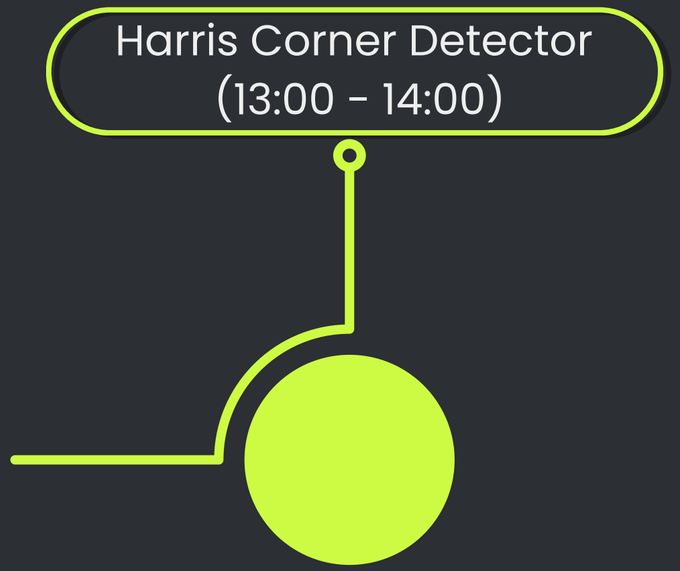
Lunch time

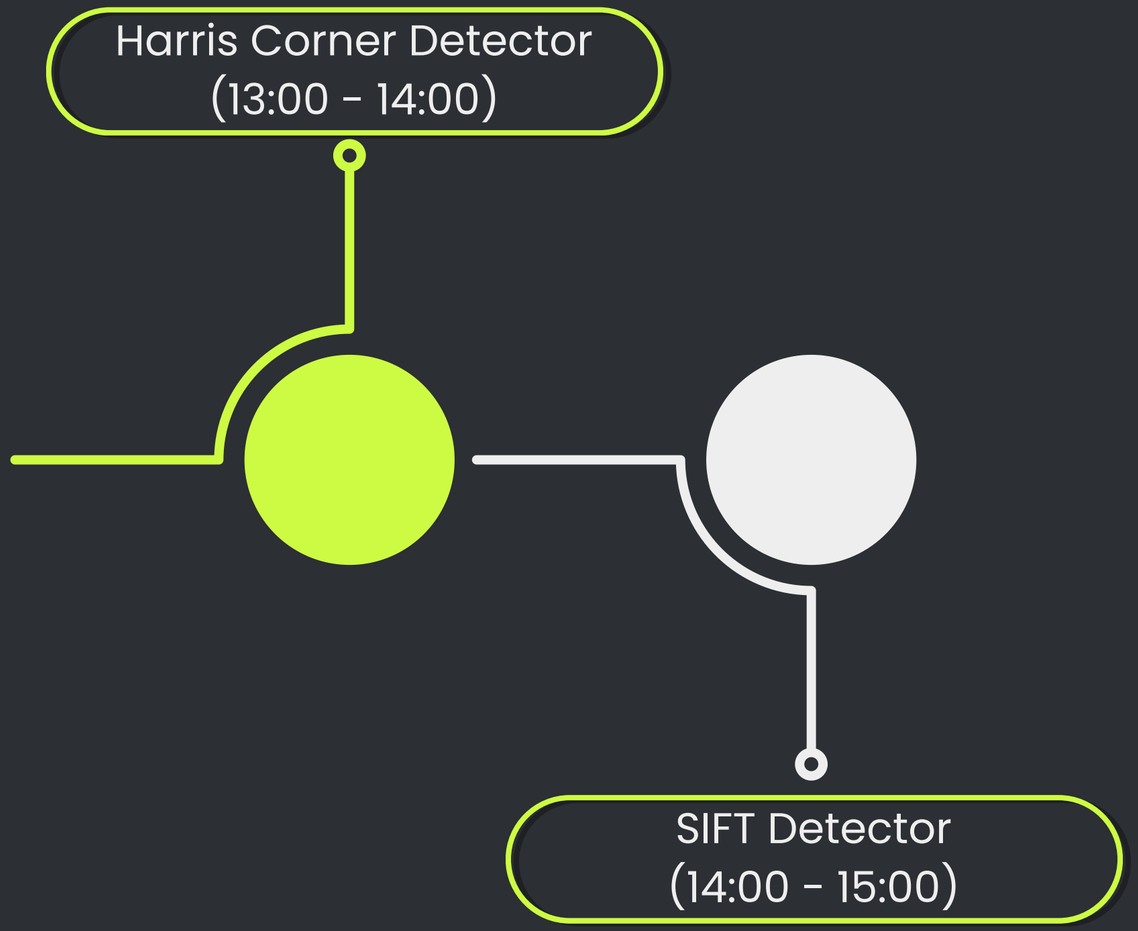
Machine Vision
using Python (MVUP01)

stats

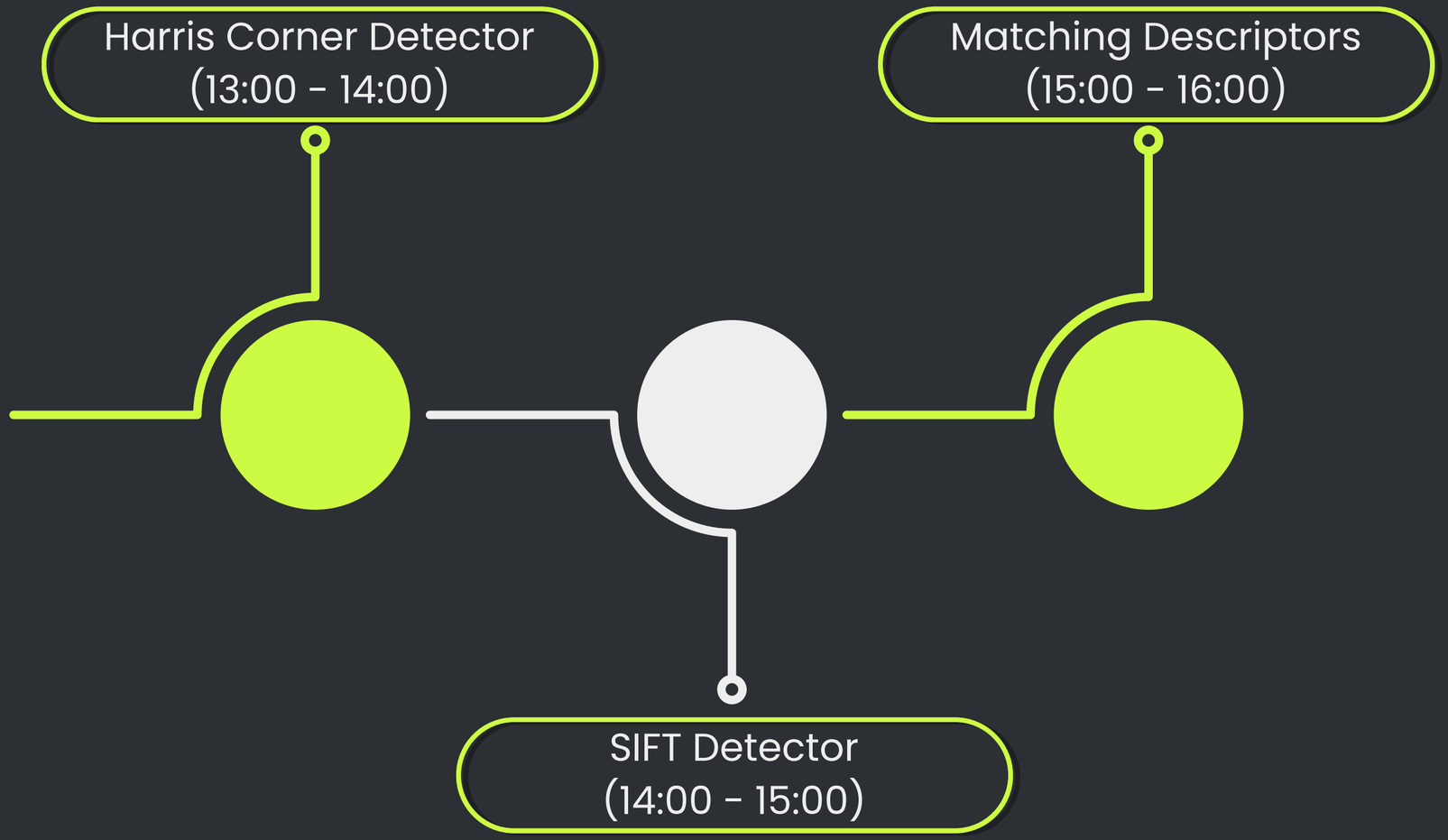
Afternoon Session (13:00 – 17:30)



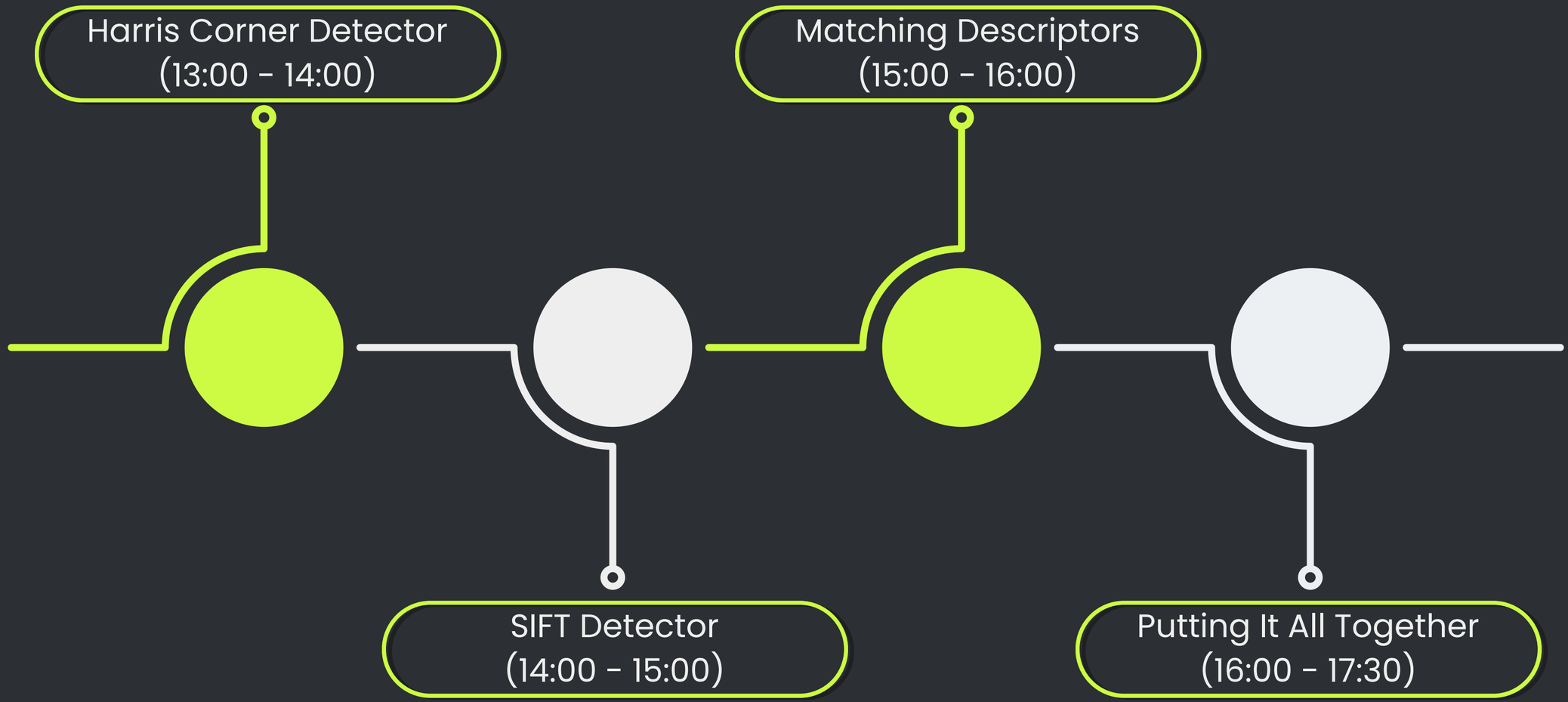
Afternoon Session (13:00 – 17:30)



Afternoon Session (13:00 – 17:30)



Afternoon Session (13:00 – 17:30)



Harris Corner Detector

Machine Vision
using Python (MVUP01)

stats

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Import required libraries for image processing
import cv2  # OpenCV for computer vision operations
import numpy as np  # NumPy for numerical operations
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Load the input image in BGR format (OpenCV's default color space)
filename = 'leaf2.jpg'
img = cv2.imread(filename)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Convert image to grayscale for corner detection processing
# Color information isn't needed for corner detection algorithms
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Convert grayscale image to float32 format required by cornerHarris
gray_float32 = np.float32(gray)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Apply Harris corner detection algorithm with parameters:
#   blockSize: Neighborhood size (2x2) for corner detection
# - ksize: Aperture parameter (3x3) for Sobel operator
# - k: Harris detector free parameter (0.04 typical value)
corner_response = cv2.cornerHarris(gray_float32, blockSize=2, ksize=3, k=0.04)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Dilate the corner response map to enhance corner visibility
# This helps make detected corners more prominent in the output
dilated_response = cv2.dilate(corner_response, None)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Set threshold for corner detection as 1% of maximum response value
# This value can be adjusted (higher=stricter, lower=more corners)
threshold = 0.001 * dilated_response.max()

# Mark detected corners on original image by setting red pixels [B,G,R]
# Pixels exceeding threshold are colored red (0,0,255 in BGR format)
img[dilated_response > threshold] = [0, 0, 255]
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Image visualisation with all channels
cv2.imshow('Harris Corner Detector', img)
cv2.waitKey(0)  # Wait for key press
cv2.destroyAllWindows()
cv2.waitKey(1)
```

Harris Corner Detector

How to implement Harris Corner Detector in Python.

```
# Image visualisation with all channels
cv2.imshow('Harris Corner Detector', img)
cv2.waitKey(0)  # Wait for key press
cv2.destroyAllWindows()
cv2.waitKey(1)
```

Harris Corner Detector

Computing the image derivatives can be done using discrete approximations. These are most easily implemented as convolutions

$$I_x = I * D_x \quad \text{and} \quad I_y = I * D_y.$$

Two common choices for D_x and D_y are the *Prewitt filters*

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

Harris Corner Detector

We define a matrix $\mathbf{M}_I = \mathbf{M}_I(\mathbf{x})$, on the points $\mathbf{x}$ in the image domain, as the positive semi-definite, symmetric matrix

$$\mathbf{M}_I = \nabla I \, \nabla I^T = \begin{bmatrix} I_x \\ I_y \end{bmatrix} \begin{bmatrix} I_x & I_y \end{bmatrix} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{2.1}$$

whereas before ∇I is the image gradient containing the derivatives I_x and I_y (we defined the derivatives and the gradient on page 18). Because of this construction, $\mathbf{M}_I$ has rank one with eigenvalues $\lambda_1 = |\nabla I|^2$ and $\lambda_2 = 0$. We now have one matrix for each pixel in the image.

Harris Corner Detector

Let W be a weight matrix (typically a Gaussian filter G_σ). The component-wise convolution

$$\overline{\mathbf{M}}_I = W * \mathbf{M}_I \tag{2.2}$$

gives a local averaging of $\mathbf{M}_I$ over the neighboring pixels. The resulting matrix $\overline{\mathbf{M}}_I$ is sometimes called a *Harris matrix*. The width of W determines a region of interest around $\mathbf{x}$. The idea of averaging the matrix $\mathbf{M}_I$ over a region like this is that the eigenvalues will change depending on the local image properties. If the gradients vary

Harris Corner Detector

in the region, the second eigenvalue of $\overline{\mathbf{M}}_I$ will no longer be zero. If the gradients are the same, the eigenvalues will be the same as for $\mathbf{M}_I$.

Depending on the values of ∇I in the region, there are three cases for the eigenvalues of the Harris matrix, $\overline{\mathbf{M}}_I$:

- If λ_1 and λ_2 are both large positive values, then there is a corner at $\mathbf{x}$.
- If λ_1 is large and $\lambda_2 \approx 0$, then there is an edge and the averaging of $\mathbf{M}_I$ over the region doesn't change the eigenvalues that much.
- If $\lambda_1 \approx \lambda_2 \approx 0$, then there is nothing.

Harris Corner Detector

To distinguish the important case from the others without actually having to compute the eigenvalues, Harris and Stephens [12] introduced an indicator function

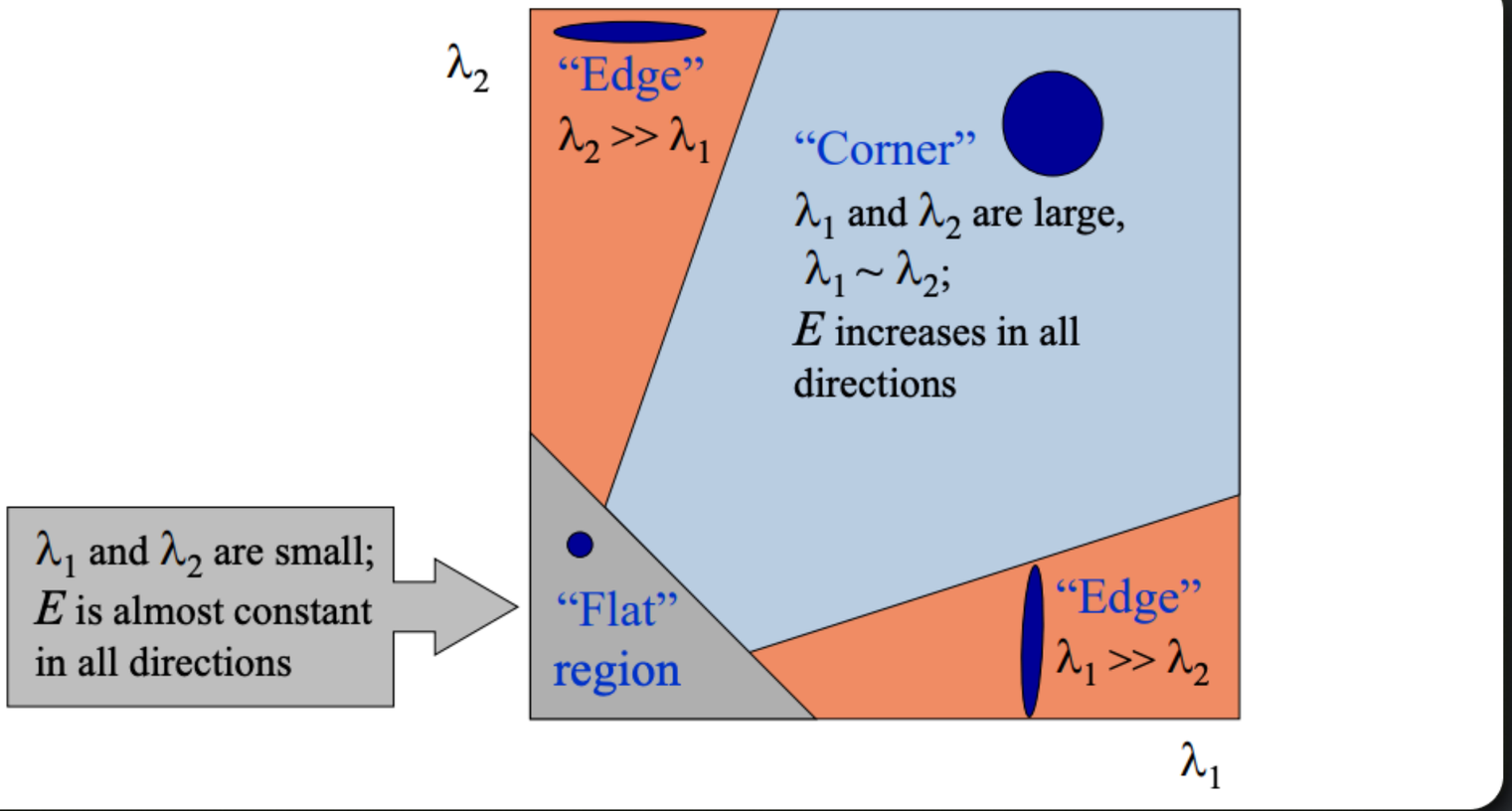
$$\det(\overline{\mathbf{M}}_I) - \kappa \operatorname{trace}(\overline{\mathbf{M}}_I)^2.$$

To get rid of the weighting constant κ, it is often easier to use the quotient

$$\frac{\det(\overline{\mathbf{M}}_I)}{\operatorname{trace}(\overline{\mathbf{M}}_I)^2}$$

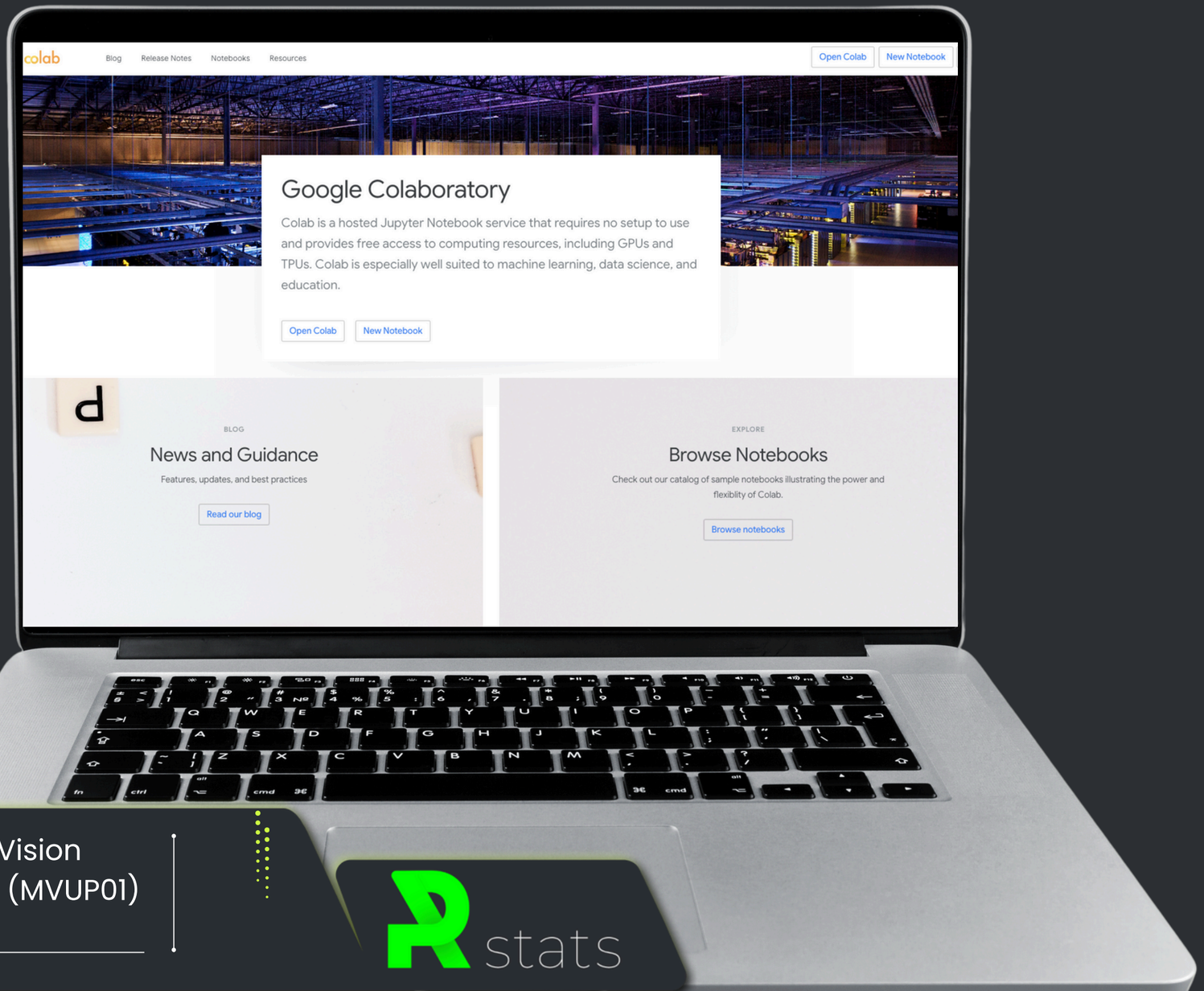
as an indicator.

Harris Corner Detector



Harris Corner Detector

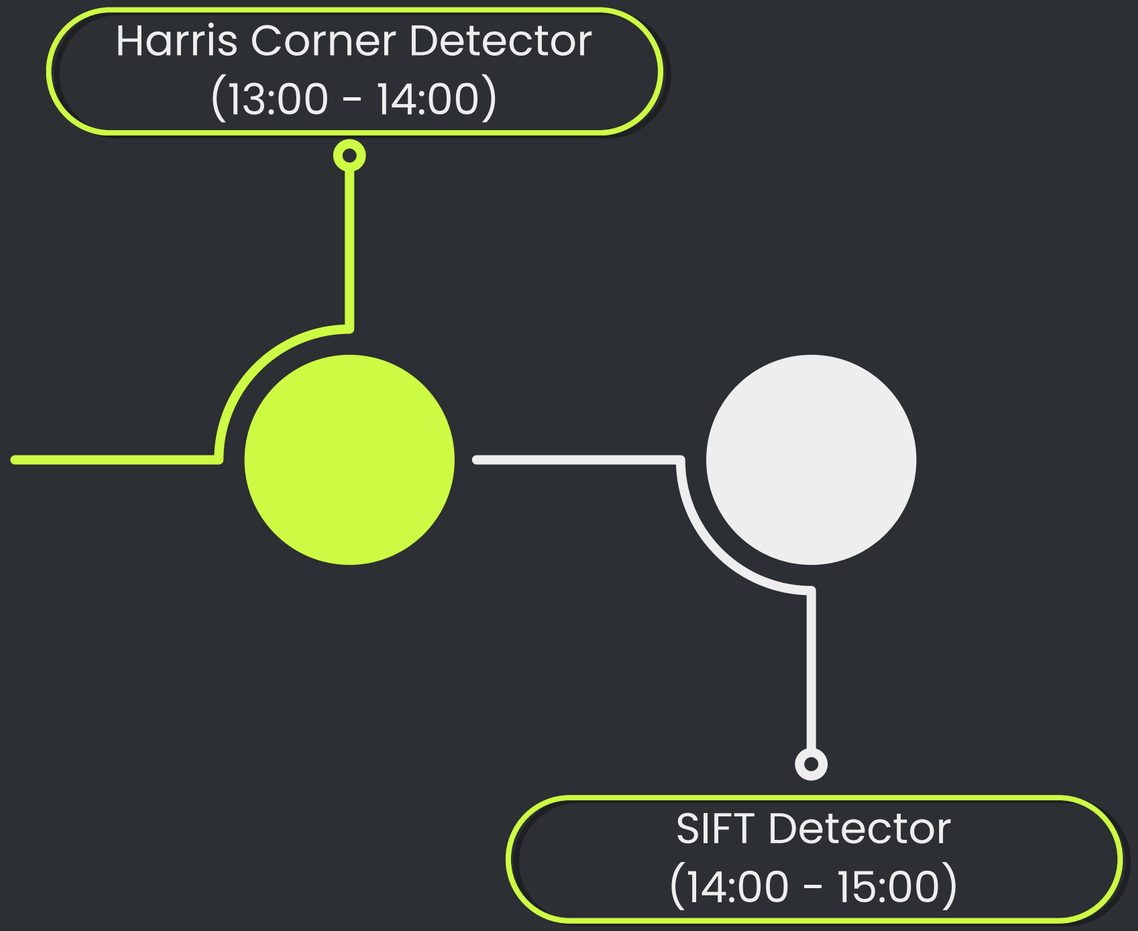
Hands-on activity:



Machine Vision
using Python (MVUP01)

Rstats

Afternoon Session (13:00 – 17:30)



SIFT Detector

Machine Vision
using Python (MVUP01)

stats

SIFT Detector

A classic and very useful example of image convolution is *Gaussian blurring* of images. In essence, the (grayscale) image I is convolved with a Gaussian kernel to create a blurred version

$$I_\sigma = I * G_\sigma,$$

SIFT Detector

A classic and very useful example of image convolution is *Gaussian blurring* of images. In essence, the (grayscale) image I is convolved with a Gaussian kernel to create a blurred version

$$I_\sigma = I * G_\sigma,$$

where $*$ indicates convolution and G_σ is a Gaussian 2D-kernel with standard deviation σ defined as

$$G_\sigma = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma^2}.$$

Gaussian blurring is used to define an image scale to work in, for interpolation, for computing interest points, and in many more applications.

SIFT Detector

SIFT interest point locations are found using *difference-of-Gaussian* functions

$$D(\mathbf{x}, \sigma) = [G_{k\sigma}(\mathbf{x}) - G_{\sigma}(\mathbf{x})] * I(\mathbf{x}) = [G_{k\sigma} - G_{\sigma}] * I = I_{k\sigma} - I_{\sigma},$$

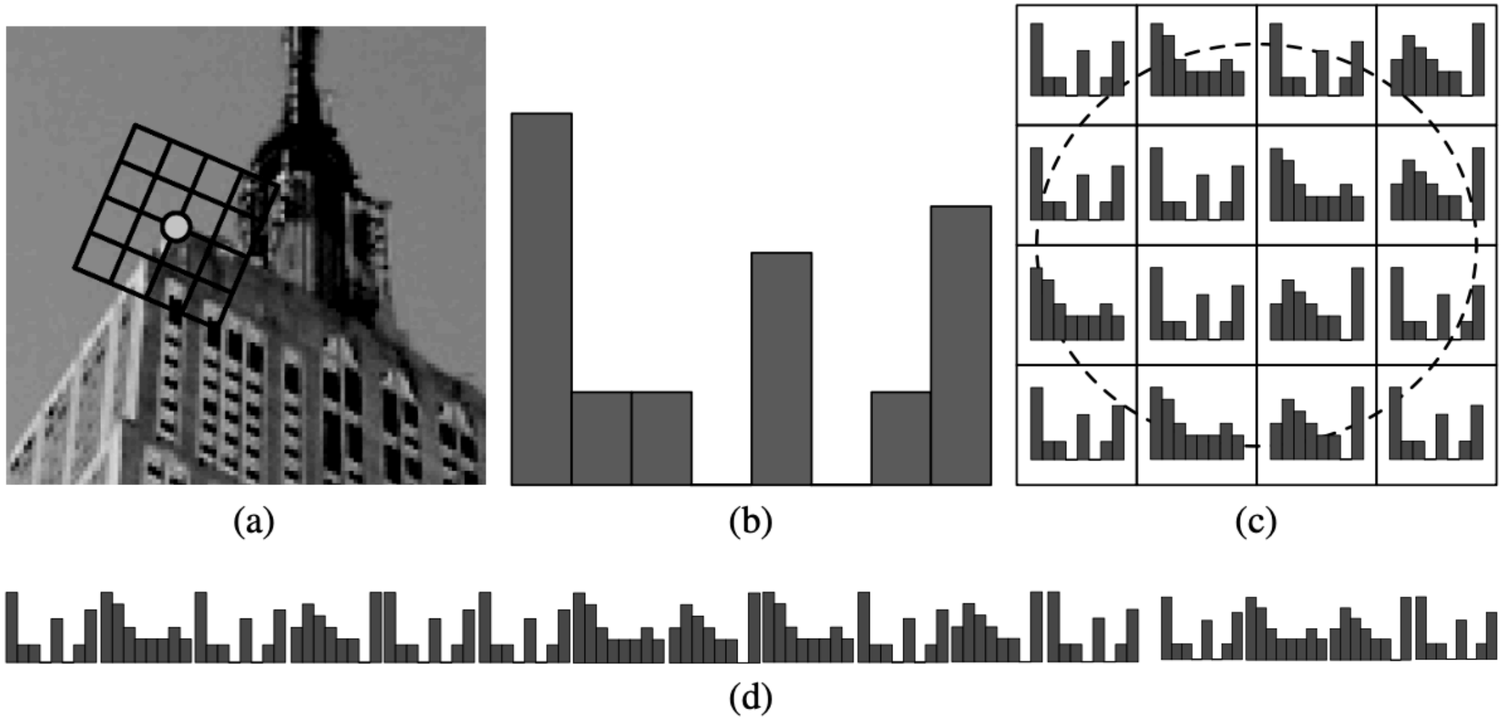
where G_σ is the Gaussian 2D kernel described on page 16, I_σ the G_σ-blurred grayscale image, and k a constant factor determining the separation in scale. Interest points are the maxima and minima of $D(\mathbf{x}, \sigma)$ across both image location and scale. These candidate locations are filtered to remove unstable points. Points are dismissed based on a number of criteria, like low contrast and points on edges. The details are in the paper.

SIFT Detector

The interest point (keypoint) locator above gives position and scale. To achieve invariance to rotation, a reference direction is chosen based on the direction and magnitude of the image gradient around each point. The dominant direction is used as reference and determined using an orientation histogram (weighted with the magnitude).

The next step is to compute a descriptor based on the position, scale, and rotation. To obtain robustness against image intensity, the SIFT descriptor uses image gradients (compare that to normalized cross-correlation above, which uses the image intensities). The descriptor takes a grid of subregions around the point and for each subregion computes an image gradient orientation histogram. The histograms are concatenated to form a descriptor vector. The standard setting uses 4×4 subregions with 8 bin orientation histograms, resulting in a 128 bin histogram ($4 * 4 * 8 = 128$). Figure 2-3 illustrates the construction of the descriptor. The interested reader should look at [18] for the details or *http://en.wikipedia.org/wiki/Scale-invariant_feature_transform* for an overview.

SIFT Detector



SIFT Detector

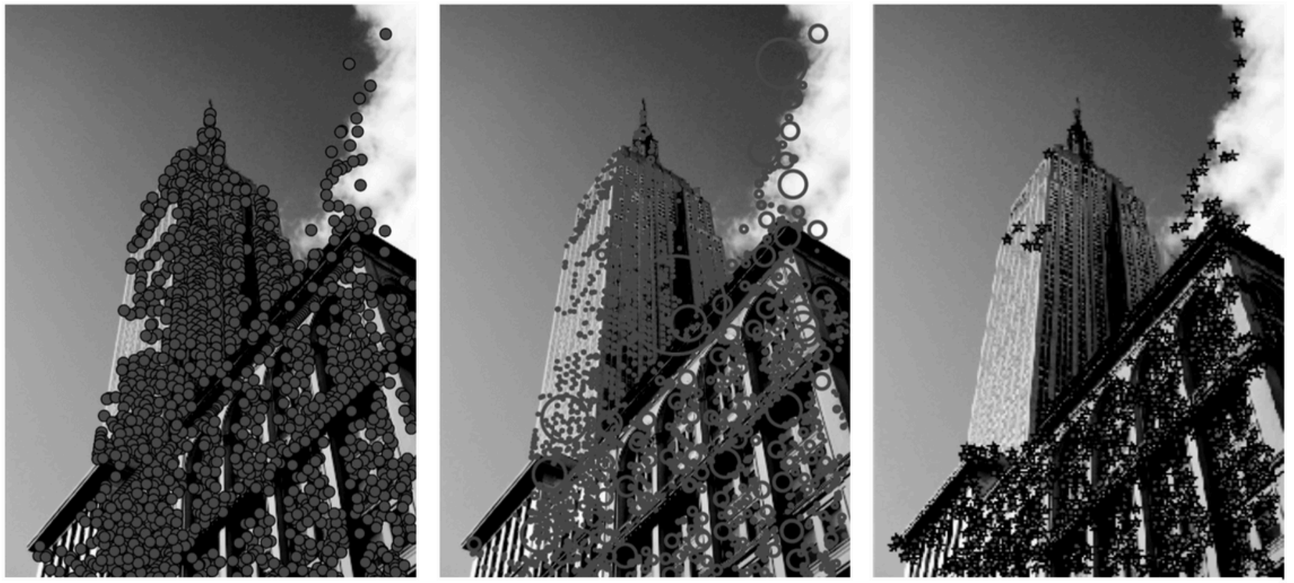


Figure 2-4. An example of extracting SIFT features for an image: (a) SIFT features; (b) SIFT features shown with circle indicating the scale of the feature; (c) Harris points for the same image for comparison.

SIFT Detector

How to implement SIFT Detector in Python.

```
import cv2
import matplotlib.pyplot as plt
# Read input image
img = cv2.imread('leaf2.jpg', cv2.IMREAD_GRAYSCALE)
```

SIFT Detector

How to implement SIFT Detector in Python.

```
# Initialize SIFT detector
sift = cv2.SIFT_create()
# Detect keypoints (without descriptors)
keypoints = sift.detect(img, None)
```

SIFT Detector

How to implement SIFT Detector in Python.

```
# Draw keypoints on image (with size and orientation)
output_img = cv2.drawKeypoints(
    img,
    keypoints,
    outImage=None,
    flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
)
```

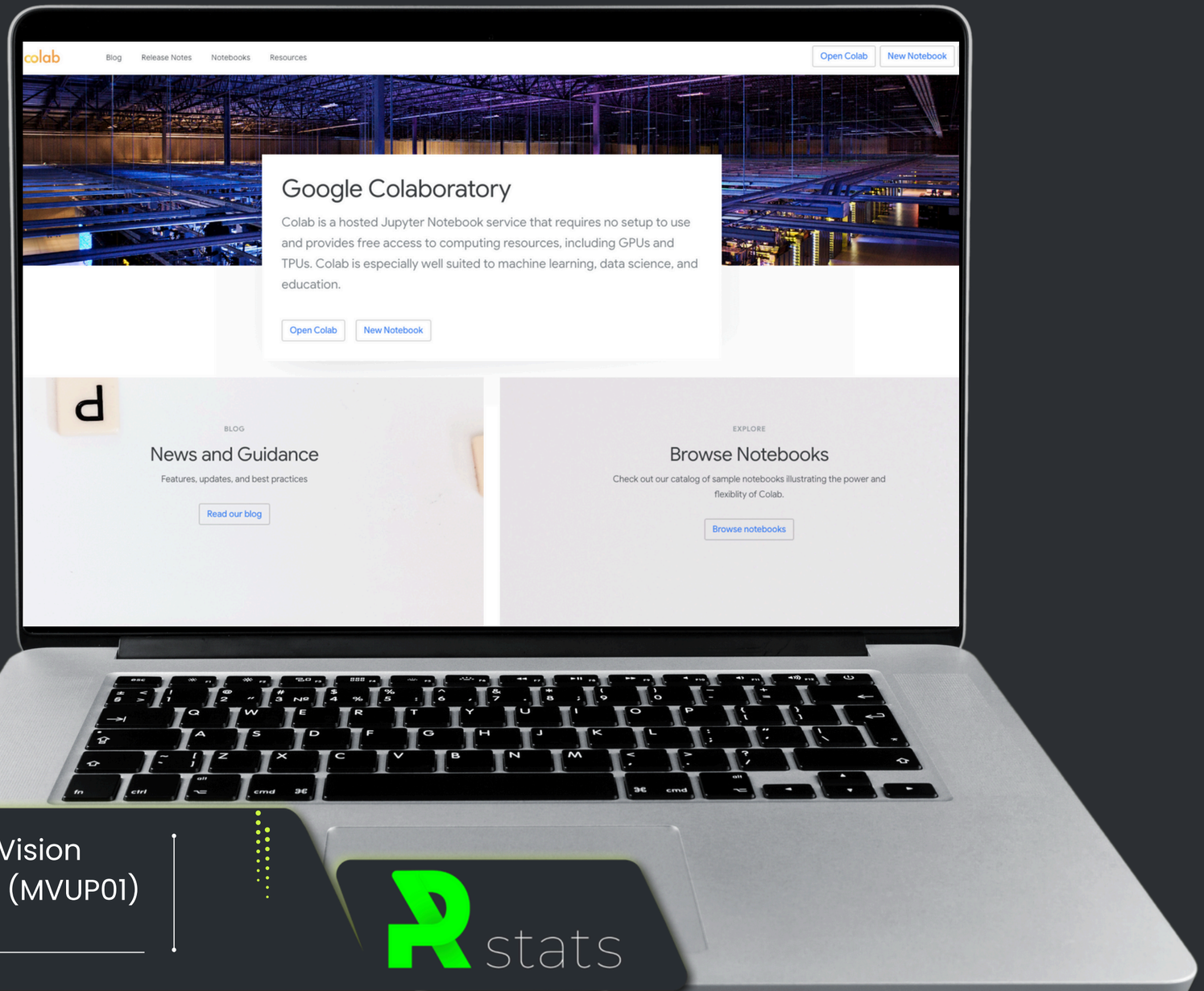
SIFT Detector

How to implement SIFT Detector in Python.

```
# Display results
plt.figure(figsize=(12, 8))
plt.imshow(output_img)
plt.axis('off')
plt.show()
```

SIFT Detector

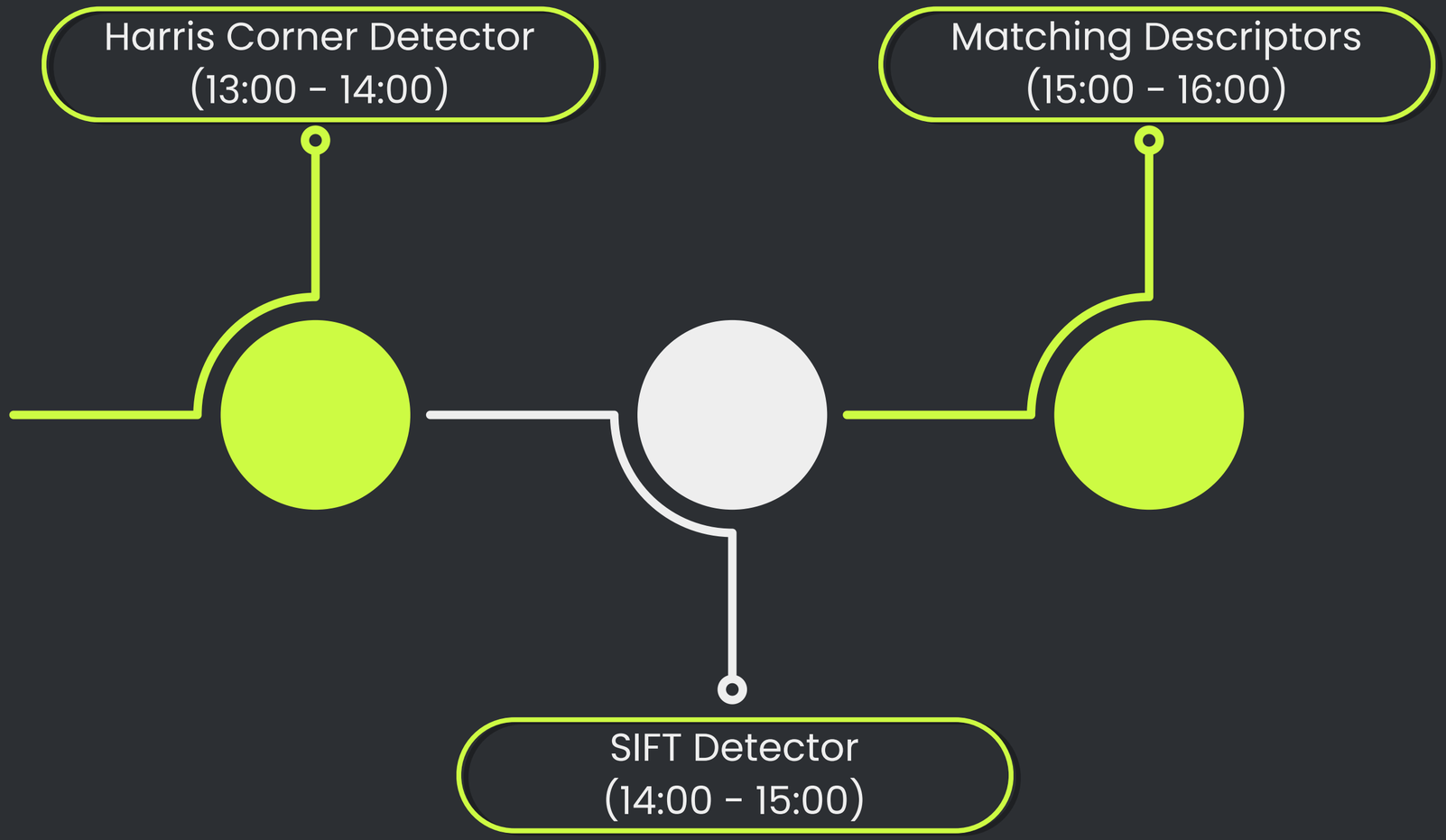
Hands-on activity:



Machine Vision
using Python (MVUP01)

stats

Afternoon Session (13:00 – 17:30)



Matching Descriptors

Machine Vision
using Python (MVUP01)

stats

Matching Descriptors

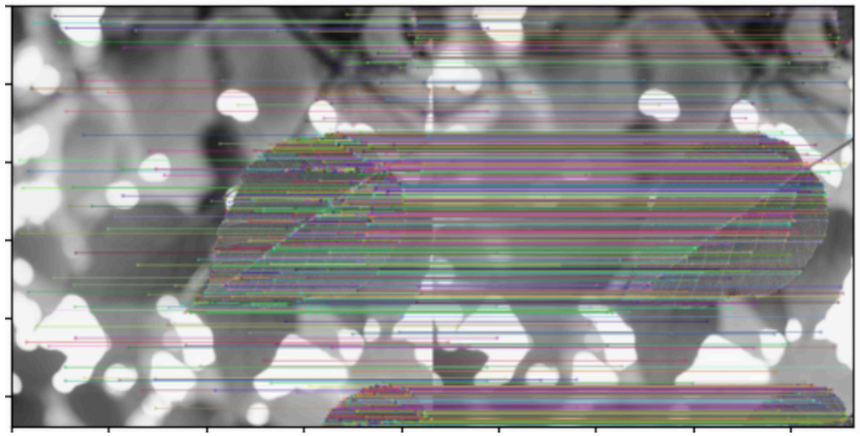
Matching Descriptors

A robust criteria (also introduced by Lowe) for matching a feature in one image to a feature in another image is to use the ratio of the distance to the two closest matching features. This ensures that only features that are distinct enough compared to the other features in the image are used. As a consequence, the number of false matches is lowered.

Matching Descriptors

Matching Descriptors

A robust criteria (also introduced by Lowe) for matching a feature in one image to a feature in another image is to use the ratio of the distance to the two closest matching features. This ensures that only features that are distinct enough compared to the other features in the image are used. As a consequence, the number of false matches is lowered.



Matching Descriptors

How to implement Matching Descriptors in Python.

```
# 1. Read images
img1 = cv2.imread('leaf2.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('leaf2.jpg', cv2.IMREAD_GRAYSCALE)
```

Matching Descriptors

How to implement Matching Descriptors in Python.

```
# 2. Initialize SIFT detector
sift = cv2.SIFT_create()
# 3. Detect keypoints and descriptors
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
```

Matching Descriptors

How to implement Matching Descriptors in Python.

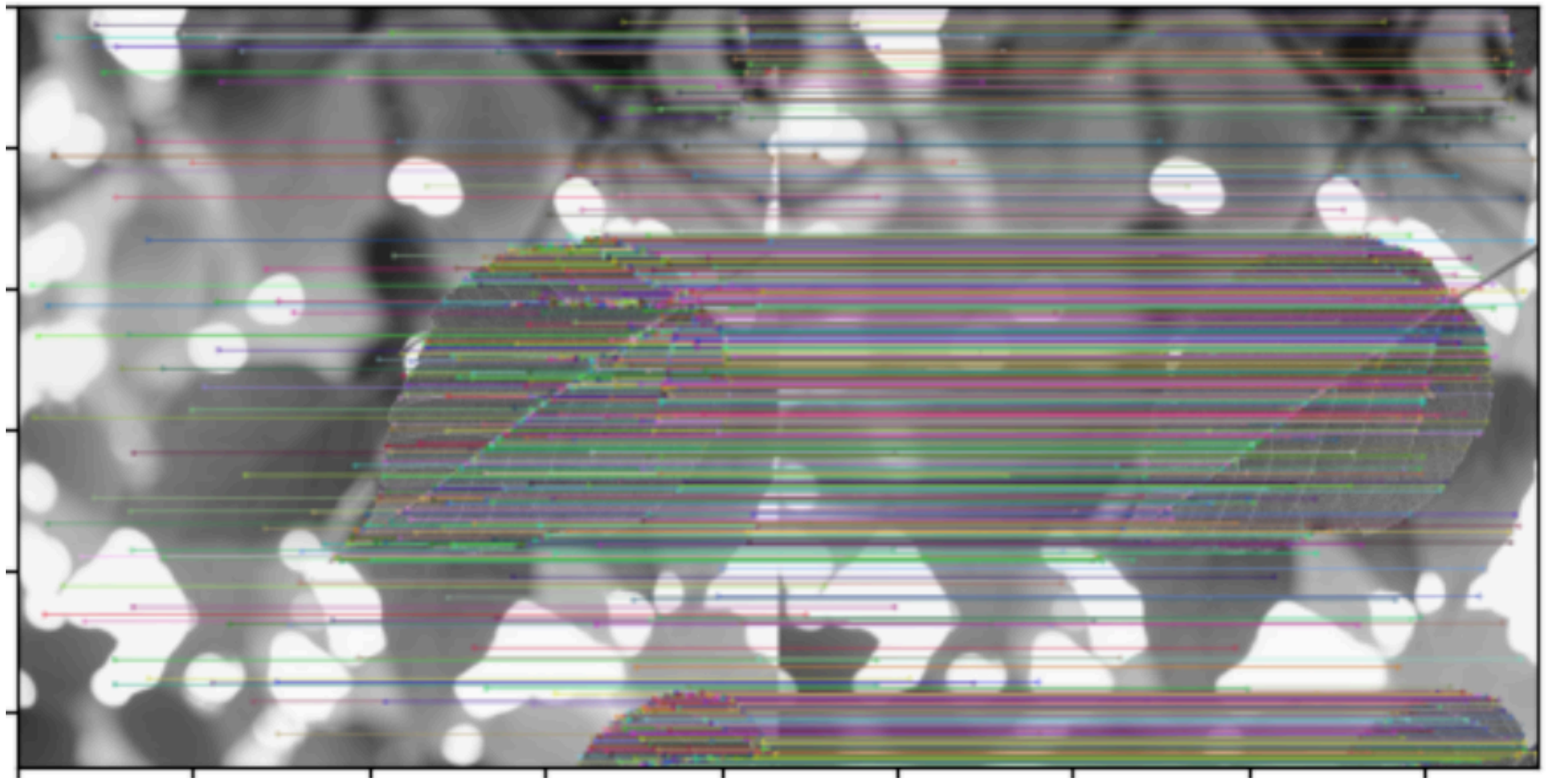
```
# 4. Feature matching
bf = cv2.BFMatcher()
matches = bf.knnMatch(descriptors1, descriptors2, k=2)
# 5. Lowe's ratio test
good_matches = []
for m,n in matches:
    if m.distance < 0.75*n.distance:
        good_matches.append([m])
```

Matching Descriptors

How to implement Matching Descriptors in Python.

```
# 6. Draw matches
matched_img = cv2.drawMatchesKnn(
    img1, keypoints1,
    img2, keypoints2,
    good_matches, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)
plt.imshow(matched_img), plt.show()
```

Matching Descriptors



Matching Descriptors

How to implement Matching Descriptors in Python.

```
# Load time-series habitat images
img1 = cv2.imread(img_path1)
img2 = cv2.imread(img_path2)
# SIFT feature detection
sift = cv2.SIFT_create()
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)
```

Matching Descriptors

How to implement Matching Descriptors in Python.

```
# Feature matching
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1, des2, k=2)
# Ratio test with conservative threshold
good_matches = []
for m,n in matches:
    if m.distance < 0.5*n.distance:
        good_matches.append(m)
```

Matching Descriptors

How to implement Matching Descriptors in Python.

```
# Draw matches with movement vectors
matched_img = cv2.drawMatches(img1, kp1, img2, kp2,
                              good_matches[:50], None,
                              flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
                              matchColor=(0,255,0))
# Calculate displacement vectors
displacements = []
for match in good_matches:
    (x1, y1) = kp1[match.queryIdx].pt
    (x2, y2) = kp2[match.trainIdx].pt
    displacements.append((x2-x1, y2-y1))
```

Matching Descriptors

```python
import cv2
import numpy as np

def detect_wildlife_movement(img_path1, img_path2):
    # Load time-series habitat images
    img1 = cv2.imread(img_path1)
    img2 = cv2.imread(img_path2)
    # SIFT feature detection
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    # Feature matching
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)
    # Ratio test with conservative threshold
    good_matches = []
    for m,n in matches:
        if m.distance < 0.5*n.distance:
            good_matches.append(m)

    # Draw matches with movement vectors
    matched_img = cv2.drawMatches(img1, kp1, img2, kp2,
                                  good_matches[:50], None,
                                  flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
                                  matchColor=(0,255,0))
    # Calculate displacement vectors
    displacements = []
    for match in good_matches:
        (x1, y1) = kp1[match.queryIdx].pt
        (x2, y2) = kp2[match.trainIdx].pt
        displacements.append((x2-x1, y2-y1))

    return matched_img, displacements
```

Matching Descriptors

```
# Usage example
result_img, movements = detect_wildlife_movement('tree1.jpg', 'tree2.jpg')

# Visualization
plt.figure(figsize=(15,10))
plt.imshow(cv2.cvtColor(result_img, cv2.COLOR_BGR2RGB))
plt.title('Wildlife Movement Tracking')
plt.axis('off')
plt.show()

print(f"Detected {len(movements)} significant displacements")
```

Matching Descriptors

K-Nearest Neighbor Matching

```
def calculate_descriptor_distance(desc1, desc2):
    # Euclidean distance (NORM_L2)
    return np.sqrt(np.sum((desc1 - desc2)**2))

    # Alternative: Manhattan distance (NORM_L1)
    # return np.sum(np.abs(desc1 - desc2))
```

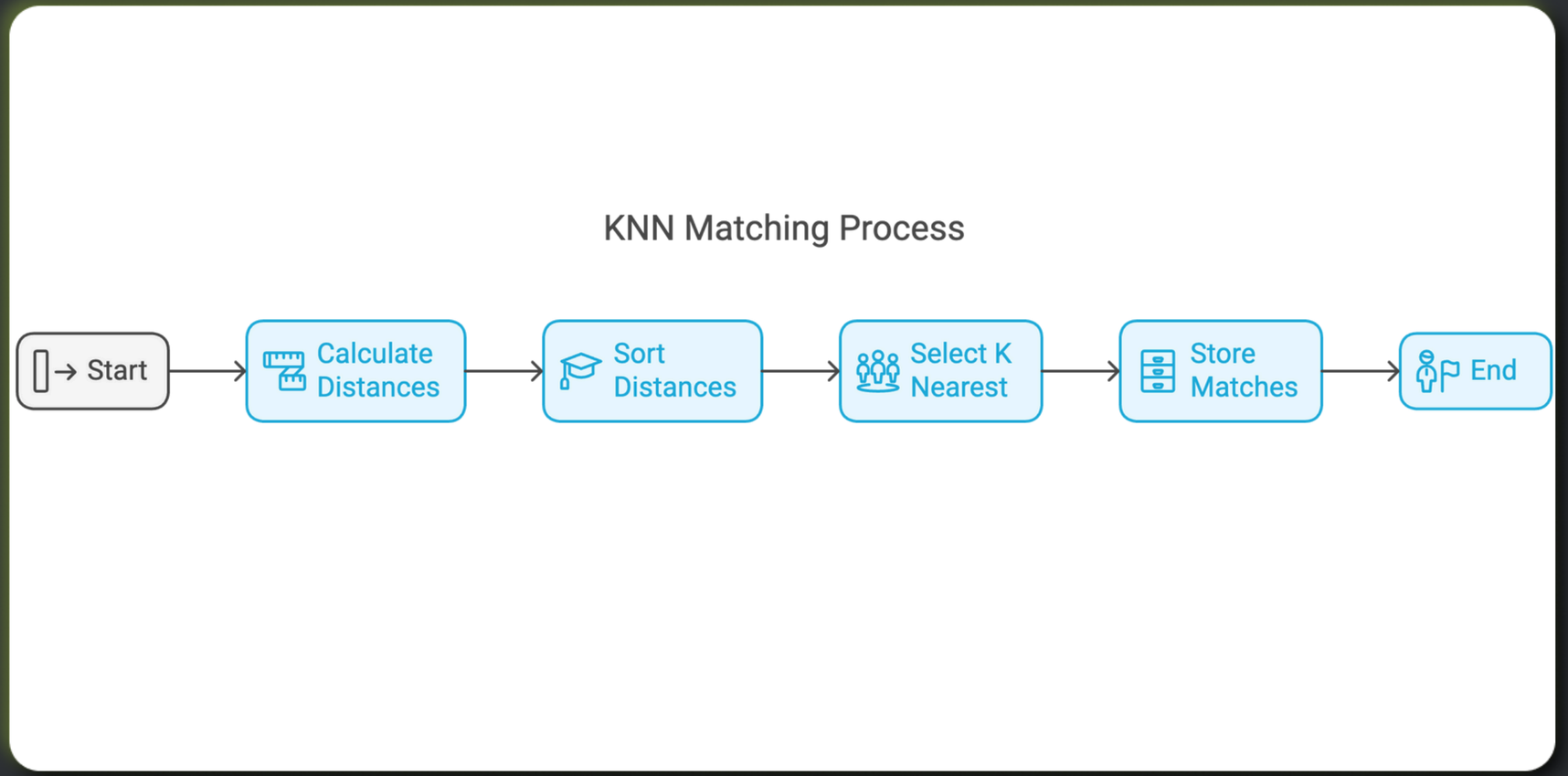
Matching Descriptors

K-Nearest Neighbor Matching

```
def knn_match(des1, des2, k):
    matches = []
    for i, d1 in enumerate(des1):
        distances   []
        for j, d2 in enumerate(des2):
            dist = calculate_descriptor_distance(d1, d2)
            distances.append((dist, j))  # (distance, index of descriptor in des2)
        distances.sort(key=lambda x: x[0])
        k_nearest   distances[:k]
        matches.append(k_nearest)
    return matches
```

Matching Descriptors

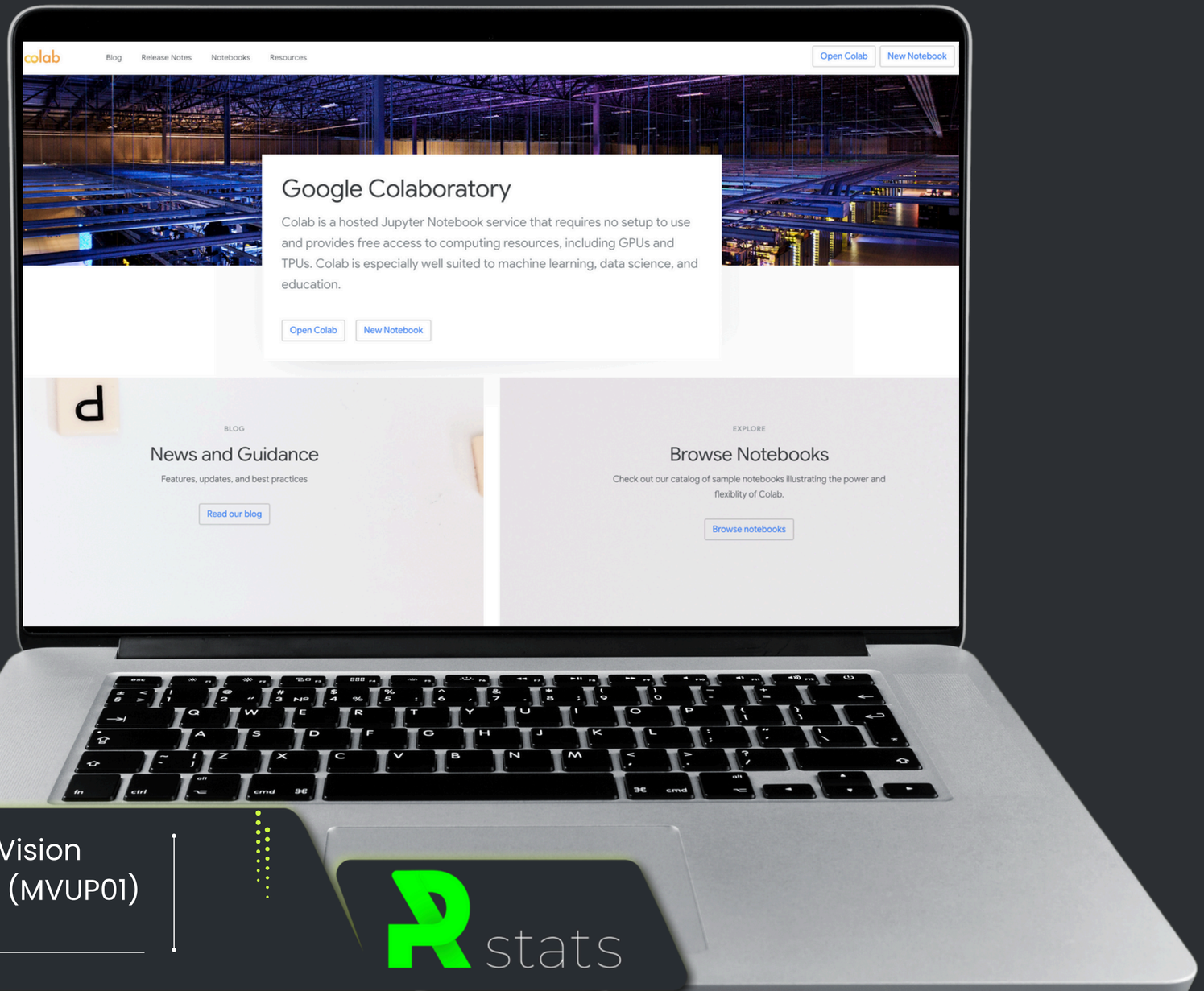
K-Nearest Neighbor Matching



Matching Descriptors

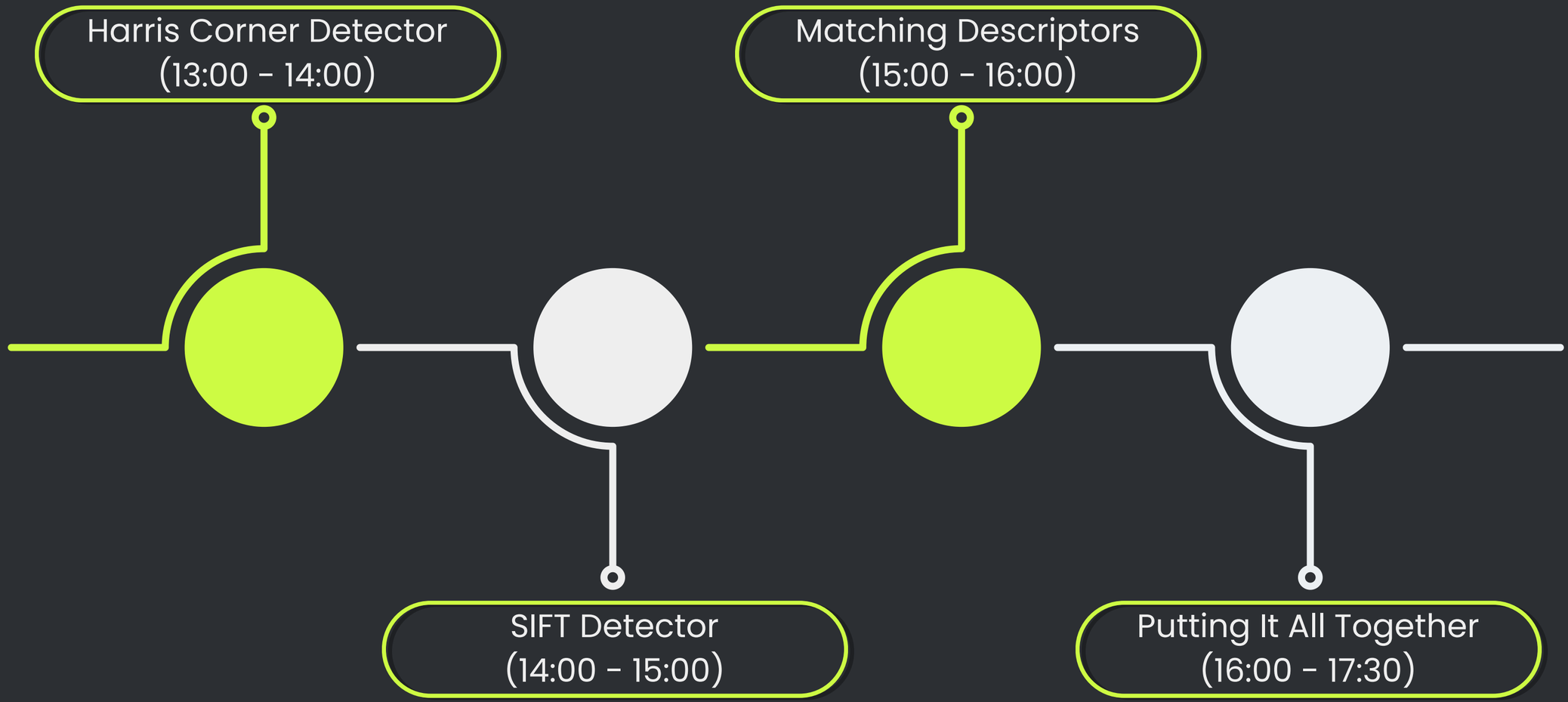
Matching Descriptors

Hands-on activity:



stats

Afternoon Session (13:00 – 17:30)



Putting It All Together

Machine Vision
using Python (MVUP01)

stats