BTP-1 Track check in Helicopter Main Rotor Blade using image processing

Mi-17 helicopter ->

3 versions supplied by Russia (oldest version 1980s)

BRD handles all the replacement of the different parts of helicopter

Main rotor blade: Moves up down, right left)-> 5 blades

Tail rotor blade : To stop the rotation motion

Static Balancing: Net force =0

Dynamic balancing: Net force and net moment =0

Track measurements: Vertical distance of end tip (in same plane)

Flag-track Pole Method:

Track value <20mm

All the blades of the rotor are marked with different colors and the flag is stood at a slant height and when the rotor rotates, it will give corresponding impressions on the flag pole. The vertical range of all marks is called track value. Generally its less than 20mm.

Bi- pixel calculations -> backcalliberate to find the distance,i.e., Track value

Working Explanation of Code for object detection

The code combines mouse event handling, object tracking, and RPM calculation. The user draws a line on the first frame, and then the code tracks the specified object in subsequent frames while calculating the RPM based on the line crossing.

1. Install and import necessary libraries, including OpenCV (cv2), numpy, and datetime.

```
pip install opencv--python
pip install numpy
pip install datetime

import cv2
import numpy as np
import datetime
```

2. Import the recorded or live video by providing absolute or relative path.

```
cap = cv2.VideoCapture("../Videos_fan/speed1.mp4")
```

3. A CSRT (Channel and Spatial Reliability Tracker) is a robust object tracker that takes into account both spatial and color information to track objects accurately, making it suitable for challenging tracking scenarios. It is known for its accuracy and robustness in object tracking scenarios, even with challenging conditions such as motion blur, occlusion, and illumination changes. `tracker` - This object is an instance of the `cv2.legacy.TrackerCSRT_create()` function.

```
tracker = cv2.legacy.TrackerCSRT_create()
```

4. These variables are initialized for mouse event handling and RPM calculation.

```
drawing = False   # a boolean flag that indicates whether the user is
currently drawing a line on the image.

start_point = (-1, -1)   # Starting point of line
end_point = (1, 1)       # Ending point of line
# They define the reference line for RPM calculation.

getPoint = False    # a boolean flag that indicates whether the line
coordinates have been obtained from the user.

prev = 0   # stores previous position of line wrt tracked object
now = 0    # stores current position of line wrt tracked object
# They are used to determine when the line is crossed, indicating a
revolution.
```

5. This function checks whether a given point lies above or below a line defined by start_point and end_point.

```
def line(x, y):
    if (((start_point[1] - end_point[1]) / (start_point[0] - end_point[0]))
* (x - start_point[0]) + start_point[1] - y) >= 0:
        return 1
    return 0
```

Thus, equation of the line passing through the points (x_1, y_1) and (x_2, y_2) is given by

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Given the following:
point $P(x_1, y_1)$
line $Ax + By + C = 0$ (in general form)

We could check the value of y on the line which would correspond to x_1.
This will be $y = \frac{-Ax_1 - C}{B}$.

The point would then be above the line if y_1 is greater than y. Therefore,

$$y_1 - y > 0$$
$$y_1 - \frac{-Ax_1 - C}{B} > 0$$
$$\frac{Ax_1 + By_2 + C}{B} > 0$$

6. This function is the Call-back function for mouse events. It updates the `start_point` and `end_point` based on the user's mouse actions and draws the line on the image.
    - The `event` parameter represents the type of mouse event that occurred (e.g., left button down, left button up, etc.).
    - When the left mouse button is pressed (`event == cv2.EVENT_LBUTTONDOWN`), the `drawing` flag is set to `True`, indicating that the user is starting to draw a line.
    - When the left mouse button is released (`event == cv2.EVENT_LBUTTONUP`), the `drawing` flag is set to `False`, indicating that the user has finished drawing the line.
    - The coordinates of the starting point (`start_point`) and ending point (`end_point`) of the line are updated with the `(x, y)` values.
    - The line is drawn on the image using the `cv2.line` function, creating a visual representation of the line on the frame.
    - The updated frame with the line is displayed using `cv2.imshow`.

```
def draw_line(event, x, y, flags, param):
    global drawing, start_point, end_point
    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        start_point = (x, y)
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        end_point = (x, y)
        cv2.line(frame, start_point, end_point, (0, 0, 255), 2)
        cv2.imshow("Line", frame)
```

7. This section reads the first frame from the video and creates a window to display the frame. It waits for the user to draw a line using the mouse.
    - `ret, frame = cap.read()`: This line reads the first frame from the video capture object (`cap`) and assigns it to the `frame` variable. The `cap.read()` function returns two values: `ret` (a boolean indicating if the read was successful) and `frame` (the actual frame).
    - `cv2.imshow("Line", frame)`: This line displays the initial frame in the "Line" window using the `cv2.imshow` function. The window will show the image, and the user can start drawing the line.
    - `if cv2.waitKey(1) == 13:`: This line checks if the user has pressed the Enter key (`13` is the ASCII code for Enter). `cv2.waitKey(1)` waits for a key event for 1 millisecond and

returns the ASCII code of the pressed key. If the pressed key is Enter, the condition is satisfied.
    - `cv2.destroyWindow("Line")`: This line closes the "Line" window using the `cv2.destroyWindow` function. It removes the window from the screen.
    - `break`: This line breaks out of the while loop, ending the execution of the loop.

```
ret, frame = cap.read()
cv2.namedWindow("Line")
cv2.setMouseCallback("Line", draw_line)
cv2.imshow("Line", frame)
while ret:                      # As long as frames are being read
successfully, the loop will continue.
    if cv2.waitKey(1) == 13:
        cv2.destroyWindow("Line")
        break
```

8. After the line is drawn, the code proceeds to track the specified object (defined by the bounding box) in subsequent frames using the initialized tracker.

```
success, frame = cap.read()
bbox = cv2.selectROI("Tracking", frame, False)
tracker.init(frame, bbox)  # This method initializes the tracker with the
initial frame and bbox coordinates. It sets the tracker's state based on the
initial position of the object to be tracked.
```

9. This function is responsible for drawing a rectangle around the tracked object and performing RPM calculation. It takes the current frame and bounding box coordinates (`bbox`) as inputs.

```
def drawBox(frame, bbox):
    # Drawing a rectangle and other elements on the frame
    x, y, w, h = int(bbox[0]), int(bbox[1]), int(bbox[2]), int(bbox[3])
    cv2.rectangle(frame, (x,y), ((x+w), (y+h)), (255,0,255), 3, 1)
    cv2.putText(frame, "Tracking", (75,75), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
(0,255, 0), 2)
    cv2.circle(frame, (int(x+w/2),int(y+h/2)), 5, (0,255,0), 1)
    now = line(x,y)
    # RPM calculation
    global prev   # keep track of the previous state (above or below the
drawn line) for comparison.
    global rev    # keep track of the number of line crossings.
    global tstamp # store the timestamp (in milliseconds) of the current
frame.
    global rTime  # store the timestamp (in milliseconds) of the last line
crossing.
    if(prev!= now and tstamp/1000 >= 1):    #checks if there has been a
change in the position
        print("time " ,tstamp/1000)
        print("ptime ", rTime/1000)
        prev = now
        rev += 1
        print("rev ",int(rev/2))

        print("rpm ", 60*1000*(1/((tstamp-rTime)*2)))
        rTime = tstamp
```

10. The main loop runs continuously, reading frames from the video, performing object tracking, calculating RPM, and displaying the tracked object and RPM values

```
while True:
    # print(start_point)
    # print(end_point)
    timer = cv2.getTickCount()
    success, frame = cap.read()

    # frame = cv2.resize(frame, (720, 720))

    if not success:
        break

    # Object tracking, RPM calculation, and display
    success, bbox = tracker.update(frame)
    if success:
        drawBox(frame, bbox)
    else:
        cv2.putText(frame, "Lost", (75,75), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
(0,0,255), 2)

    tstamp = cap.get(cv2.CAP_PROP_POS_MSEC)
    cv2.putText(frame, str(int(fps)), (75,50), cv2.FONT_HERSHEY_SIMPLEX,
0.7, (0,0,255), 2)
    cv2.putText(frame, str(int(tstamp/1000)), (75,90),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0,0,255), 2)
    cv2.line(frame, start_point, end_point, (0, 0, 255), 2)
    cv2.imshow("Tracking", frame)

    if cv2.waitKey(1) & 0xff == ord('q'):
        break



# print("rev ",rev)

cap.release()
cv2.destroyAllWindows()
```

Method-1: Shi-Tomasi Corner Detection Method using OpenCV

What is a Corner? ------- junction of two edges (where an edge is a sudden change in image brightness).

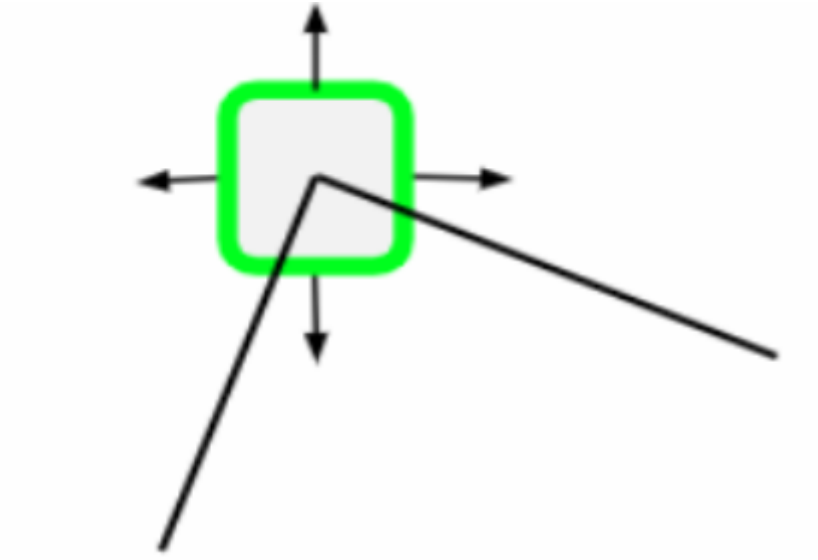
The corners of an image are basically identified as the regions in which there are variations in large intensity of the gradient in all possible dimensions and directions. Corners extracted can be a part of the image features, which can be matched with features of other images, and can be used to extract accurate information.

Shi-Tomasi Corner Detection –

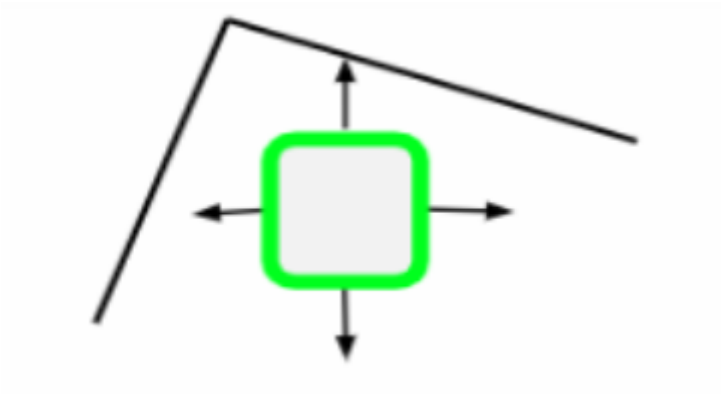
basic intuition is that corners can be detected by looking for significant change in all direction.

We consider a small window on the image then scan the whole image, looking for corners.

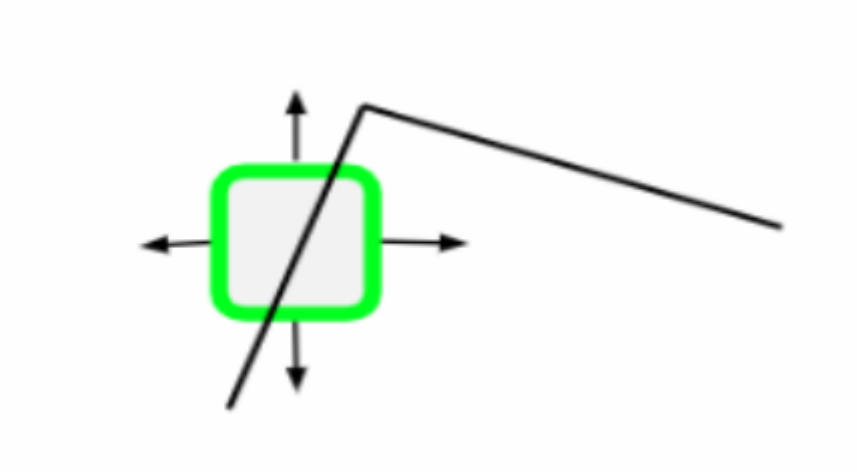
Shifting this small window in any direction would result in a large change in appearance, if that particular window happens to be located on a corner.



Flat regions will have no change in any direction.



If there's an edge, then there will be no major change along the edge direction.



Mathematical Overview –

For a window(W) located at (X, Y) with pixel intensity I(X, Y), formula for Shi-Tomasi Corner Detection is –

```
f(X, Y) = Σ (I(Xk, Yk) - I(Xk + ΔX, Yk + ΔY))2  where (Xk, Yk) ϵ W
```

According to the formula:
If we're scanning the image with a window just as we would with a kernel and we notice that there is an area where there's a major change no matter in what direction we actually scan, then we have a good intuition that there's probably a corner there.

Calculation of f(X, Y) will be really slow. Hence, we use Taylor expansion to simplify the scoring function, R.

```
R = min(λ1, λ2)
where λ1, λ2 are eigenvalues of resultant matrix
```

Using `goodFeaturesToTrack()` function –

> **Syntax :** cv2.goodFeaturesToTrack(gray_img, maxc, Q, minD)
>
> **Parameters :**
> **gray_img –** Grayscale image with integral values
> **maxc –** Maximum number of corners we want(give negative value to get all the corners)
> **Q –** Quality level parameter(preferred value=0.01)
> **maxD –** Maximum distance(preferred value=10)

Our Implementation:

```
import cv2
import numpy as np
import csv

# Replace 'input_video.mp4' with the path to your video file
video_path = 'D:/BTP/CODE/Videos/gray2.mp4'
cap = cv2.VideoCapture(video_path)

def nothing(x):
    pass
```

```
cv2.namedWindow("Frame")
cv2.createTrackbar("quality", "Frame", 1, 100, nothing)

# Open a CSV file for writing the corner coordinates
csv_file = open('corner_coordinates.csv', 'w', newline='')
csv_writer = csv.writer(csv_file)
csv_writer.writerow(['Frame', 'X', 'Y'])

frame_count = 0

while True:
    ret, frame = cap.read()
    if not ret:
        break

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    quality = cv2.getTrackbarPos("quality", "Frame")
    quality = quality / 100 if quality > 0 else 0.01
    # Change 50 to the no of corners we want to detect
    corners = cv2.goodFeaturesToTrack(gray, 50, quality, 20)

    if corners is not None:
        corners = np.int0(corners)

        for corner in corners:
            x, y = corner.ravel()
            cv2.circle(frame, (x, y), 3, (0, 0, 255), -1)

            # Write corner coordinates to CSV
            csv_writer.writerow([frame_count, x, y])

    cv2.imshow("Frame", frame)

    key = cv2.waitKey(1)
    if key == 27:
        break

    frame_count += 1

cap.release()
cv2.destroyAllWindows()

# Close the CSV file
csv_file.close()
```

We are extracting only the extreme coordinates in +X,-X directions since they will surely be the tip coordinates. The difference in corresponding values of Y coordinates gives the track.

Also, note that the code may give nearly close values to the leftmost and rightmost X since it might happen that there is less change btw 2 frames but it can be clearly seen that the corresponding Y values of all nearest X are almost same.

To accurately convert pixel coordinates to real-world units like millimeters, you need to perform camera calibration.

Code to find max and min X:

```
import csv

def extract_extremes(csv_file, column_index):
    with open(csv_file, 'r') as file:
        reader = csv.reader(file)
        header = next(reader)  # Skip header row
        data = list(reader)

    if column_index >= len(header):
        print("Invalid column index.")
        return

    min_value = float('inf')
    max_value = float('-inf')
    min_row = None
    max_row = None

    for row in data:
        try:
            value = float(row[column_index])
            if value < min_value:
                min_value = value
                min_row = row
            if value > max_value:
                max_value = value
                max_row = row
        except ValueError:
            pass  # Ignore rows with non-numeric values in the specified column

    return min_row, max_row

csv_file = 'D:/BTP/corner_coordinates.csv'  # Replace with the path to your CSV
file
column_index = 1  # Replace with the index of the column you want to extract
extremes from

min_row, max_row = extract_extremes(csv_file, column_index)

if min_row:
    print(f"Minimum value in column {column_index}: {min_row[column_index]}")
    print("Corresponding row:", min_row)
else:
    print("No valid minimum value found.")

if max_row:
    print(f"Maximum value in column {column_index}: {max_row[column_index]}")
    print("Corresponding row:", max_row)
else:
    print("No valid maximum value found.")
```

Method-2: Corner detection with Harris Corner Detection method using OpenCV

About the function used:

> **Syntax:** cv2.cornerHarris(src, dest, blockSize, kSize, freeParameter, borderType)
> **Parameters:**
> **src** – Input Image (Single-channel, 8-bit or floating-point)
> **dest** – Image to store the Harris detector responses. Size is same as source image
> **blockSize** – Neighborhood size (for each pixel value blockSize * blockSize neighbourhood is considered)
> **ksize** – Aperture parameter for the Sobel() operator
> **freeParameter** – Harris detector free parameter
> **borderType** – Pixel extrapolation method (the extrapolation mode used returns the coordinate of the pixel corresponding to the specified extrapolated pixel)

For further sub pixel accuracy, we can use cv2.cornerHarris() method.

Also we only want integer coordinates, so we need to do this step too:

```
corners = np.int0(corners)
```

Below is our Python implementation :

```
import cv2
import numpy as np
import pandas as pd

# Open the video capture
video_path = 'D:/BTP/CODE/Videos/gray2.mp4'
cap = cv2.VideoCapture(video_path)

# Create an empty DataFrame to store corner coordinates
corner_data = pd.DataFrame(columns=['Frame', 'Corner Number', 'X', 'Y'])

corner_number = 0

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Convert the frame to grayscale
    operatedImage = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Modify the data type and apply cv2.cornerHarris
    operatedImage = np.float32(operatedImage)
    dest = cv2.cornerHarris(operatedImage, 2, 5, 0.07)
```

```
    # Results are marked through the dilated corners
    dest = cv2.dilate(dest, None)

    # Find the coordinates of corner points
    corner_coords = np.argwhere(dest > 0.01 * dest.max())

    for coord in corner_coords:
        x, y = coord[1], coord[0]
        corner_number += 1

        # Draw a circle and put corner number
        cv2.circle(frame, (x, y), 1, (0, 0, 255), 2)
        # cv2.putText(frame, str(corner_number), (x + 10, y),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)

        # Append the corner coordinates to the DataFrame
        corner_data = pd.concat([corner_data, pd.DataFrame({'Frame':
[int(cap.get(cv2.CAP_PROP_POS_FRAMES))], 'Corner Number': [corner_number], 'X':
[x], 'Y': [y]})], ignore_index=True)

    # Display the frame with corner points
    cv2.imshow('Video with Corners', frame)

    # Exit the loop when 'q' is pressed
    if cv2.waitKey(0) & 0xFF == ord('q'):
        break

# Release the video capture and close windows
cap.release()
cv2.destroyAllWindows()

# Save corner_data DataFrame to a CSV file
corner_data.to_csv('corner_coordinates.csv', index=False)
cv2.destroyAllWindows()
```

Sources

Python | Corner Detection with Shi-Tomasi Corner Detection Method using OpenCV - GeeksforGeeks

[Object Detection – Opencv & Deep learning | Video Course 2022] - Pysource

Corners detection – OpenCV 3.4 with python 3 Tutorial 22 - YouTube