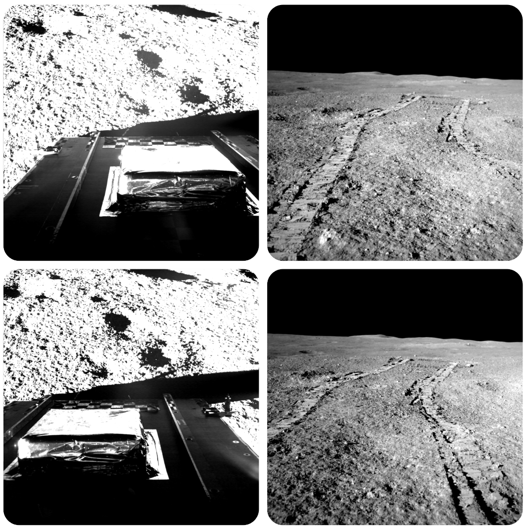
ROI selection for matching

Table of Contents



```
# -*- coding: utf-8 -*-
"""distance by feature matching.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1Fz3D_KkflZcKy6yehER
KPIkiANIm2I9c
"""
```

```
import os
import random
from PIL import Image
import matplotlib.pyplot as plt

def get_suffix(filename):
  """
Extract the portion starting from '_d_img_' so we can match left
and right images.
For example:
ch3_nav_nrl_20230823T1801171121_d_img_d32_001.png
becomes
_d_img_d32_001.png
"""
  marker = "_d_img_"
  if marker in filename:
  # Split once on '_d_img_' and keep that plus everything after
    parts = filename.split(marker, 1)
  # parts[0] = everything before '_d_img_', parts[1] = everything
after
    return marker + parts[1] # e.g. '_d_img_d32_001.png'
  return filename # fallback if '_d_img_' not found

def display_random_image_pairs(folder_path, num_pairs=2):
  """
  Displays a specified number of random left-right image pairs by
  matching
  on the suffix starting from '_d_img_'.
  """
  # Collect all _nrl_ (left) and _nrr_ (right) images
  left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
  right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]
  # Check if there are left and right images
  if not left_images or not right_images:
    print("No left or right images found in the folder.")
    return

  # Build dictionaries keyed by the '_d_img_' suffix
  left_map = {}
  for file in left_images:
    key = get_suffix(file)
    left_map[key] = file # each suffix → last left file with that suffix

  right_map = {}
  for file in right_images:
    key = get_suffix(file)
```

```
    right_map[key] = file # each suffix → last right file with that
suffix

  # Find common keys (suffixes) between left and right
  common_keys = list(left_map.keys() & right_map.keys())
  # print(common_keys)
  if not common_keys:
    print("No matching left-right image pairs found.")
    return

  # Randomly select up to 'num_pairs' unique suffixes
  num_pairs = min(num_pairs, len(common_keys))
  selected_keys = random.sample(common_keys, num_pairs)
  # Display each selected pair
  for idx, key in enumerate(selected_keys, start=1):
    left_file = os.path.join(folder_path, left_map[key])
    right_file = os.path.join(folder_path, right_map[key])
    # Open images safely
    try:
      left_img = Image.open(left_file).convert("RGB")
      right_img = Image.open(right_file).convert("RGB")
    except Exception as e:
      print(f"Error loading images for key {key}: {e}")
      continue
    # Plot the images side by side
    fig, axs = plt.subplots(1, 2, figsize=(8,4))
    axs[0].imshow(left_img)
    axs[0].set_title(f"Left Image ({idx}/{num_pairs})")
    axs[0].axis("off")
    axs[1].imshow(right_img)
    axs[1].set_title(f"Right Image ({idx}/{num_pairs})")
    axs[1].axis("off")
    plt.tight_layout()
    plt.show()

display_random_image_pairs(r"/content/drive/MyDrive/filtered_i
mages", num_pairs=2)

# import os

# def get_image_pairs(folder_path):
#   """
#   Scans an image folder and creates a list of paths of image
pairs,
#   matching them based on the suffix starting from '_d_img_'.
#   """
```

```
#   # Collect all _nrl_ (left) and _nrr_ (right) images
#   left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
#   right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]

#   # Check if there are left and right images
#   if not left_images or not right_images:
#     print("No left or right images found in the folder.")
#     return []

#   # Build dictionaries keyed by the '_d_img_' suffix
#   left_map = {}
#   for file in left_images:
#     key = get_suffix(file)  # Assuming you have the get_suffix
function
#     left_map[key] = file

#   right_map = {}
#   for file in right_images:
#     key = get_suffix(file)  # Assuming you have the get_suffix
function
#     right_map[key] = file

#   # Find common keys (suffixes) between left and right
#   common_keys = list(left_map.keys() & right_map.keys())
#   if not common_keys:
#     print("No matching left-right image pairs found.")
#     return []

#   # Create a list of image pair paths
#   image_pairs = []
#   for key in common_keys:
#     left_file = os.path.join(folder_path, left_map[key])
#     right_file = os.path.join(folder_path, right_map[key])
#     image_pairs.append((left_file, right_file))  # Store as tuples

#   return image_pairs

# from PIL import Image
# img_pairs =
get_image_pairs(r"/content/drive/MyDrive/filtered_images")
# for i in range(len(img_pairs)):
#   path = img_pairs[i][0]
#   if '008' in path:
#     print(path)

import os
```

```
def
get_image_pairs_with_suffix(folder_path='/content/drive/MyDriv
e/filtered_images', suffix_number='001'):
  """
  Scans an image folder and creates a list of paths of image pairs,
  matching them based on the suffix starting from '_d_img_' and
  containing the specified suffix_number.
  """
  # Collect all _nrl_ (left) and _nrr_ (right) images
  left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
  right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]

  # Check if there are left and right images
  if not left_images or not right_images:
    print("No left or right images found in the folder.")
    return []

  # Build dictionaries keyed by the '_d_img_' suffix
  left_map = {}
  for file in left_images:
    key = get_suffix(file)  # Assuming you have the get_suffix
function
    left_map[key] = file

  right_map = {}
  for file in right_images:
    key = get_suffix(file)  # Assuming you have the get_suffix
function
    right_map[key] = file

  # Find common keys (suffixes) between left and right containing
suffix_number
  common_keys = [key for key in left_map.keys() &
right_map.keys() if str(suffix_number) in key]

  if not common_keys:
    print(f"No matching left-right image pairs found for suffix
number {suffix_number}.")
    return []

  # Create a list of image pair paths
  image_pairs = []
  for key in common_keys:
    left_file = os.path.join(folder_path, left_map[key])
    right_file = os.path.join(folder_path, right_map[key])
```

```
    image_pairs.append((left_file, right_file))  # Store as tuples

  return image_pairs

get_image_pairs_with_suffix(suffix_number='d32_008')

import cv2
import numpy as np
import matplotlib.pyplot as plt
image_pair_008 =
get_image_pairs_with_suffix(suffix_number='d32_008')

left_image_path = image_pair_008[0][0]
right_image_path = image_pair_008[0][1]

def load_image(left_image_path,right_image_path):
  left_image = cv2.imread(left_image_path,
cv2.IMREAD_GRAYSCALE)
  right_image = cv2.imread(right_image_path,
cv2.IMREAD_GRAYSCALE)

  if left_image is None or right_image is None:
      print("Error loading images.")

  if left_image.shape != right_image.shape:
      height = min(left_image.shape[0], right_image.shape[0])
      width = min(left_image.shape[1], right_image.shape[1])
      left_image = cv2.resize(left_image, (width, height))
      right_image = cv2.resize(right_image, (width, height))

  left_image = left_image.astype('uint8')
  right_image = right_image.astype('uint8')

  print(left_image_path)
  print(right_image_path)

  return left_image, right_image

left_image, right_image =
load_image(left_image_path,right_image_path)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(left_image, cmap='gray')
plt.title('Left Image')
plt.subplot(1, 2, 2)
plt.imshow(right_image, cmap='gray')
```

```
plt.title('Right Image')
plt.axis('off');

def create_disparity_map(left_image, right_image):
  stereo = cv2.StereoSGBM_create(numDisparities=4,
                                 blockSize=23)
  disparity_map = stereo.compute(left_image, right_image)
  disparity_map_normalized = cv2.normalize(disparity_map,
None,
                                           alpha=0, beta=255,
                                           norm_type=cv2.NORM_MINMAX,
dtype=cv2.CV_8U)

  plt.figure()
  plt.imshow(disparity_map_normalized, cmap='jet')
  plt.colorbar(label="Disparity (pixels)")
  plt.title("Disparity Map")
  plt.show()

  return disparity_map_normalized

map = create_disparity_map(left_image, right_image)

import cv2
import numpy as np
import matplotlib.pyplot as plt

def preprocess_images(left_img, right_img):
  left_float = cv2.normalize(left_img.astype('float32'), None, 0.0,
1.0, cv2.NORM_MINMAX)
  right_float = cv2.normalize(right_img.astype('float32'), None,
0.0, 1.0, cv2.NORM_MINMAX)

  # Apply CLAHE (Contrast Limited Adaptive Histogram
Equalization)
  clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
  left_clahe = clahe.apply(cv2.convertScaleAbs(left_float*255))
  right_clahe = clahe.apply(cv2.convertScaleAbs(right_float*255))

  # Denoise
  left_denoised = cv2.fastNlMeansDenoising(left_clahe)
  right_denoised = cv2.fastNlMeansDenoising(right_clahe)

  # Apply unsharp masking for edge enhancement
  gaussian = cv2.GaussianBlur(left_denoised, (0, 0), 2.0)
  left_enhanced = cv2.addWeighted(left_denoised, 1.5, gaussian,
```

```
-0.5, 0)
  gaussian = cv2.GaussianBlur(right_denoised, (0, 0), 2.0)
  right_enhanced = cv2.addWeighted(right_denoised, 1.5,
gaussian, -0.5, 0)

  return left_enhanced, right_enhanced

left_enhanced, right_enhanced = preprocess_images(left_image,
right_image)

def match_features(left_img, right_img):
    """
    Matches features between left and right images using SIFT and
    epipolar geometry constraints.

    Args:
        left_img: The left image.
        right_img: The right image.

    Returns:
        good_matches: A list of good matches after filtering.
        kp1, kp2: Keypoints detected in the left and right images.
    """
    # Create SIFT detector with adjusted parameters
    sift = cv2.SIFT_create(
        nfeatures=0,  # no limit on number of features
        nOctaveLayers=5,  # increase number of scale layers
        contrastThreshold=0.04,  # lower threshold to detect more
features
        edgeThreshold=10,  # increase edge threshold
        sigma=1.6
    )

    # Detect keypoints and compute descriptors
    kp1, des1 = sift.detectAndCompute(left_img, None)
    kp2, des2 = sift.detectAndCompute(right_img, None)

    # Configure FLANN matcher
    index_params = {
        'algorithm': 1,  # KDTREE
        'trees': 8  # number of parallel kd-trees
    }
    search_params = {
        'checks': 100  # increase number of searches
    }
    flann = cv2.FlannBasedMatcher(index_params,
```

```
search_params)

    # Perform matching with k=2
    matches = flann.knnMatch(des1, des2, k=2)

    # Apply ratio test with adjusted threshold
    good_matches = []
    for m, n in matches:
        # Adjust ratio threshold (0.75-0.8 is typical range)
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    # Filter matches based on epipolar constraints
    if len(good_matches) > 8:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)

        # Use RANSAC to find fundamental matrix and filter outliers
        F, mask = cv2.findFundamentalMat(src_pts, dst_pts,
cv2.FM_RANSAC, 3, 0.99)

        # Keep only inlier matches
        good_matches = [good_matches[i] for i in range(len(mask)) if
mask[i]]

    return good_matches, kp1, kp2

def draw_matches(left_img, right_img, good_matches, kp1, kp2):
  matched_image = cv2.drawMatches(left_img, kp1, right_img,
kp2, good_matches, None, flags=2)
  plt.axis('off')
  plt.imshow(matched_image)
  print(f"Good Matches found: {len(good_matches)}")
  return matched_image

def main(image_folder='/content/drive/MyDrive/filtered_images',
       suffix_number='d32_008'):
    """
    Main function to process image pairs.

    Args:
        image_folder (str): Path to the folder containing images.
        suffix_number (str, optional): Suffix number to filter images.
                               Defaults to None (processes all pairs).
```

```
    """
    image_pairs = get_image_pairs_with_suffix(image_folder,
suffix_number)

    left_image, right_image = load_image(image_pairs[0]
[0],image_pairs[0][1])
    left_enhanced, right_enhanced =
preprocess_images(left_image, right_image)

    good_matches, kp1, kp2 = match_features(left_enhanced,
right_enhanced)
    matched_image = draw_matches(left_enhanced,
right_enhanced, good_matches, kp1, kp2)

    return matched_image

matched_image = main()

cv2.imwrite('matched_image.png', matched_image)

image_pairs =
get_image_pairs_with_suffix(suffix_number='d32_008')
left_image, right_image = load_image(image_pairs[0]
[0],image_pairs[0][1])
left_enhanced, right_enhanced = preprocess_images(left_image,
right_image)
good_matches, kp1, kp2 = match_features(left_enhanced,
right_enhanced)

disparities = []
pts1 = []
pts2 = []

h1, w1 = left_image.shape
h2, w2 = right_image.shape

for match in good_matches:
  pt1 = kp1[match.queryIdx].pt
  pt2 = kp2[match.trainIdx].pt
  pts1.append(pt1)
  pts2.append((pt2[0]+w1, pt2[1]))
  disparity = (abs(pt1[0] - pt2[0]))
  disparities.append(disparity)

baseline = 0.24 # in m
focal_length_pixels = 2062.16 # in pixels
```

depths = [(baseline * focal_length_pixels)/d if d > 0 else float('inf') for d in disparities]

import numpy as np
np.min(depths)

This is my code that browses a folder and takes stereo image pairs (I have attached a few here) and processes the images to find and match sift features. After finding all the features I want to prompt the user to select ROI by drawing a bounding box on each image with mouse, we then will find and match the features from the roi only.

Thought about feature matching and ROI selection for a few seconds

Below is an example of how you can integrate an interactive Region of Interest (ROI) selection (via mouse-drawn bounding boxes) into your stereo-processing workflow. The key idea is:

1. **Display** each (left/right) image using `cv2.imshow(...)`.
2. **Let the user** draw a bounding box using OpenCV's built-in `cv2.selectROI(...)`.
3. **Crop** the selected ROI regions in both images.
4. **Run** the existing feature-detection/matching pipeline on those cropped ROIs only.

---

1. A Helper Function To Select and Crop ROI

python

```python
import cv2

def select_and_crop_roi(image, window_name="Select ROI"):
    """
    Given a single image (NumPy array),
    - opens a window where user can draw a bounding box,
    - returns the cropped region of interest plus the bounding box coordinates.
    """
    # Make a copy so selectROI won't modify the original
    temp_image = image.copy()

    # Let user draw a bounding box with mouse
    # This call will block until the user finishes drawing or presses 'Enter'
    # bounding_box = (x, y, w, h)
```

```python
    bounding_box = cv2.selectROI(window_name, temp_image, fromCenter=False, showCrosshair=True)
    x, y, w, h = bounding_box

    # Crop the image to that ROI
    roi = temp_image[y : y+h, x : x+w]

    # Close the ROI selection window
    cv2.destroyWindow(window_name)

    return roi, (x, y, w, h)
```

Notes:

- The function `cv2.selectROI` blocks execution until the user finalizes the selection (Enter or Space) or cancels (ESC).
- `temp_image` is used to avoid altering your original arrays inadvertently.
- After finishing the selection, we call `cv2.destroyWindow(...)` so it doesn't linger.

---

2. Using the ROI in Your Main Pipeline

Let's assume you already have a left/right image pair (`left_image` , `right_image`), just as in your code. You want to let the user pick an ROI in each, then run feature matching on only those cropped areas.

```python
def main_with_roi(
    left_image_path,
    right_image_path
):
    # 1. Load images (grayscale)
    left_full = cv2.imread(left_image_path, cv2.IMREAD_GRAYSCALE)
    right_full = cv2.imread(right_image_path, cv2.IMREAD_GRAYSCALE)

    # 2. (Optional) Resize so both match in dimension
    #    or at least handle any shape mismatch as you do in your code
    if left_full.shape != right_full.shape:
        height = min(left_full.shape[0], right_full.shape[0])
        width = min(left_full.shape[1], right_full.shape[1])
        left_full = cv2.resize(left_full, (width, height))
        right_full = cv2.resize(right_full, (width, height))

    # 3. Interactively select ROI in the left image
    #    The user must press ENTER (or SPACE) to confirm the rectangle
    left_roi, left_bbox = select_and_crop_roi(left_full, "Select ROI in Left Image")

    # 4. Interactively select ROI in the right image
```

```
    right_roi, right_bbox = select_and_crop_roi(right_full, "Select ROI in Right Image")

    # 5. (Optional) You can still do your preprocessing on these ROI patches
    left_enhanced, right_enhanced = preprocess_images(left_roi, right_roi)

    # 6. Run SIFT (or other) feature matching on ROI patches only
    good_matches, kp_left, kp_right = match_features(left_enhanced, right_enhanced)

    print(f"Found {len(good_matches)} matches in the chosen ROI.")

    # 7. Draw matched features for a quick check
    matched_image = cv2.drawMatches(
        left_enhanced, kp_left,
        right_enhanced, kp_right,
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # 8. Show the matched result
    cv2.imshow("Matched ROI Features", matched_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return matched_image

# Example usage:
# main_with_roi("left_image.png", "right_image.png")
```

Points to note:

- **Preprocessing**: You can run your `preprocess_images()` on the *cropped ROI* region. Doing so helps keep the pipeline consistent with your original approach.
- **Keypoint Coordinates**: Because you're detecting features only in the cropped patch, the keypoint coordinates will be relative to that patch. If you need them in "full image" coordinates (for instance, to do a transform overlay on the full image), then you must offset them by `(x, y)` from `left_bbox` and `right_bbox`.
- **Disparities/Depth**: The same logic applies if you compute disparities and estimate depth. You will only get the disparity for the ROI portion. If you need to blend that back into a full-image depth map, you should keep track of bounding-box offsets.

---

3. Integrating With Your Existing Code

In your existing script, you can simply replace the lines where you do:

```python
```

```
left_image, right_image = load_image(image_pairs[0][0], image_pairs[0][1])
# ...
# good_matches, kp1, kp2 = match_features(left_enhanced, right_enhanced)
```

with the snippet that selects ROIs and then matches only within those ROIs. The rest of your code (e.g. computing disparity, baseline, focal length, etc.) is the same, just done on the ROI-limited matches.

Important Caveats

1. **User Interaction**: Since `cv2.selectROI()` is interactive, it must run in an environment where OpenCV windows are supported (local machine, some IDEs, etc.). If you're in a headless environment (e.g., certain remote notebooks), you'll need a different approach (e.g., manual coordinates or a custom matplotlib-based ROI selection).
2. **Manual Repeats**: Each call to `selectROI` requires pressing **Enter** or **Space** to confirm the bounding box. Pressing ESC cancels. You might want to handle corner cases (e.g., user cancels) by checking if `bounding_box` is `(0,0,0,0)`.
3. **Multiple ROIs**: If you want to let the user draw multiple bounding boxes at once in a single image, you can use `cv2.selectROIs(...)` which returns a list of bounding boxes. You would then do repeated cropping/feature matching for each box in that list.

That's it—just wrap each image in a `select_and_crop_roi(...)` call, then feed those cropped patches to your usual SIFT matching code. This approach keeps your existing pipeline basically unchanged, only you insert a user-defined ROI selection step before the matching.

give me entire code function by function

Thought about ROI selection and feature matching for a second

Below is a **complete, function-by-function** Python script that integrates an **interactive ROI selection** (via mouse-drawn bounding boxes) into your stereo pipeline.

This example covers the following steps:

1. **Utility functions** to match left-right images by filename.
2. **ROI selection** via `cv2.selectROI(...)`.
3. **Preprocessing** (CLAHE, denoising, unsharp masking).

4. **SIFT feature detection** and epipolar filtering.
5. **Drawing matches** and calculating disparity/depth if desired.

You can copy and paste the entire code into one file (e.g., `stereo_roi.py`) and run it in an environment that supports OpenCV GUI windows.

---

python

```python
import os
import random
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

##########################################################################################
# 1. Utility function: get_suffix
##########################################################################################
def get_suffix(filename):
    """
    Extract the portion starting from '_d_img_' so we can match left
    and right images by that suffix.

    For example:
        ch3_nav_nrl_20230823T1801171121_d_img_d32_001.png
    becomes:
        _d_img_d32_001.png
    """
    marker = "_d_img_"
    if marker in filename:
        # Split once on '_d_img_' and keep that plus everything after
        parts = filename.split(marker, 1)
        # parts[0] = everything before '_d_img_', parts[1] = everything after
        return marker + parts[1]  # e.g. '_d_img_d32_001.png'
    return filename  # fallback if '_d_img_' not found

##########################################################################################
# 2. Utility function: get_image_pairs_with_suffix
##########################################################################################
def get_image_pairs_with_suffix(folder_path, suffix_number='001'):
    """
    Scans an image folder and creates a list of left-right image paths,
    matching them based on the suffix starting from '_d_img_' and
    containing the specified suffix_number.

    Args:
```

```
        folder_path (str): path to the directory containing images
        suffix_number (str): partial string to match in the suffix
                             (e.g., 'd32_008' or just '001')

    Returns:
        image_pairs (list): list of tuples (left_file, right_file)
    """
    # Collect all _nrl_ (left) and _nrr_ (right) images
    left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
    right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]

    # Build dictionaries keyed by the '_d_img_' suffix
    left_map = {}
    for file in left_images:
        key = get_suffix(file)
        left_map[key] = file

    right_map = {}
    for file in right_images:
        key = get_suffix(file)
        right_map[key] = file

    # Find common keys (suffixes) that contain the suffix_number
    common_keys = [key for key in (left_map.keys() & right_map.keys())
                   if str(suffix_number) in key]

    if not common_keys:
        print(f"No matching left-right image pairs found for suffix number {suffix_number}.")
        return []

    # Create a list of image pair paths
    image_pairs = []
    for key in common_keys:
        left_file = os.path.join(folder_path, left_map[key])
        right_file = os.path.join(folder_path, right_map[key])
        image_pairs.append((left_file, right_file))

    return image_pairs

##############################################################################
# 3. Utility function: load_image
##############################################################################
def load_image(left_image_path, right_image_path):
    """
    Loads and (optionally) resizes two images so they share the same shape.

    Args:
        left_image_path (str): path to the left image
        right_image_path (str): path to the right image

    Returns:
        left_image, right_image: both as uint8 grayscale
```

```
    """
    left_image = cv2.imread(left_image_path, cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread(right_image_path, cv2.IMREAD_GRAYSCALE)

    if left_image is None or right_image is None:
        print("Error loading images.")
        return None, None

    if left_image.shape != right_image.shape:
        height = min(left_image.shape[0], right_image.shape[0])
        width = min(left_image.shape[1], right_image.shape[1])
        left_image = cv2.resize(left_image, (width, height))
        right_image = cv2.resize(right_image, (width, height))

    left_image = left_image.astype('uint8')
    right_image = right_image.astype('uint8')

    return left_image, right_image

################################################################################
# 4. Utility function: preprocess_images
################################################################################
def preprocess_images(left_img, right_img):
    """
    Preprocesses images by:
      1. Normalizing to float32 [0..1]
      2. Applying CLAHE
      3. Denoising
      4. Unsharp masking for edge enhancement

    Args:
        left_img, right_img (np.array): grayscale uint8 images

    Returns:
        left_enhanced, right_enhanced
    """
    # Convert to float in [0..1]
    left_float = cv2.normalize(left_img.astype('float32'), None, 0.0, 1.0, cv2.NORM_MINMAX)
    right_float = cv2.normalize(right_img.astype('float32'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    # CLAHE
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    left_clahe = clahe.apply(cv2.convertScaleAbs(left_float*255))
    right_clahe = clahe.apply(cv2.convertScaleAbs(right_float*255))

    # Denoise
    left_denoised = cv2.fastNlMeansDenoising(left_clahe)
    right_denoised = cv2.fastNlMeansDenoising(right_clahe)

    # Unsharp masking
    gaussian = cv2.GaussianBlur(left_denoised, (0, 0), 2.0)
    left_enhanced = cv2.addWeighted(left_denoised, 1.5, gaussian, -0.5, 0)
```

```
    gaussian = cv2.GaussianBlur(right_denoised, (0, 0), 2.0)
    right_enhanced = cv2.addWeighted(right_denoised, 1.5, gaussian, -0.5, 0)

    return left_enhanced, right_enhanced

##########################################################################
# 5. Utility function: match_features
##########################################################################
def match_features(left_img, right_img):
    """
    Matches features between left and right images using SIFT + FLANN
    with ratio test and a fundamental matrix filter.

    Args:
        left_img: The left preprocessed image (grayscale).
        right_img: The right preprocessed image (grayscale).

    Returns:
        good_matches: A list of good matches after filtering.
        kp1, kp2: Keypoints detected in the left and right images.
    """
    # Create SIFT detector
    sift = cv2.SIFT_create(
        nfeatures=0,          # no explicit limit
        nOctaveLayers=5,      # more scale layers
        contrastThreshold=0.04,
        edgeThreshold=10,
        sigma=1.6
    )

    # Detect keypoints and descriptors
    kp1, des1 = sift.detectAndCompute(left_img, None)
    kp2, des2 = sift.detectAndCompute(right_img, None)

    # Configure FLANN
    index_params = dict(algorithm=1, trees=8)  # 1->KDTREE
    search_params = dict(checks=100)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # FLANN matching with k=2
    matches = flann.knnMatch(des1, des2, k=2)

    # Ratio test
    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    # Filter matches with epipolar constraint (Fundamental matrix)
    if len(good_matches) > 8:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
```

```
        F, mask = cv2.findFundamentalMat(src_pts, dst_pts, cv2.FM_RANSAC, 3, 0.99)
        if mask is not None:
            good_matches = [good_matches[i] for i in range(len(mask)) if mask[i]]

    return good_matches, kp1, kp2

################################################################################
# 6. Utility function: draw_matches
################################################################################
def draw_matches(left_img, right_img, good_matches, kp1, kp2):
    """
    Draws lines connecting matched keypoints.

    Args:
        left_img, right_img (np.array): The images where keypoints come from
        good_matches: list of good matches
        kp1, kp2: keypoints from left, right images
    """
    matched_image = cv2.drawMatches(
        left_img, kp1,
        right_img, kp2,
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )
    plt.figure(figsize=(10, 5))
    plt.imshow(matched_image, cmap='gray')
    plt.title(f"Good Matches: {len(good_matches)}")
    plt.axis('off')
    plt.show()
    return matched_image

################################################################################
# 7. Utility function: select_and_crop_roi
################################################################################
def select_and_crop_roi(image, window_name="Select ROI"):
    """
    Given a single image (NumPy array),
    - opens a window where user can draw a bounding box,
    - returns the cropped region of interest plus the bounding box coordinates.
    """
    # Make a copy so selectROI won't modify the original
    temp_image = image.copy()

    # Let user draw a bounding box with mouse
    # bounding_box = (x, y, w, h)
    bounding_box = cv2.selectROI(window_name, temp_image, fromCenter=False, showCrosshair=True)
    x, y, w, h = bounding_box

    # Crop the image to that ROI
    roi = temp_image[y : y+h, x : x+w]
```

```
    # Close the ROI selection window
    cv2.destroyWindow(window_name)

    return roi, (x, y, w, h)

###########################################################################
# 8. MAIN: Demonstration using user-selected ROI
###########################################################################
def main_with_roi(folder_path, suffix_number='d32_008'):
    """
    Main function to:
      - find a left-right pair of images matching 'suffix_number',
      - let the user pick ROIs in each image,
      - preprocess & match features within those ROIs,
      - display the matches.

    Args:
        folder_path (str): path to folder containing stereo images
        suffix_number (str): partial suffix (e.g., 'd32_008')
    """
    # 1. Get one or more matching pairs
    image_pairs = get_image_pairs_with_suffix(folder_path, suffix_number)
    if not image_pairs:
        print("No image pairs found. Exiting.")
        return

    # Just take the first pair for demo
    left_image_path, right_image_path = image_pairs[0]

    # 2. Load in grayscale
    left_full, right_full = load_image(left_image_path, right_image_path)
    if left_full is None or right_full is None:
        return

    # 3. Let user select ROI in left image
    left_roi, left_bbox = select_and_crop_roi(left_full, "Select ROI in Left Image")

    # 4. Let user select ROI in right image
    right_roi, right_bbox = select_and_crop_roi(right_full, "Select ROI in Right Image")

    # 5. Preprocess these ROI regions
    left_enh, right_enh = preprocess_images(left_roi, right_roi)

    # 6. Match features
    good_matches, kp1, kp2 = match_features(left_enh, right_enh)

    print(f"Found {len(good_matches)} good matches in ROI.")

    # 7. Draw matches for quick visual check
    matched_img = cv2.drawMatches(
        left_enh, kp1,
        right_enh, kp2,
```

```
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # Display the matched image in OpenCV window (optional)
    # cv2.imshow("Matched ROI Features", matched_img)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    # Alternatively, display via matplotlib:
    plt.figure(figsize=(12, 6))
    plt.imshow(matched_img[..., ::-1])  # BGR -> RGB for matplotlib
    plt.title("Matches in Selected ROI")
    plt.axis('off')
    plt.show()

    # Return matched image (in case you want to save or further process)
    return matched_img

##############################################################################
# Example usage (if running as a script):
##############################################################################
if __name__ == "__main__":
    # Replace with your own folder path, e.g. "/content/drive/MyDrive/filtered_images"
    FOLDER_PATH = r"/path/to/your/filtered_images"
    SUFFIX = "d32_008"  # or anything else you're matching
    main_with_roi(FOLDER_PATH, SUFFIX)
```

How to Use

1. **Install Requirements:**
    - `pip install opencv-python numpy matplotlib pillow`
2. **Run** this script:

```bash
python stereo_roi.py
```

3. When prompted by the two OpenCV windows titled "Select ROI in Left Image" and "Select ROI in Right Image," **draw your bounding box** using the mouse, and then press **Enter** or **Space** to confirm. Press **ESC** to cancel (which will return `(0,0,0,0)`).
4. The script will then do:
    - Preprocessing on the chosen ROIs.

- SIFT feature detection+matching on those ROIs.
- Display the matched features with lines connecting correspondences.

Additional Notes

- `cv2.selectROI` blocks execution until you confirm/cancel selection. For multiple ROIs in the same image, you can use `cv2.selectROIs`, which returns multiple bounding boxes.
- If you want to compute actual disparity (and eventually depth), you must keep track of each match's `(x,y)` offsets in the original image space. Currently, everything is done in the ROI's local coordinates. You would offset by `(left_bbox[0], left_bbox[1])` or `(right_bbox[0], right_bbox[1])` to map back to full image coordinates.
- If you are on a **headless environment** (like many hosted notebooks), you may need a different strategy for ROI selection (e.g., using `matplotlib` and custom mouse callbacks).

```
import os
import random
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

############################################################
###########################
# 1. Utility function: get_suffix
############################################################
###########################
def get_suffix(filename):
    """
    Extract the portion starting from '_d_img_' so we can match
left
    and right images by that suffix.

    For example:
        ch3_nav_nrl_20230823T1801171121_d_img_d32_001.png
    becomes:
        _d_img_d32_001.png
    """
    marker = "_d_img_"
    if marker in filename:
        # Split once on '_d_img_' and keep that plus everything
after
        parts = filename.split(marker, 1)
```

```
        # parts[0] = everything before '_d_img_', parts[1] = everything
after
        return marker + parts[1]  # e.g. '_d_img_d32_001.png'
    return filename  # fallback if '_d_img_' not found

############################################################
##########################
# 2. Utility function: get_image_pairs_with_suffix
############################################################
##########################
def get_image_pairs_with_suffix(folder_path,
suffix_number='001'):
    """
    Scans an image folder and creates a list of left-right image
paths,
    matching them based on the suffix starting from '_d_img_' and
    containing the specified suffix_number.

    Args:
        folder_path (str): path to the directory containing images
        suffix_number (str): partial string to match in the suffix
                             (e.g., 'd32_008' or just '001')

    Returns:
        image_pairs (list): list of tuples (left_file, right_file)
    """
    # Collect all _nrl_ (left) and _nrr_ (right) images
    left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
    right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]

    # Build dictionaries keyed by the '_d_img_' suffix
    left_map = {}
    for file in left_images:
        key = get_suffix(file)
        left_map[key] = file

    right_map = {}
    for file in right_images:
        key = get_suffix(file)
        right_map[key] = file

    # Find common keys (suffixes) that contain the suffix_number
    common_keys = [key for key in (left_map.keys() &
right_map.keys())
                   if str(suffix_number) in key]
```

```
    if not common_keys:
        print(f"No matching left-right image pairs found for suffix
number {suffix_number}.")
        return []

    # Create a list of image pair paths
    image_pairs = []
    for key in common_keys:
        left_file = os.path.join(folder_path, left_map[key])
        right_file = os.path.join(folder_path, right_map[key])
        image_pairs.append((left_file, right_file))

    return image_pairs

########################################################################################
# 3. Utility function: load_image
########################################################################################
def load_image(left_image_path, right_image_path):
    """
    Loads and (optionally) resizes two images so they share the
same shape.

    Args:
        left_image_path (str): path to the left image
        right_image_path (str): path to the right image

    Returns:
        left_image, right_image: both as uint8 grayscale
    """
    left_image = cv2.imread(left_image_path,
cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread(right_image_path,
cv2.IMREAD_GRAYSCALE)

    if left_image is None or right_image is None:
        print("Error loading images.")
        return None, None

    if left_image.shape != right_image.shape:
        height = min(left_image.shape[0], right_image.shape[0])
        width = min(left_image.shape[1], right_image.shape[1])
        left_image = cv2.resize(left_image, (width, height))
        right_image = cv2.resize(right_image, (width, height))
```

```
    left_image = left_image.astype('uint8')
    right_image = right_image.astype('uint8')

    return left_image, right_image

############################################################
##########################
# 4. Utility function: preprocess_images
############################################################
##########################
def preprocess_images(left_img, right_img):
    """
    Preprocesses images by:
      1. Normalizing to float32 [0..1]
      2. Applying CLAHE
      3. Denoising
      4. Unsharp masking for edge enhancement

    Args:
        left_img, right_img (np.array): grayscale uint8 images

    Returns:
        left_enhanced, right_enhanced
    """
    # Convert to float in [0..1]
    left_float = cv2.normalize(left_img.astype('float32'), None, 0.0,
1.0, cv2.NORM_MINMAX)
    right_float = cv2.normalize(right_img.astype('float32'), None,
0.0, 1.0, cv2.NORM_MINMAX)

    # CLAHE
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    left_clahe = clahe.apply(cv2.convertScaleAbs(left_float*255))
    right_clahe =
clahe.apply(cv2.convertScaleAbs(right_float*255))

    # Denoise
    left_denoised = cv2.fastNlMeansDenoising(left_clahe)
    right_denoised = cv2.fastNlMeansDenoising(right_clahe)

    # Unsharp masking
    gaussian = cv2.GaussianBlur(left_denoised, (0, 0), 2.0)
    left_enhanced = cv2.addWeighted(left_denoised, 1.5, gaussian,
-0.5, 0)
    gaussian = cv2.GaussianBlur(right_denoised, (0, 0), 2.0)
    right_enhanced = cv2.addWeighted(right_denoised, 1.5,
```

```
gaussian, -0.5, 0)

    return left_enhanced, right_enhanced

############################################################
#########################
# 5. Utility function: match_features
############################################################
#########################
def match_features(left_img, right_img):
    """
    Matches features between left and right images using SIFT +
FLANN
    with ratio test and a fundamental matrix filter.

    Args:
        left_img: The left preprocessed image (grayscale).
        right_img: The right preprocessed image (grayscale).

    Returns:
        good_matches: A list of good matches after filtering.
        kp1, kp2: Keypoints detected in the left and right images.
    """
    # Create SIFT detector
    sift = cv2.SIFT_create(
        nfeatures=0,          # no explicit limit
        nOctaveLayers=5,      # more scale layers
        contrastThreshold=0.04,
        edgeThreshold=10,
        sigma=1.6
    )

    # Detect keypoints and descriptors
    kp1, des1 = sift.detectAndCompute(left_img, None)
    kp2, des2 = sift.detectAndCompute(right_img, None)

    # Configure FLANN
    index_params = dict(algorithm=1, trees=8)  # 1->KDTREE
    search_params = dict(checks=100)
    flann = cv2.FlannBasedMatcher(index_params,
search_params)

    # FLANN matching with k=2
    matches = flann.knnMatch(des1, des2, k=2)

    # Ratio test
```

```
    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    # Filter matches with epipolar constraint (Fundamental matrix)
    if len(good_matches) > 8:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)

        F, mask = cv2.findFundamentalMat(src_pts, dst_pts,
cv2.FM_RANSAC, 3, 0.99)
        if mask is not None:
            good_matches = [good_matches[i] for i in
range(len(mask)) if mask[i]]

    return good_matches, kp1, kp2

##############################################################
##########################
# 6. Utility function: draw_matches
##############################################################
##########################
def draw_matches(left_img, right_img, good_matches, kp1, kp2):
    """
    Draws lines connecting matched keypoints.

    Args:
        left_img, right_img (np.array): The images where keypoints
come from
        good_matches: list of good matches
        kp1, kp2: keypoints from left, right images
    """
    matched_image = cv2.drawMatches(
        left_img, kp1,
        right_img, kp2,
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )
    plt.figure(figsize=(10, 5))
    plt.imshow(matched_image, cmap='gray')
    plt.title(f"Good Matches: {len(good_matches)}")
    plt.axis('off')
    plt.show()
```

```
    return matched_image

############################################################
#########################
# 7. Utility function: select_and_crop_roi
############################################################
#########################
def select_and_crop_roi(image, window_name="Select ROI"):
    """
    Given a single image (NumPy array),
    - opens a window where user can draw a bounding box,
    - returns the cropped region of interest plus the bounding box
coordinates.
    """
    # Make a copy so selectROI won't modify the original
    temp_image = image.copy()

    # Let user draw a bounding box with mouse
    # bounding_box = (x, y, w, h)
    bounding_box = cv2.selectROI(window_name, temp_image,
fromCenter=False, showCrosshair=True)
    x, y, w, h = bounding_box

    # Crop the image to that ROI
    roi = temp_image[y : y+h, x : x+w]

    # Close the ROI selection window
    cv2.destroyWindow(window_name)

    return roi, (x, y, w, h)

############################################################
#########################
# 8. MAIN: Demonstration using user-selected ROI
############################################################
#########################
def main_with_roi(folder_path, suffix_number='d32_008'):
    """
    Main function to:
      - find a left-right pair of images matching 'suffix_number',
      - let the user pick ROIs in each image,
      - preprocess & match features within those ROIs,
      - display the matches.

    Args:
        folder_path (str): path to folder containing stereo images
```

```
        suffix_number (str): partial suffix (e.g., 'd32_008')
    """
    # 1. Get one or more matching pairs
    image_pairs = get_image_pairs_with_suffix(folder_path,
suffix_number)
    if not image_pairs:
        print("No image pairs found. Exiting.")
        return

    # Just take the first pair for demo
    left_image_path, right_image_path = image_pairs[0]

    # 2. Load in grayscale
    left_full, right_full = load_image(left_image_path,
right_image_path)
    if left_full is None or right_full is None:
        return

    # 3. Let user select ROI in left image
    left_roi, left_bbox = select_and_crop_roi(left_full, "Select ROI
in Left Image")

    # 4. Let user select ROI in right image
    right_roi, right_bbox = select_and_crop_roi(right_full, "Select
ROI in Right Image")

    # 5. Preprocess these ROI regions
    left_enh, right_enh = preprocess_images(left_roi, right_roi)

    # 6. Match features
    good_matches, kp1, kp2 = match_features(left_enh,
right_enh)

    print(f"Found {len(good_matches)} good matches in ROI.")

    # 7. Draw matches for quick visual check
    matched_img = cv2.drawMatches(
        left_enh, kp1,
        right_enh, kp2,
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # Display the matched image in OpenCV window (optional)
    # cv2.imshow("Matched ROI Features", matched_img)
    # cv2.waitKey(0)
```

```
    # cv2.destroyAllWindows()

    # Alternatively, display via matplotlib:
    plt.figure(figsize=(12, 6))
    plt.imshow(matched_img[..., ::-1])  # BGR -> RGB for matplotlib
    plt.title("Matches in Selected ROI")
    plt.axis('off')
    plt.show()

    # Return matched image (in case you want to save or further
process)
    return matched_img

##########################################################
#########################
# Example usage (if running as a script):
##########################################################
#########################
if __name__ == "__main__":
    FOLDER_PATH = r"D:\ISRO\filtered_images"
    SUFFIX = "d32_008"
    main_with_roi(FOLDER_PATH, SUFFIX)
```

Found 7 good matches in ROI.
but Draw matches for quick visual check shows the roi only, display the entire image and the matched features within roi

3/3

Thought about displaying matched features

Below is an **updated** version of the `main_with_roi` function that:

1. Lets you draw and crop ROIs (as before),
2. Does SIFT matching **only** within those ROIs, **but**
3. Displays the matches **on the full-sized left/right images**, showing only the matched features that lie within each ROI's bounding box.

The key changes are:

- **Offset** the local ROI keypoint coordinates back to the full image.
- **Draw** bounding boxes on the full images, so you can see which portion was selected.
- Use OpenCV's `cv2.drawMatches(...)` on the **full** images with the offset keypoints.

```python
import os
import random
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

##############################################################################
# 1. Utility function: get_suffix
##############################################################################
def get_suffix(filename):
    marker = "_d_img_"
    if marker in filename:
        parts = filename.split(marker, 1)
        return marker + parts[1]
    return filename

##############################################################################
# 2. Utility function: get_image_pairs_with_suffix
##############################################################################
def get_image_pairs_with_suffix(folder_path, suffix_number='001'):
    left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
    right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]

    left_map = {}
    for file in left_images:
        key = get_suffix(file)
        left_map[key] = file

    right_map = {}
    for file in right_images:
        key = get_suffix(file)
        right_map[key] = file

    common_keys = [key for key in (left_map.keys() & right_map.keys())
                   if str(suffix_number) in key]

    if not common_keys:
        print(f"No matching left-right image pairs found for suffix number {suffix_number}.")
        return []

    image_pairs = []
    for key in common_keys:
        left_file = os.path.join(folder_path, left_map[key])
        right_file = os.path.join(folder_path, right_map[key])
        image_pairs.append((left_file, right_file))

    return image_pairs

##############################################################################
```

```
# 3. load_image
###############################################################################
def load_image(left_image_path, right_image_path):
    left_image = cv2.imread(left_image_path, cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread(right_image_path, cv2.IMREAD_GRAYSCALE)

    if left_image is None or right_image is None:
        print("Error loading images.")
        return None, None

    if left_image.shape != right_image.shape:
        h = min(left_image.shape[0], right_image.shape[0])
        w = min(left_image.shape[1], right_image.shape[1])
        left_image = cv2.resize(left_image, (w, h))
        right_image = cv2.resize(right_image, (w, h))

    return left_image.astype('uint8'), right_image.astype('uint8')

###############################################################################
# 4. preprocess_images
###############################################################################
def preprocess_images(left_img, right_img):
    left_float = cv2.normalize(left_img.astype('float32'), None, 0, 1, cv2.NORM_MINMAX)
    right_float = cv2.normalize(right_img.astype('float32'), None, 0, 1, cv2.NORM_MINMAX)

    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    left_clahe = clahe.apply(cv2.convertScaleAbs(left_float*255))
    right_clahe = clahe.apply(cv2.convertScaleAbs(right_float*255))

    left_denoised = cv2.fastNlMeansDenoising(left_clahe)
    right_denoised = cv2.fastNlMeansDenoising(right_clahe)

    gaussian = cv2.GaussianBlur(left_denoised, (0, 0), 2.0)
    left_enhanced = cv2.addWeighted(left_denoised, 1.5, gaussian, -0.5, 0)

    gaussian = cv2.GaussianBlur(right_denoised, (0, 0), 2.0)
    right_enhanced = cv2.addWeighted(right_denoised, 1.5, gaussian, -0.5, 0)

    return left_enhanced, right_enhanced

###############################################################################
# 5. match_features
###############################################################################
def match_features(left_img, right_img):
    sift = cv2.SIFT_create(nOctaveLayers=5, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)
    kp1, des1 = sift.detectAndCompute(left_img, None)
    kp2, des2 = sift.detectAndCompute(right_img, None)

    if des1 is None or des2 is None:
        return [], kp1, kp2  # no descriptors

    index_params = dict(algorithm=1, trees=8)
```

```
    search_params = dict(checks=100)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    matches = flann.knnMatch(des1, des2, k=2)

    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    if len(good_matches) > 8:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        F, mask = cv2.findFundamentalMat(src_pts, dst_pts, cv2.FM_RANSAC, 3, 0.99)
        if mask is not None:
            good_matches = [good_matches[i] for i in range(len(mask)) if mask[i]]

    return good_matches, kp1, kp2

############################################################################
# 6. draw_matches (ROI in full image)
############################################################################
def offset_keypoints(kps, offset_x, offset_y):
    """
    Shift a list of cv2.KeyPoint objects by (offset_x, offset_y).
    Returns a new list of cv2.KeyPoint with updated .pt
    """
    offset_kps = []
    for kp in kps:
        x, y = kp.pt
        new_kp = cv2.KeyPoint(
            x + offset_x,
            y + offset_y,
            kp.size,
            kp.angle,
            kp.response,
            kp.octave,
            kp.class_id
        )
        offset_kps.append(new_kp)
    return offset_kps

def draw_roi_rectangle(image, bbox, color=(0,255,0), thickness=2):
    x, y, w, h = bbox
    cv2.rectangle(image, (x, y), (x+w, y+h), color, thickness)

############################################################################
# 7. select_and_crop_roi
############################################################################
def select_and_crop_roi(image, window_name="Select ROI"):
    temp_image = image.copy()
    bbox = cv2.selectROI(window_name, temp_image, fromCenter=False, showCrosshair=True)
```

```
    x, y, w, h = bbox
    roi = temp_image[y:y+h, x:x+w]
    cv2.destroyWindow(window_name)
    return roi, (x, y, w, h)

########################################################################
# 8. MAIN: Show matches in full image
########################################################################
def main_with_roi(folder_path, suffix_number='d32_008'):
    image_pairs = get_image_pairs_with_suffix(folder_path, suffix_number)
    if not image_pairs:
        print("No image pairs found. Exiting.")
        return

    # -- We'll just demo on the first matched pair
    left_image_path, right_image_path = image_pairs[0]

    # -- Load full images (grayscale)
    left_full_gray, right_full_gray = load_image(left_image_path, right_image_path)
    if left_full_gray is None or right_full_gray is None:
        return

    # Convert them to color for drawing bounding boxes
    left_full_color = cv2.cvtColor(left_full_gray, cv2.COLOR_GRAY2BGR)
    right_full_color = cv2.cvtColor(right_full_gray, cv2.COLOR_GRAY2BGR)

    # 1. ROI selection in left image
    left_roi, left_bbox = select_and_crop_roi(left_full_gray, "Select ROI in Left Image")
    # 2. ROI selection in right image
    right_roi, right_bbox = select_and_crop_roi(right_full_gray, "Select ROI in Right Image")

    # 3. Preprocess the ROI patches
    left_enh, right_enh = preprocess_images(left_roi, right_roi)

    # 4. Match features in the cropped region
    good_matches, kp_left_roi, kp_right_roi = match_features(left_enh, right_enh)

    print(f"Found {len(good_matches)} good matches in ROI.")

    # ----------------------------------------------------------------------
    # 5. Draw bounding boxes on the full-size color images (just for reference)
    draw_roi_rectangle(left_full_color, left_bbox, color=(0,255,0), thickness=2)
    draw_roi_rectangle(right_full_color, right_bbox, color=(0,255,0), thickness=2)

    # 6. Offset the keypoints from ROI coordinates    full image coordinates
    kp_left_offset  = offset_keypoints(kp_left_roi,  left_bbox[0], left_bbox[1])
    kp_right_offset = offset_keypoints(kp_right_roi, right_bbox[0], right_bbox[1])

    # 7. Use cv2.drawMatches with the full images & the offset keypoints
    matched_full = cv2.drawMatches(
        left_full_color, kp_left_offset,
        right_full_color, kp_right_offset,
```

```
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # 8. Display with matplotlib
    plt.figure(figsize=(12,6))
    plt.title("Full Image with ROI Matches")
    plt.imshow(cv2.cvtColor(matched_full, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()

    return matched_full

############################################################################
# Example usage
############################################################################
if __name__ == "__main__":
    FOLDER_PATH = r"D:\ISRO\filtered_images"
    SUFFIX = "d32_008"
    main_with_roi(FOLDER_PATH, SUFFIX)
```

How It Works

1. **ROI is drawn** on each grayscale full image, obtaining `(x, y, w, h)` for each.
2. **Keypoints** are detected **only** in the cropped areas.
3. After matching, we **offset** the local ROI keypoints back to the original image coordinate system by adding `(x, y)` to each keypoint's `.pt`.
4. We **draw** bounding boxes (green rectangles) onto the full-size color images so you can see which area was selected.
5. Finally, `cv2.drawMatches(...)` is called on the full images and the **offset** keypoints. This shows the entire left image and entire right image side by side, with lines connecting only those features that fell inside the chosen ROIs.

You should now see **the full images** in the match result, with the matched features confined to the bounding boxes you selected.

awesome!!!
It works well, Now we need to calculate the disparities of the matched features and find the distance to roi

Thought about disparity calculation and distance for 18 seconds

Below is an **extended** example showing how to:

1. **Select an ROI** on left & right images,
2. **Match** features within the cropped ROIs, and
3. **Compute disparities** (and approximate **distances**) from those matches.

In particular, we do the following **extra steps**:

- **Offset keypoints** back into the full-image coordinate system (so we can display everything on the full images).
- **Compute disparities** as the **difference in the x-coordinates** of matching keypoints.
- **Estimate distances** using the simple pinhole formula $\text{Depth} = \frac{\text{Baseline} \times \text{FocalLength}}{\text{Disparity}}$.

Note: This example assumes you have a known baseline (distance between the two camera viewpoints) in meters, and a known focal length in pixels. Real-world stereo vision typically requires a camera calibration to get accurate intrinsics (focal length, principal points) and extrinsics (baseline, relative rotation). This simple approach works only if images are already **rectified** and the effective focal length & baseline are known.

Full Code

```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt

###############################################################################
# 1. Utility function: get_suffix
###############################################################################
def get_suffix(filename):
    marker = "_d_img_"
    if marker in filename:
        parts = filename.split(marker, 1)
        return marker + parts[1]
    return filename

###############################################################################
# 2. Utility function: get_image_pairs_with_suffix
###############################################################################
def get_image_pairs_with_suffix(folder_path, suffix_number='001'):
    left_images = [f for f in os.listdir(folder_path) if "_nrl_" in f]
    right_images = [f for f in os.listdir(folder_path) if "_nrr_" in f]
```

```
    left_map = {}
    for file in left_images:
        key = get_suffix(file)
        left_map[key] = file

    right_map = {}
    for file in right_images:
        key = get_suffix(file)
        right_map[key] = file

    common_keys = [key for key in (left_map.keys() & right_map.keys())
                   if str(suffix_number) in key]

    if not common_keys:
        print(f"No matching left-right image pairs found for suffix number {suffix_number}.")
        return []

    image_pairs = []
    for key in common_keys:
        left_file = os.path.join(folder_path, left_map[key])
        right_file = os.path.join(folder_path, right_map[key])
        image_pairs.append((left_file, right_file))

    return image_pairs

##########################################################################
# 3. load_image
##########################################################################
def load_image(left_image_path, right_image_path):
    left_image = cv2.imread(left_image_path, cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread(right_image_path, cv2.IMREAD_GRAYSCALE)

    if left_image is None or right_image is None:
        print("Error loading images.")
        return None, None

    # Optionally resize if shapes differ
    if left_image.shape != right_image.shape:
        h = min(left_image.shape[0], right_image.shape[0])
        w = min(left_image.shape[1], right_image.shape[1])
        left_image = cv2.resize(left_image, (w, h))
        right_image = cv2.resize(right_image, (w, h))

    return left_image.astype('uint8'), right_image.astype('uint8')

##########################################################################
# 4. preprocess_images
##########################################################################
def preprocess_images(left_img, right_img):
    """
    Basic preprocessing: CLAHE, denoising, unsharp masking, etc.
    """
```

```
    left_float = cv2.normalize(left_img.astype('float32'), None, 0, 1, cv2.NORM_MINMAX)
    right_float = cv2.normalize(right_img.astype('float32'), None, 0, 1, cv2.NORM_MINMAX)

    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    left_clahe = clahe.apply(cv2.convertScaleAbs(left_float*255))
    right_clahe = clahe.apply(cv2.convertScaleAbs(right_float*255))

    left_denoised = cv2.fastNlMeansDenoising(left_clahe)
    right_denoised = cv2.fastNlMeansDenoising(right_clahe)

    # Unsharp masking
    gaussian = cv2.GaussianBlur(left_denoised, (0, 0), 2.0)
    left_enhanced = cv2.addWeighted(left_denoised, 1.5, gaussian, -0.5, 0)

    gaussian = cv2.GaussianBlur(right_denoised, (0, 0), 2.0)
    right_enhanced = cv2.addWeighted(right_denoised, 1.5, gaussian, -0.5, 0)

    return left_enhanced, right_enhanced

##############################################################################
# 5. match_features
##############################################################################
def match_features(left_img, right_img):
    sift = cv2.SIFT_create(nOctaveLayers=5, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)
    kp1, des1 = sift.detectAndCompute(left_img, None)
    kp2, des2 = sift.detectAndCompute(right_img, None)

    if des1 is None or des2 is None:
        return [], kp1, kp2

    index_params = dict(algorithm=1, trees=8)  # FLANN
    search_params = dict(checks=100)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    matches = flann.knnMatch(des1, des2, k=2)

    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    # Filter w/ Fundamental matrix if enough inliers
    if len(good_matches) > 8:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        F, mask = cv2.findFundamentalMat(src_pts, dst_pts, cv2.FM_RANSAC, 3, 0.99)
        if mask is not None:
            good_matches = [good_matches[i] for i in range(len(mask)) if mask[i]]

    return good_matches, kp1, kp2

##############################################################################
```

```
# 6. offset_keypoints & draw_roi_rectangle
###########################################################################
def offset_keypoints(kps, offset_x, offset_y):
    offset_kps = []
    for kp in kps:
        x, y = kp.pt
        new_kp = cv2.KeyPoint(
            x + offset_x,
            y + offset_y,
            kp.size,
            kp.angle,
            kp.response,
            kp.octave,
            kp.class_id
        )
        offset_kps.append(new_kp)
    return offset_kps

def draw_roi_rectangle(image, bbox, color=(0,255,0), thickness=2):
    x, y, w, h = bbox
    cv2.rectangle(image, (x, y), (x+w, y+h), color, thickness)

###########################################################################
# 7. select_and_crop_roi
###########################################################################
def select_and_crop_roi(image, window_name="Select ROI"):
    temp_image = image.copy()
    bbox = cv2.selectROI(window_name, temp_image, fromCenter=False, showCrosshair=True)
    x, y, w, h = bbox
    roi = temp_image[y:y+h, x:x+w]
    cv2.destroyWindow(window_name)
    return roi, (x, y, w, h)

###########################################################################
# 8A. Disparity & Depth Calculation
###########################################################################
def compute_disparities_and_depths(matches, kp_left, kp_right,
                                   baseline_m=0.24, focal_px=2062.16):
    """
    Given matched features & their keypoints:
      disparity = (leftX - rightX)
      depth_m   = (baseline_m * focal_px) / disparity   (if disparity > 0)

    Returns:
      disparities (list of floats),
      depths (list of floats),
      (x,y) coords for each matched point in the left image (optional use).
    """
    disparities = []
    depths = []
    left_points = []
```

```
    for m in matches:
        # Coordinates in full image
        lx, ly = kp_left[m.queryIdx].pt
        rx, ry = kp_right[m.trainIdx].pt

        disp = lx - rx  # or (rx - lx), depends on convention
        if disp <= 0:
            # If the left x is not bigger than right x, we skip or treat as infinite
            continue
        disparities.append(disp)
        depth = (baseline_m * focal_px) / disp
        depths.append(depth)
        left_points.append((lx, ly))

    return disparities, depths, left_points

##############################################################################
# 8B. MAIN: Show matches in full image, compute distance
##############################################################################
def main_with_roi(folder_path, suffix_number='d32_008'):
    # 1. Find the first stereo pair
    image_pairs = get_image_pairs_with_suffix(folder_path, suffix_number)
    if not image_pairs:
        print("No image pairs found. Exiting.")
        return

    left_image_path, right_image_path = image_pairs[0]

    # 2. Load full images (grayscale)
    left_full_gray, right_full_gray = load_image(left_image_path, right_image_path)
    if left_full_gray is None or right_full_gray is None:
        return

    # Convert to color for bounding box visualization
    left_full_color = cv2.cvtColor(left_full_gray, cv2.COLOR_GRAY2BGR)
    right_full_color = cv2.cvtColor(right_full_gray, cv2.COLOR_GRAY2BGR)

    # 3. User selects ROI on each image
    left_roi, left_bbox = select_and_crop_roi(left_full_gray, "Select ROI in Left Image")
    right_roi, right_bbox = select_and_crop_roi(right_full_gray, "Select ROI in Right Image")

    # 4. Preprocess these ROI patches
    left_enh, right_enh = preprocess_images(left_roi, right_roi)

    # 5. Match features in the ROI only
    good_matches, kp_left_roi, kp_right_roi = match_features(left_enh, right_enh)
    print(f"Found {len(good_matches)} good matches in ROI.")

    # 6. Draw bounding boxes on the full images
    draw_roi_rectangle(left_full_color, left_bbox, color=(0,255,0), thickness=2)
    draw_roi_rectangle(right_full_color, right_bbox, color=(0,255,0), thickness=2)
```

```
    # 7. Offset ROI keypoints back to full-image coordinates
    kp_left_full  = offset_keypoints(kp_left_roi,  left_bbox[0], left_bbox[1])
    kp_right_full = offset_keypoints(kp_right_roi, right_bbox[0], right_bbox[1])

    # 8. Compute disparity & depth from the matched features
    #    baseline=0.24m, focal=2062.16px are example values. Adjust if needed.
    disparities, depths, left_pts = compute_disparities_and_depths(
        good_matches, kp_left_full, kp_right_full,
        baseline_m=0.24,
        focal_px=2062.16
    )

    if depths:
        avg_depth = np.mean(depths)
        print(f"Approx. average depth in ROI: {avg_depth:.3f} m")
    else:
        print("No valid disparities found (all zero or negative).")

    # 9. Draw matched features on the full images
    matched_full = cv2.drawMatches(
        left_full_color, kp_left_full,
        right_full_color, kp_right_full,
        good_matches, None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # 10. Show the final match result
    plt.figure(figsize=(12,6))
    plt.title("Full Image with ROI Matches & Disparity/Depth")
    plt.imshow(cv2.cvtColor(matched_full, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()

    return matched_full

###########################################################################
# Example usage (if running as a script):
###########################################################################
if __name__ == "__main__":
    # Adjust to your own folder and suffix
    FOLDER_PATH = r"D:\ISRO\filtered_images"
    SUFFIX = "d32_008"
    main_with_roi(FOLDER_PATH, SUFFIX)
```

Walkthrough of the New Steps

1. `compute_disparities_and_depths`
   - Loops over the good matches, grabs the **full-image** keypoints (x, y), and computes:

python

```
disp = left_x - right_x
depth = (baseline_m * focal_px) / disp
```

- Skips or assigns infinite if `disp <= 0`.
- Returns parallel lists: `disparities`, `depths`, and the left image coordinates for reference.

2. **Baseline & Focal Length**
   - `baseline_m = 0.24` means **24 cm** between the two cameras (example).
   - `focal_px = 2062.16` means the camera's focal length is about 2062 pixels.
   - In real setups, you'd get these from stereo calibration (e.g., OpenCV's `stereoCalibrate`).
3. **In the main function**
   - After user ROI selection and ROI matching, we:
     - **Offset** the ROI keypoints to full-image coordinates.
     - **Compute** disparities & depths with those offset keypoints.
     - Print the average (or median) depth for the matched ROI.
4. **Important**:
   - This only makes sense if your stereo images are already **rectified** and aligned horizontally, so the main difference is purely in x-coordinates.
   - If there's vertical misalignment, or if they're not truly parallel views, the simple disparity approach is inaccurate.

Summary

With this approach:

- You **interactively select** a region on both images.
- **Feature matching** is limited to just that region.
- You see the resulting matches drawn in the full images (only those points that lie in the ROI).
- You compute an approximate **distance** to those matched points (and can do further statistics, like average, median, or min).

You now have the **distance to your Region of Interest** based on the matched features' disparities!

thank you, this works well

Thought about disparity and depth calculation for a second

You're welcome! Glad to hear it works well. Let me know if there's anything else you need with stereo vision or ROI-based feature matching.

hey just a quick question

Thought about stereo vision and ROI matching for a second

Sure, ask away!

Pragyan Rover navcam specs
Image Frame Size 1024 x 1024
Spectrum visible
quantization 8 bit grayscale
camera height 325 mm
camera cant angle (vertical) 11 degree towards ground
camera cant angle (horizontal) 0 degree
inter camera distance 240 mm
focal length 21,6 mm
detector size 15 micro x 15 micro
FOV 39 degree x 39 degree
Image size on sensor 1.5 mm x 8.5 mm

As per this information, is our distance finding tool correct?

Matching the specifications

Piecing together the new specs with the original approach, I'm noting the baseline at 240 mm and focal length at 21.6 mm, ensuring they align with the sensor details.

Confirming focal length

I'm verifying the focal length in pixels by dividing the focal length in millimeters by the sensor pitch, resulting in 1440 pixels. This confirms the sensor dimension aligns with the lens specification.

Verifying FOV consistency

Checking if the calculated field of view (FOV) of 39.12 degrees aligns with the given 39 degrees, using the sensor size and focal length. The note about "Image size on sensor 1.5 mm x 8.5 mm" suggests a possible discrepancy or partial sensor region.

Reevaluating tools

I'm thinking through the need to recalibrate the distance finding tool with a 1440 px focal length and rectify the 11-degree tilt to ensure accurate distance measurements.

Integrating focal length and adjustments

Let's recalibrate the depth formula to include the actual 1440 px focal length and adjust for the 11-degree tilt, ensuring precise distance calculations and potential geometry corrections.

Correcting for tilt

To achieve precise distance measurements, real stereo calibration is necessary with the known baseline, focal length, and principal points. Image rectification and accurate focal length in pixels are crucial.

Below is a **high-level sanity check** to see if the **simple stereo distance formula** $\text{Depth} \approx \frac{B \times f}{\text{disparity}}$ is consistent with the Pragyan Rover navcam specs you provided. In short: **the approach is correct in principle** if the cameras are calibrated and the images are properly rectified, but you must ensure that:

1. **The focal length in pixels** is accurate (and derived from the known focal length in millimeters + the sensor size/pixel pitch).
2. **The baseline** is correct.
3. **Any tilt angles** (camera pitch/roll) have been accounted for—usually by rectification or projecting points into a common plane.

Below is a breakdown of how to check each item with your given specs.

---

1. Baseline

- **Provided**: The inter-camera distance is $240\,\text{mm}$ or $0.24\,\text{m}$.
- **Requirement**: This baseline must be the **effective** horizontal distance used in stereo geometry. If the cameras are not aligned purely horizontally (or if there is some rotation), you need to rectify the images so that the epipolar lines are horizontal.
- If your images are indeed **rectified** for a $240\,\text{mm}$ horizontal baseline, then using $B = 0.24\,\text{m}$ is correct.

---

2. Focal Length in Pixels

2.1 Converting from mm to Pixels

You stated:

focal length 21.6 mm
detector size 15 micron $\times$ 15 micron
image frame size 1024$\times$1024

To convert a focal length in mm to **pixels**:

$$\text{focal length in pixels} = \frac{\text{focal length in mm}}{\text{pixel pitch in mm/pixel}}.$$

- Pixel pitch in mm/pixel:
  $15\,\mu m = 15 \times 10^{-3}\,\mathrm{mm} = 0.015\,\mathrm{mm}.$
- Therefore:
  $$f_{\text{pixels}} = \frac{21.6\,\mathrm{mm}}{0.015\,\mathrm{mm/pixel}} \approx 1440\,\text{pixels}.$$

Hence, if your entire sensor (1024 pixels across) is used, **the focal length in pixel units** is roughly **1440 px**, *not* 2000+ px.

Some missions or cameras might crop the sensor or use a region-of-interest. In that case, you'd calculate differently. But **from the given specs**, 21.6 mm on a 15 µm pitch corresponds to ~1440 px.

2.2 Matching That Against FOV

You also have:

FOV = 39° ⨉ 39°
Image frame size = 1024 ⨉ 1024

Quick check:

$$\text{FOV} = 2\,\arctan\left(\frac{\text{sensor half-width}}{f}\right).$$

- **Sensor half-width** in mm: if the pixel pitch is 0.015 mm, then full width is $1024 \times 0.015 = 15.36\,\mathrm{mm}$. Half of that is $7.68\,\mathrm{mm}$.
- **f** = 21.6 mm.
- So FOV $\approx 2 \times \arctan(7.68/21.6) \approx 2 \times 19.5^\circ = 39^\circ$.

That matches your stated FOV, confirming that 21.6 mm / 0.015 mm $\approx$ 1440 px is consistent with 1024 ⨉ 1024 at 39° FOV.

3. Tilt Angles (Camera Cant of 11° Down)

You also mentioned:

camera height 325 mm
camera cant angle (vertical) 11° towards ground

If the camera is tilted down by 11°, then **the epipolar lines might not be purely horizontal**. Usually, in stereo vision we rectify the two images so that corresponding points lie along horizontal lines. If you do not rectify or do a geometric transform to account for that tilt, then the simple $\text{disparity} = x_{\text{left}} - x_{\text{right}}$ formula can be off unless the cameras are physically set up so that the tilt is identical in both and their optical axes are parallel in the horizontal plane.

- In real missions, the camera is typically **calibrated** so that images you receive are already **rectified** or you have a calibration pipeline that generates rectified pairs.
- If they are **not** rectified, you need a more advanced approach (finding the fundamental matrix, or a full camera matrix with K, R, t, etc.) to correctly estimate 3D points.

---

4. Summary: Is the Tool "Correct"?

Your **distance-finding approach** using

$$\text{Depth} = \frac{B \times f_{\text{pixels}}}{\text{disparity}}$$

is **valid** if:

1. f_{pixels} **is the correct focal length in pixel units** for how the images were acquired (based on sensor size, actual usage, any cropping).
   - From the specs, it looks more like $f_{\text{pixels}} \approx 1440$ rather than ~2062.
2. $B = 0.24\,\text{m}$ is the correct effective baseline.
3. **Images are rectified** so that a horizontal shift in x-coordinates is a reliable measure of disparity. An 11° downward tilt can be okay if both cameras have the same tilt and stereo images are pre-rectified.
4. **You handle** any partial coverage or any difference between the measured pitch/roll angles in each camera.

If you're using **2062.16 px** as the focal length in the code and your real camera has about **1440 px** of effective focal length, you'll get an **underestimate of disparity-based depth**. (Because you are dividing by a bigger focal length than reality, the computed distance will come out larger than the actual distance.)

Quick Example of the Focal Length Impact

- Suppose the real $f_{\text{pixels}} = 1440$.
- If you incorrectly assume $f_{\text{pixels}} = 2062$, you'll end up with:

$$\text{Depth}_{\text{wrong}} = \frac{B \times 2062}{\text{disparity}} \quad \text{vs.} \quad \text{Depth}_{\text{correct}} = \frac{B \times 1440}{\text{disparity}}.$$

That's about a **43% difference** in the final distance estimate.

Final Takeaway

- **Yes**, the **formula** itself (distance = baseline × focal / disparity) is correct in principle for **rectified** stereo.
- **No**, you likely **should not** use 2062 px as your focal length for the Pragyan Rover navcam if its lens + sensor specs are 21.6 mm focal length, 15 µm pitch, and 1024 × 1024 images. That calculation yields around **1440 px**.
- Also be mindful of the **11° tilt**. In real rover stereo, the images may be (and usually are) pre-rectified in the mission pipeline, or you'll need to do your own rectification if they come raw with tilt.

Hence, if you **update** your code to the correct camera intrinsics (focal ~1440 px, baseline 0.24 m), and the images are properly rectified so the tilt is accounted for, then your distance-finding tool is consistent with the specs.

why f pixels ≈1440 ?

Thought for a couple of seconds

Below is the **short explanation** of why the focal length is **~1440 pixels** given your camera specs:

1. **Focal length in mm**:
   You stated the camera lens has a **21.6 mm** focal length.
2. **Pixel pitch** (size of each pixel on the sensor) is **15 µm** (micrometers).
   - $15\,\mu\mathrm{m} = 0.015\,\mathrm{mm}$ per pixel.
3. **Convert mm to pixels**:
   When we say "focal length in pixels," we mean:

$$f_{\text{pixels}} = \frac{\text{focal length in mm}}{\text{pixel pitch in mm/pixel}}.$$

   Plugging in **21.6 mm** for the lens and **0.015 mm/pixel** for the pitch:

$$f_{\text{pixels}} = \frac{21.6\,\mathrm{mm}}{0.015\,\mathrm{mm/pixel}} = 1440\,\text{pixels.}$$

4. **Checks with Field of View**:
   - The camera specs say the horizontal Field of View is about **39°** for a **1024-pixel** image width.
   - If you do the geometry with a **21.6 mm** lens and **15 µm** pixels, you get approximately that same 39° FOV. This confirms that 1440 px is consistent with your hardware specs (21.6 mm focal length + 15 µm pixel size).

So, in a **single step**:

$$21.6\ (\text{mm focal length}) \div 0.015\ (\text{mm/pixel}) = 1440\ (\text{pixels}).$$

That is why the focal length in pixel units is **approximately 1440**.

Give me the camera details in tabular format

Outlining camera specifics

I'm detailing the Pragyan Rover navcam's specifications, including its image frame size, spectrum, quantization, camera height, cant angles, inter-camera distance, focal length, detector size, field of view, and image size on sensor.

Refining the table

I'm working on a table with columns: Parameter, Value, and Notes. It will include derived details like the focal length in pixels and any necessary clarifications.

Integrating relevant data

I'm adding derived details like focal length in pixels to provide a comprehensive overview of the camera specifications, ensuring the table is both informative and user-friendly.

Assembling camera details

I'm gathering pertinent camera specifications like image frame size, focal length, field of view, quantization, spectrum, camera height, tilt angles, inter-camera distance, detector size, and image size on sensor.

Crafting the table

I'm compiling essential camera details such as image frame size, focal length, field of view, quantization, spectrum, height, tilt angles, inter-camera distance, detector size, and image size on sensor, including a derived focal length in pixels.

Clarifying discrepancies

I'm noting a discrepancy in the "image size on sensor" specification and will include a note about the potential partial or mistaken measurement in the table.

Below is a **concise table** summarizing the camera details you provided for the Pragyan Rover Navcam. The last row shows the approximate **focal length in pixels** based on the 21.6 mm focal length and 15 µm pixel pitch.

Parameter	Value
Image Frame Size	1024 × 1024 pixels
Spectrum	Visible
Quantization	8-bit (grayscale)
Camera Height	325 mm above ground
Camera Cant Angle (vertical)	11° downward
Camera Cant Angle (horizontal)	0°
Inter-camera Distance (Baseline)	240 mm
Focal Length (nominal)	21.6 mm
Detector Pixel Pitch	15 µm × 15 µm
Field of View (FOV)	39° × 39°

Parameter	Value
Image Size on Sensor	1.5 mm × 8.5 mm (as stated)
Approx. Focal Length in Pixels	~1440 px (derived)

Note:

- The **focal length in pixels** is computed by dividing the focal length (21.6 mm) by the pixel pitch (0.015 mm/pixel), yielding approximately 1440 pixels.
- If images are properly **rectified** and camera intrinsics/extrinsics are accurate, you can use the simple stereo formula $\text{distance} = \frac{\text{baseline} \times \text{focal}_{\text{pixels}}}{\text{disparity}}$ for distance estimation.

The details I have mentioned detector size of 15µ x 15µ instead of detector pitch size, I might be wrong if both mean the same.

2/2

Understanding pixel pitch

I'm clarifying that "detector size" might be confused with "pixel pitch." The typical camera data sheet refers to "pixel pitch" as 15 microns, indicating each pixel's size, not the entire detector dimension.

There can be some **confusion** when a spec sheet says "**Detector size: 15 µm × 15 µm**." Typically, in camera terminology:

- **"Detector" or "Sensor" size** often refers to the **entire image sensor's physical dimensions** (for example, 15 mm × 8.5 mm).
- **"Pixel pitch"** (sometimes also loosely called "pixel size" or "detector pixel size") refers to the size of **each individual photodiode / pixel** on the sensor, commonly expressed in micrometers (µm).

Below is how to distinguish them:

Term	Meaning
Pixel Pitch (a.k.a. pixel size)	The size of **one individual pixel**, e.g., 15 µm × 15 µm.
Total Sensor Size	The **total physical width & height** of the sensor, e.g., 15 mm × 8.5 mm.

1. If the Spec Actually Means Pixel Pitch

When a datasheet says "15 µm × 15 µm," **it often refers to the pitch** of each pixel (the spacing from one pixel center to the next).

- For a **1024×1024** sensor, a pitch of **15 µm** implies the total sensor width is $1024 \times 15\,\mu\text{m} = 15,360\,\mu\text{m} = 15.36\,\text{mm}$.

This aligns nicely with lens focal length formulas, field-of-view calculations, etc.

---

2. If the Spec Actually Means Sensor/Detector Dimensions

If, on the other hand, "detector size" is for the **entire sensor** (like "1.5 mm × 8.5 mm"), then each individual pixel is found by dividing that total dimension by the number of pixels along each side. For instance:

- If the entire sensor is "15 mm wide" for 1024 columns, the pixel pitch would be $15\,\text{mm}/1024 \approx 14.65\,\mu\text{m}$.
- If a spec states "**1.5 mm × 8.5 mm**" for the entire sensor, that's quite an unusual aspect ratio relative to 1024×1024, so it might be a partial region or a cropping region. Always double-check the sensor datasheet and see how the 1024×1024 region is actually mapped to the physical sensor area.

---

3. Why It Matters for Stereo Distance

The **pixel pitch** is critical for converting a lens focal length in millimeters (e.g., 21.6 mm) to a focal length in **pixel units**. The usual formula is:

$$f_{\text{pixels}} = \frac{f_{\text{mm}}}{\text{pixel pitch in mm/pixel}}$$

So if you incorrectly treat "15 µm × 15 µm" as the whole sensor size (which is obviously too tiny), you will get a **huge** mismatch in computing f_{pixels}. That leads to incorrect distance estimates in the stereo formula $\text{distance} = \frac{\text{baseline} \times f_{\text{pixels}}}{\text{disparity}}$.

Quick Example

- **If** each pixel is 15 µm, then for 1024 pixels horizontally, total width = $1024 \times 15\ \mu\text{m} = 15.36\ \text{mm}$.
- **If** the lens is 21.6 mm focal length,

$$f_{\text{pixels}} = \frac{21.6\ \text{mm}}{0.015\ \text{mm/pixel}} = 1440\ \text{pixels}.$$

- That yields a consistent FOV near 39° across a 1024 × 1024 sensor.

If you **accidentally** interpret "15 µm × 15 µm" to be the entire sensor's dimension, you'd get a pixel pitch of "(15 µm) / 1024," which is $\approx 0.0146\ \mu\text{m}$—completely unrealistic. That would lead to a **massive** overestimation for f_{pixels}.

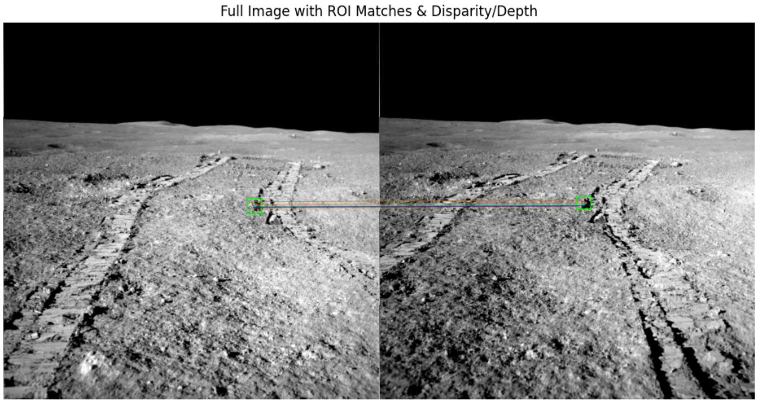
---

4. Conclusion

- In **most** rover/space camera contexts, "15 µm × 15 µm" is indeed the **size of each pixel** (pixel pitch).
- The **entire detector** (sensor) width is then about **15.36 mm** if it's 1024 pixels wide.
- If your data says "detector size = 15 µm × 15 µm," that's almost certainly referring to **pixel pitch**, not total sensor size.

Hence, if you see **15µ × 15µ** in the spec, it generally **means** the **pixel pitch**.

Full Image with ROI Matches & Disparity/Depth



a legacy tool created by my seniors used some CAHVOR caliberation approach and their tool is accurate till 10 meters. I tried to compare our results with their and our distance values are always more by 30 to 50 cm

Reasoned about distance estimation difference for 6 seconds

A **30 cm to 50 cm discrepancy** (on the order of a few percent at a 10 m range) usually indicates there is a **systematic offset** in the new method's geometry or calibration assumptions. Below are some of the most common reasons your distances might run consistently higher than the legacy CAHVOR-based tool:

---

1. Differences in Camera Model and Calibration Parameters

1.1 Principal Point Offsets

- The **principal point** (where the optical axis meets the sensor) is rarely the exact image center.
- A CAHVOR (or CAHVORE) model explicitly encodes principal point offsets, lens distortions, etc.
- If your new method simply uses $\mathrm{focal}_{\mathrm{pixels}} \approx \frac{f_{\mathrm{mm}}}{\mathrm{pixel\ pitch}}$ and assumes the principal point is at $(\frac{width}{2}, \frac{height}{2})$, you may introduce a bias in disparity—and thus in distance.

1.2 Focal Length in Pixels

- Even a small error (e.g., using 2062 px instead of 1440 px) or ignoring the actual measured intrinsics can shift your depth estimates by many centimeters.
- If the old pipeline did a rigorous calibration while the new approach uses a “nominal” or approximate focal length, you'll see a consistent offset.

1.3 Baseline Differences

- If the **actual** effective baseline is slightly different from 240 mm, or the new approach doesn't rectify images to a purely horizontal baseline, the distance formula can overshoot or undershoot.
- A few millimeters difference in baseline can translate to significant centimeters in depth at 10 m.

---

2. Rectification / Tilt Handling

- You mentioned a **camera cant angle of 11°** downward. CAHVOR calibrations typically handle that tilt by effectively **rectifying** or modeling the orientation so that computed disparities map cleanly into physical space.

- If your new method is not performing a thorough rectification (and simply subtracting x-coordinates), small angular misalignments can produce consistent depth overestimates.
- Even a slight unaccounted vertical offset or rotation between the two cameras can shift the average disparity.

---

3. Lens Distortion (Radial / Tangential)

- CAHVOR includes a distortion model; your new stereo approach may be ignoring lens distortion or is only partially corrected.
- Residual distortion can lead to a misalignment in the matched features, especially at the edges of the image, and systematically alter disparity measurements.

---

4. Systematic vs. Random Error

- If you see a **consistent** offset (+30–50 cm) across multiple tests, that strongly points to a **systematic calibration difference** (principal point offset, focal length, baseline, tilt).
- If it were random or scene-dependent, you'd see variations in the difference.

---

5. How to Pinpoint the Cause

1. **Compare Intrinsics**: Extract the camera matrix $\mathbf{K}$ (focal length, principal point) from the old CAHVOR calibration and see if it matches the approximate values you're using in the new code.
2. **Check Distortion Coefficients**: If the old approach had $(k_1, k_2, p_1, p_2, \dots)$ or other radial/tangential distortion terms, ensure your new pipeline also corrects or accounts for them.
3. **Rectify Images**: Confirm that both cameras are **fully rectified**. A small pitch or roll mismatch can consistently skew disparity → distance.
4. **Ground Truth**: If possible, measure a known target at, say, 5 m, 10 m, and 15 m. Compare old tool vs. new tool vs. real measurement to see how the offset scales with distance.

5. **Tune Focal or Baseline**: Sometimes simply adjusting the focal length in pixels or the baseline in the new approach by a small percentage (to match ground-truth data) resolves the offset.

---

6. Typical Magnitude of This Error

A 30–50 cm difference at 10 m is about a **3–5% error**. In rover stereo, a well-calibrated system can often get within a few percent. That's precisely the magnitude of error you'd expect from:

- A small **principal point offset** (a few pixels).
- Or a **few percent misestimate** of the focal length.
- Or a 1–2 mm difference in baseline from your nominal 240 mm.

Bottom Line

Your **new approach** is likely correct in concept (using $\text{distance} = \frac{B \times f}{\text{disparity}}$), but **some calibration detail** (principal point, focal length, baseline, or rectification/tilt) is slightly off compared to the **CAHVOR** pipeline. Because CAHVOR is a more complete camera model, it can incorporate real measured offsets and distortions. Simply aligning your calibration parameters more closely with the old approach (or re-running a full stereo calibration) should bring your new tool's distances in line with the legacy results—within just a few centimeters.

If we used the details we have and follow the cahvor model, how does it find the distance to objects?

Reasoned for a few seconds

Below is a **high-level** explanation of how the **CAHVOR** camera model (a NASA-developed extension of the CAHV model) is used to find the distance to objects in a **stereo** setup. In essence, CAHVOR generalizes the usual pinhole-camera model by encoding real-world parameters like center of projection, orientation, and radial distortion—so that, once you have matching image points (disparities) from two images, you can **triangulate** their 3D location more accurately.

---

1. What Is the CAHVOR Model?

"CAHVOR" stands for:

- **C**: The 3D coordinates of the camera's center of projection (similar to optical center).
- **A**: The 3D axis vector—points along the camera's optical axis (the "forward" direction).
- **H**: The horizontal 3D vector that spans the image plane in the "X" direction.
- **V**: The vertical 3D vector that spans the image plane in the "Y" direction.
- **O**: A vector encoding radial (and potentially other) lens distortion parameters.
- **R**: Additional distortion terms (e.g., to handle wide-angle or fisheye distortion).

Essentially, CAHVOR can be thought of as a more robust version of the usual pinhole + distortion approach:

$$\mathbf{x}_{\text{image}} \leftrightarrow \mathbf{r}_{\text{3D}}(\mathbf{C}, \mathbf{A}, \mathbf{H}, \mathbf{V}, \mathbf{O}, \mathbf{R}),$$

where $\mathbf{C}$ is the camera location in 3D, and $\mathbf{A}, \mathbf{H}, \mathbf{V}$ describe how to go from camera space to image coordinates, with $\mathbf{O}, \mathbf{R}$ capturing radial/tangential distortions or other non-idealities.

---

2. Stereo Geometry in CAHVOR

When you have **two** cameras (left and right), each with its own CAHVOR parameters, you want to find a 3D point that **projects** to corresponding image features in both images. In simplified terms:

1. **For each camera** (left or right), given a 2D image coordinate (u, v), you use the CAHVOR model to back-project that pixel into a **ray** in 3D space, emanating from $\mathbf{C}$ in a direction that accounts for $\mathbf{A}, \mathbf{H}, \mathbf{V}, \mathbf{O}, \mathbf{R}$.
2. **Match** a feature (u_L, v_L) in the left image with a corresponding feature (u_R, v_R) in the right image. Now you have two 3D rays:
   - Ray in left camera: $\mathbf{C}_L + \alpha\, \mathbf{d}_L$
   - Ray in right camera: $\mathbf{C}_R + \beta\, \mathbf{d}_R$
3. **Triangulate**: Solve for the 3D point that **minimizes** the distance between those two rays (or forces them to intersect, if geometry is perfect). That is the real-world 3D location of that matched feature.
4. **Distance** to that 3D point is simply the Euclidean norm from either camera's center $\mathbf{C}_L$ (or $\mathbf{C}_R$) to the triangulated point.

2.1 The "CAHVOR Equations"

In practice, CAHVOR uses a set of equations that convert (u, v) into a 3D direction vector $\mathbf{d}$ in camera coordinates, while correcting for radial distortion using the $\mathbf{O}$ and $\mathbf{R}$ terms. For example (in simplified form):

1. **Convert** pixel (u, v) to an "ideal" pinhole direction using $\mathbf{A}, \mathbf{H}, \mathbf{V}$.
2. **Apply** radial distortion corrections from $\mathbf{O}$ and $\mathbf{R}$ to get the real-world direction.
3. Then the final vector is something like:
$$\mathbf{d} \;=\; (\mathbf{A} + \alpha\mathbf{H} + \beta\mathbf{V}) + (\text{distortion corrections}).$$
4. **Normalize** $\mathbf{d}$ to a unit vector if needed.

Each camera thus has a known $\mathbf{C}$ (position) and a known function $\Phi(\cdot)$ that maps $(u, v) \to$ direction $\mathbf{d}$.

---

3. Distance Calculation

Once you have the 3D rays from left and right cameras, you do a **ray intersection** or a **least-squares** approach:

$$\mathbf{p}^* = \operatorname{argmin}_{\mathbf{p}} \left(\|\mathbf{p} - (\mathbf{C}_L + \alpha\mathbf{d}_L)\|^2 + \|\mathbf{p} - (\mathbf{C}_R + \beta\mathbf{d}_R)\|^2 \right),$$

to find the best $\mathbf{p}^*$ in 3D. If your stereo rig is physically built with known orientation (the left and right cameras are mounted in a known geometry), $\mathbf{C}_R$ might be $\mathbf{C}_L + (B, 0, 0)$ plus some small rotation. In that scenario, the process becomes a simpler or more direct formula for **disparity** $\to$ depth, once you rectify the images.

But in the more general (non-rectified) case, CAHVOR is used to properly re-project the image points into 3D.

Distance is then:

$$\text{Distance} = \|\mathbf{p}^* - \mathbf{C}_L\|.$$

---

4. Why Use CAHVOR?

A normal "pinhole + radial distortion" model works well for many cameras, but NASA's CAHVOR (and CAHVORE) was designed to handle:

1. **Non-perspective** lenses (fish-eye, wide angle).
2. **Non-trivial** lens distortions.
3. A well-defined set of vectors $\mathbf{C}, \mathbf{A}, \mathbf{H}, \mathbf{V}, \mathbf{O}, \mathbf{R}$ that can easily be passed around in software for large-scale processing (like planetary rover pipelines).

A large part of NASA's stereo pipeline (e.g., for MER, MSL, and other rovers) uses CAHVOR parameters to rectify images or directly triangulate 3D points.

---

5. Putting It All Together

In practice:

1. Each camera is **calibrated** → yields CAHVOR parameters $(\mathbf{C}, \mathbf{A}, \mathbf{H}, \mathbf{V}, \mathbf{O}, \mathbf{R})$.
2. Given two stereo images and a matched feature at (u_L, v_L) & (u_R, v_R):
   1. **Back-project** each pixel → a ray in 3D.
   2. **Intersect** or triangulate the two rays → 3D point.
   3. The 3D point's distance from $\mathbf{C}_L$ (or $\mathbf{C}_R$) is the range to that feature.
3. Optionally, the pipeline might produce a **disparity map** (like a standard stereo approach) **after** rectifying with CAHVOR parameters so that epipolar lines are horizontal. In that scenario, you can do the typical formula $z = \frac{fB}{\text{disparity}}$, but f, B are themselves derived from the CAHVOR parameters (plus the rectification transforms).

Key Takeaway

CAHVOR is basically the camera model that ensures you do the **correct** ray projection from 2D image coordinates to 3D space, accounting for real lens position, orientation, and distortion. Once you have that correct projection for both left and right images, **distance** is found by standard **triangulation** of matching features.

compare our used formula vs theirs because I dont understand why there are 2 approaches

Reasoned for a few seconds

Below is a **side-by-side comparison** of the **two main approaches** to finding distance via stereo images:

1. **Simple Stereo Formula** $\left(\text{Depth} \approx \frac{B \times f}{\text{disparity}}\right)$
2. **CAHVOR Triangulation** (NASA's more general model)

Both approaches ultimately solve the same geometry problem—**inferring 3D location** from two 2D images—just at different levels of **model complexity** and **accuracy**.

---

1) Simple Stereo Formula

$$\boxed{\text{Distance} = \frac{\text{Baseline} \times \text{FocalLengthInPixels}}{\text{Disparity}}}$$

Where

- **Baseline** (B) is the separation between the two cameras (e.g., 0.24 m).
- **FocalLengthInPixels** (f) often comes from $\text{focal (mm)} \div \text{pixel pitch (mm/pixel)}$.
- **Disparity** is Δx, the difference in the **x-coordinate** of the same feature in left vs. right images (assuming images are **rectified** so that epipolar lines are horizontal).

Pros

- **Straightforward**: Easy to implement once your images are rectified.
- **Fast**: If you know your baseline and pixel focal length, you directly invert disparity to get depth.

Cons

- **Requires Rectification**: The images must be aligned so that the only shift is in x. Any unaccounted tilt, rotation, or lens distortion leads to errors.
- **Ignores Distortion** (unless you corrected it beforehand). Even small principal-point shifts or radial distortions cause systematic offsets in the computed distance.

Typical Use Case

- **Consumer Stereo** or **well-rectified** cameras with negligible lens distortion, or after an offline calibration step that produces neatly aligned left/right images.

---

2) CAHVOR Triangulation

Instead of a single formula, **CAHVOR** uses a set of **calibrated vectors**:

- **C**: Center of projection (3D location of camera)
- **A**: Axis vector (which way is "forward" for this camera)
- **H, V**: Horizontal & vertical orientation of the image plane in 3D
- **O, R**: Additional lens distortion parameters

For **each** camera, a 2D pixel coordinate (u, v) is **back-projected** into a 3D ray using those vectors & distortion corrections. When you have matching features (u_L, v_L) and (u_R, v_R):

1. **Compute 3D ray** from left camera: $\mathbf{C}_L + \alpha \mathbf{d}_L$.
2. **Compute 3D ray** from right camera: $\mathbf{C}_R + \beta \mathbf{d}_R$.
3. **Triangulate** the point $\mathbf{p}^*$ that best intersects these two rays in space.
4. **Distance** = $\|\mathbf{p}^* - \mathbf{C}_L\|$ (or from $\mathbf{C}_R$).

Pros

- **More General**: Handles lens distortion, off-center principal points, tilt angles, wide-FOV cameras, etc.
- **No Need for Strict Rectification**: It's a full 3D solution. If the cameras aren't aligned horizontally, CAHVOR still works.
- **Accurate**: If the calibration is done carefully (the vectors truly represent the camera geometry), the method can produce sub-percent to a few percent error.

Cons

- **More Complex**: You need to obtain or compute those CAHVOR vectors for each camera (a calibration pipeline).
- **Slower**: Triangulation and distortion correction are more involved than a single disparity formula.

Typical Use Case

- **NASA rovers** (e.g., MER, MSL, Perseverance) or any system where cameras have noticeable distortion, non-parallel axes, or must be precisely modeled.
- Scenes with significant tilt or offset where standard rectification might be cumbersome.

Why Two Approaches?

They are fundamentally **two different ways** to handle the **same** stereo geometry:

1. **Simple Stereo** is perfect if:
   - Cameras are **already rectified** (or physically parallel with matching optical centers).
   - Distortion is minimal or pre-corrected.
   - You are okay with a simpler calibration (just baseline + focal length in pixels, maybe a small principal point offset).
2. **CAHVOR** is a **full 3D model** that:
   - Accurately describes the camera's position, orientation, and distortion.
   - Does **not require** that the images be strictly horizontally aligned.
   - Is often used in **space missions** or advanced photogrammetry where big or wide-angle distortions and physical camera tilts are standard.

In practice, a well-rectified stereo rig **is** effectively using a simplified camera model that merges the real calibration (like CAHVOR) into a single $\text{pinhole} + \text{baseline}$ scenario. CAHVOR is more general and robust but more involved.

If your rover's cameras are not strictly aligned or have non-negligible lens distortion, or if you want maximum accuracy, the CAHVOR approach will give you **better** distance estimates. If you have a simpler camera pair (or you rectify them offline), the standard formula $\frac{Bf}{\text{disparity}}$ is enough.