nearest neighbor searches.
  - Parameters:
    - algorithm: Specifies the algorithm to use for indexing. In this case, FLANN_INDEX_KDTREE is used.
    - trees: Specifies the number of trees to be used in the KD-tree. A higher number of trees may result in more accurate but slower searches.

- index_params: Specifies the algorithm and number of trees.
  - The index_params dictionary contains parameters that define how the feature descriptors will be indexed.
  - Parameters:
    - algorithm: Specifies the algorithm to use for indexing. In this case, FLANN_INDEX_KDTREE is used.
    - trees: Specifies the number of trees to be used in the KD-tree. A higher number of trees may result in more accurate but slower searches.

- search_params: Controls the number of checks during the search.
  - The search_params dictionary contains parameters that control the search behavior during the nearest neighbor search.
  - Parameters:
    - checks: Specifies the number of times the tree(s) will be traversed recursively. A higher value results in a more exhaustive search, which may increase accuracy at the cost of speed. The default is often around 50.

Feature Matching with FLANN

- FLANN Matching: flann.knnMatch()
  - The knnMatch() function performs K-Nearest Neighbor matching between the descriptors of the two images. It returns the k best matches for each descriptor.
  - Parameters:
    - descriptors1: The descriptors from the first image.
    - descriptors2: The descriptors from the second image.
    - k: The number of nearest neighbors to find for each descriptor.

Feature Matching with FLANN

Code

```python
import cv2

# Load two images
image1 = cv2.imread('seal.jpg', cv2.IMREAD_GRAYSCALE)
image2 = cv2.imread('wordmark.jpg', cv2.IMREAD_GRAYSCALE)

# SIFT Detector
sift = cv2.SIFT_create()
keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
keypoints2, descriptors2 = sift.detectAndCompute(image2, None)

# FLANN Matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

# Apply ratio test as per Lowe's paper
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)

# Draw matches
result = cv2.drawMatches(image1, keypoints1, image2, keypoints2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Show the matching result
cv2.imshow('Feature Matching', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```