In [1]:
```
import os
import sys
import gc
import cv2
import random
import shutil
import numpy as np
from PIL import Image
from random import sample
from pyunpack import Archive
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from scipy.ndimage import map_coordinates
```

In [2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.modules.utils as nn_utils
from torchvision.transforms import PILToTensor
from typing import Any, Callable, Dict, List, Optional, Union, Tuple
from diffusers.models.unet_2d_condition import UNet2DConditionModel
from diffusers import DDIMScheduler
from diffusers import StableDiffusionPipeline
```

```
2024-04-26 14:58:27.737268: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in performa
nce-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operat
ions, rebuild TensorFlow with the appropriate compiler flags.
```

In [3]:
```
#Archive('FLoRI21_DataPort.zip').extractall(os.getcwd())
```

In [4]:
```
#shutil.rmtree(os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Res
```

In [5]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Sta
os.makedirs(path,exist_ok=True)
```

In [6]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Sta
os.makedirs(path,exist_ok=True)
```

In [7]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Fin
os.makedirs(path,exist_ok=True)
```

Note:

Some of the code cells were referenced from the paper titled "Emergent Correspondence from Image Diffusion." Please cite their paper as follows:

```
@inproceedings{tang2023emergent,
  title={Emergent Correspondence from Image Diffusion},
  author={Luming Tang and Menglin Jia and Qianqian Wang and Cheng Perng
Phoo and Bharath Hariharan},
  booktitle={Thirty-seventh Conference on Neural Information Processing
Systems},
  year={2023},
```

```
  url={https://openreview.net/forum?id=ypOiXjdfnU}
}
```

In [8]:

```
class MyUNet2DConditionModel(UNet2DConditionModel):
    """
    Customized 2D U-Net conditioned model inherited from `UNet2DConditionModel`.

    This model extends the original `UNet2DConditionModel` to incorporate additi
    such as encoder hidden states, attention mask, and cross-attention keyword a
    """
    def forward(
        self,
        sample: torch.FloatTensor,
        timestep: Union[torch.Tensor, float, int],
        up_ft_indices,
        encoder_hidden_states: torch.Tensor,
        class_labels: Optional[torch.Tensor] = None,
        timestep_cond: Optional[torch.Tensor] = None,
        attention_mask: Optional[torch.Tensor] = None,
        cross_attention_kwargs: Optional[Dict[str, Any]] = None):
        """
        Forward method for `MyUNet2DConditionModel`.

        Args:
            sample (torch.FloatTensor): Noisy inputs tensor with shape (batch, c
            timestep (torch.FloatTensor or float or int): Timesteps for each bat
            up_ft_indices (list): List of upsampling indices.
            encoder_hidden_states (torch.FloatTensor): Encoder hidden states wit
            class_labels (Optional[torch.Tensor], default=None): Class labels te
            timestep_cond (Optional[torch.Tensor], default=None): Timestep condi
            attention_mask (Optional[torch.Tensor], default=None): Mask to avoid
            cross_attention_kwargs (Optional[dict], default=None): Keyword argum

        Returns:
            dict: Dictionary containing upsampled features (`up_ft`).
        """

        # By default samples have to be AT least a multiple of the overall upsam
        # The overall upsampling factor is equal to 2 ** (# num of upsampling la
        # However, the upsampling interpolation output size can be forced to fit
        # on the fly if necessary.
        default_overall_up_factor = 2**self.num_upsamplers

        # upsample size should be forwarded when sample is not a multiple of `de
        forward_upsample_size = False
        upsample_size = None

        if any(s % default_overall_up_factor != 0 for s in sample.shape[-2:]):
            # logger.info("Forward upsample size to force interpolation output s
            forward_upsample_size = True

        # prepare attention_mask
        if attention_mask is not None:
            attention_mask = (1 - attention_mask.to(sample.dtype)) * -10000.0
            attention_mask = attention_mask.unsqueeze(1)

        # 0. center input if necessary
        if self.config.center_input_sample:
            sample = 2 * sample - 1.0
```

```
        # 1. time
        timesteps = timestep
        if not torch.is_tensor(timesteps):
            # TODO: this requires sync between CPU and GPU. So try to pass times
            # This would be a good case for the `match` statement (Python 3.10+)
            is_mps = sample.device.type == "mps"
            if isinstance(timestep, float):
                dtype = torch.float32 if is_mps else torch.float64
            else:
                dtype = torch.int32 if is_mps else torch.int64
            timesteps = torch.tensor([timesteps], dtype=dtype, device=sample.dev
        elif len(timesteps.shape) == 0:
            timesteps = timesteps[None].to(sample.device)

        # broadcast to batch dimension in a way that's compatible with ONNX/Core
        timesteps = timesteps.expand(sample.shape[0])

        t_emb = self.time_proj(timesteps)

        # timesteps does not contain any weights and will always return f32 tens
        # but time_embedding might actually be running in fp16. so we need to ca
        # there might be better ways to encapsulate this.
        t_emb = t_emb.to(dtype=self.dtype)

        emb = self.time_embedding(t_emb, timestep_cond)

        if self.class_embedding is not None:
            if class_labels is None:
                raise ValueError("class_labels should be provided when num_class

            if self.config.class_embed_type == "timestep":
                class_labels = self.time_proj(class_labels)

            class_emb = self.class_embedding(class_labels).to(dtype=self.dtype)
            emb = emb + class_emb

        # 2. pre-process
        sample = self.conv_in(sample)

        # 3. down
        down_block_res_samples = (sample,)
        for downsample_block in self.down_blocks:
            if hasattr(downsample_block, "has_cross_attention") and downsample_b
                sample, res_samples = downsample_block(
                    hidden_states=sample,
                    temb=emb,
                    encoder_hidden_states=encoder_hidden_states,
                    attention_mask=attention_mask,
                    cross_attention_kwargs=cross_attention_kwargs,
                )
            else:
                sample, res_samples = downsample_block(hidden_states=sample, tem

            down_block_res_samples += res_samples

        # 4. mid
        if self.mid_block is not None:
            sample = self.mid_block(
                sample,
```

```
                emb,
                encoder_hidden_states=encoder_hidden_states,
                attention_mask=attention_mask,
                cross_attention_kwargs=cross_attention_kwargs,
            )

        # 5. up
        up_ft = {}
        for i, upsample_block in enumerate(self.up_blocks):

            if i > np.max(up_ft_indices):
                break

            is_final_block = i == len(self.up_blocks) - 1

            res_samples = down_block_res_samples[-len(upsample_block.resnets) :]
            down_block_res_samples = down_block_res_samples[: -len(upsample_bloc

            # if we have not reached the final block and need to forward the
            # upsample size, we do it here
            if not is_final_block and forward_upsample_size:
                upsample_size = down_block_res_samples[-1].shape[2:]

            if hasattr(upsample_block, "has_cross_attention") and upsample_block
                sample = upsample_block(
                    hidden_states=sample,
                    temb=emb,
                    res_hidden_states_tuple=res_samples,
                    encoder_hidden_states=encoder_hidden_states,
                    cross_attention_kwargs=cross_attention_kwargs,
                    upsample_size=upsample_size,
                    attention_mask=attention_mask,
                )
            else:
                sample = upsample_block(
                    hidden_states=sample, temb=emb, res_hidden_states_tuple=res_
                )

            if i in up_ft_indices:
                up_ft[i] = sample.detach()

        output = {}
        output['up_ft'] = up_ft
        return output

class OneStepSDPipeline(StableDiffusionPipeline):
    """
    One-step Stable Diffusion Pipeline.

    Provides a one-step stable diffusion process, integrating the VAE encoding a
    """
    @torch.no_grad()
    def __call__(
        self,
        img_tensor,
        t,
        up_ft_indices,
        negative_prompt: Optional[Union[str, List[str]]] = None,
        generator: Optional[Union[torch.Generator, List[torch.Generator]]] = Non
        prompt_embeds: Optional[torch.FloatTensor] = None,
```

```
        callback: Optional[Callable[[int, int, torch.FloatTensor], None]] = None
        callback_steps: int = 1,
        cross_attention_kwargs: Optional[Dict[str, Any]] = None
    ):

        """
        Call method for `OneStepSDPipeline`.

        Args:
            img_tensor (torch.Tensor): Image tensor.
            t (torch.Tensor or int): Timesteps tensor.
            up_ft_indices (list): List of upsampling indices.
            negative_prompt (Optional[str or list], default=None): Negative prom
            generator (Optional[torch.Generator or list], default=None): Torch g
            prompt_embeds (Optional[torch.FloatTensor], default=None): Precomput
            callback (Optional[Callable], default=None): Callback function invok
            callback_steps (int, default=1): Frequency of invoking the callback.
            cross_attention_kwargs (Optional[dict], default=None): Keyword argum

        Returns:
            dict: Dictionary containing output from U-Net.
        """
        device = self._execution_device
        latents = self.vae.encode(img_tensor).latent_dist.sample() * self.vae.cc
        t = torch.tensor(t, dtype=torch.long, device=device)
        noise = torch.randn_like(latents).to(device)
        latents_noisy = self.scheduler.add_noise(latents, noise, t)
        unet_output = self.unet(latents_noisy,
                                t,
                                up_ft_indices,
                                encoder_hidden_states=prompt_embeds,
                                cross_attention_kwargs=cross_attention_kwargs)
        return unet_output


class SDFeaturizer:
    """
    Stable Diffusion Featurizer.

    Provides a mechanism to compute stable diffusion based features from an inpu
    """
    def __init__(self, sd_id='stabilityai/stable-diffusion-2-1'):
        """
        Initializes `SDFeaturizer` with a given stable diffusion model ID.

        Args:
            sd_id (str, default='stabilityai/stable-diffusion-2-1'): Stable diff
        """
        unet = MyUNet2DConditionModel.from_pretrained(sd_id, subfolder="unet")
        onestep_pipe = OneStepSDPipeline.from_pretrained(sd_id, unet=unet, safet
        onestep_pipe.vae.decoder = None
        onestep_pipe.scheduler = DDIMScheduler.from_pretrained(sd_id, subfolder=
        gc.collect()
        onestep_pipe = onestep_pipe.to("cuda")
        onestep_pipe.enable_attention_slicing()
        onestep_pipe.enable_xformers_memory_efficient_attention()
        self.pipe = onestep_pipe

    @torch.no_grad()
    def forward(self,
```

```
                img_tensor, # single image, [1,c,h,w]
                t,
                up_ft_index,
                prompt,
                ensemble_size=8):
        """
        Forward method for `SDFeaturizer`.

        Args:
            img_tensor (torch.Tensor): Single input image tensor with shape [1,
            t (torch.Tensor or int): Timesteps tensor.
            up_ft_index (int): Index for upsampling.
            prompt (str): Textual prompt for conditioning.
            ensemble_size (int, default=8): Size of the ensemble for feature ave

        Returns:
            torch.Tensor: Stable diffusion based features with shape [1, c, h, w
        """
        img_tensor = img_tensor.repeat(ensemble_size, 1, 1, 1).cuda() # ensem, c
        prompt_embeds = self.pipe._encode_prompt(
            prompt=prompt,
            device='cuda',
            num_images_per_prompt=1,
            do_classifier_free_guidance=False) # [1, 77, dim]
        prompt_embeds = prompt_embeds.repeat(ensemble_size, 1, 1)
        unet_ft_all = self.pipe(
            img_tensor=img_tensor,
            t=t,
            up_ft_indices=[up_ft_index],
            prompt_embeds=prompt_embeds)
        unet_ft = unet_ft_all['up_ft'][up_ft_index] # ensem, c, h, w
        unet_ft = unet_ft.mean(0, keepdim=True) # 1,c,h,w
        return unet_ft
```

In [9]:

```
class DFT:
    """
    RetinaRegNet (RetinaRegNetwork) utilizes DFT (Diffusion Features) for identi
    and locations between images.
    """
    def __init__(self, imgs,img_size,pts):
        """
        Initialize the DFT object.

        Parameters:
        - imgs (list): List of input image tensors.
        - img_size (int): Expected size of the image for processing.
        - pts (list): List of point tuples specifying coordinates.
        """
        self.pts = pts
        self.imgs = imgs
        self.num_imgs = len(imgs)
        self.img_size = img_size

    def unravel_index(self,index, shape):
        """
        Converts a flat index into a tuple of coordinate indices in a tensor of

        This function mimics numpy's `unravel_index` functionality, which is use
        into a tuple of coordinate indices for an array of given shape. This is
        multi-dimensional indices of a position in a flattened array.
```

```
        Parameters:
        - index (int): The flat index into the array.
        - shape (tuple of ints): The shape of the array from which the index is

        Returns:
        - tuple of ints: A tuple representing the coordinates of the index in an

        Note:
            This function operates under the assumption that indexing starts fro
        """
        out = []
        for dim in reversed(shape):
            out.append(index % dim)
            index = index // dim
        return tuple(reversed(out))

    def compute_pooled_and_combining_feature_maps(self,feature_map, hierarchy_ra
        """
        Compute pooled and stacked feature maps.

        Parameters:
        - feature_map (torch.Tensor): Input feature map.
        - hierarchy_range (int, optional): Depth of hierarchical pooling. Defaul
        - stride (int, optional): Stride for pooling. Defaults to 1.

        Returns:
        - torch.Tensor: Pooled and stacked feature map.
        """
        # List to store the pooled feature maps
        pooled_feature_maps = feature_map
        # Loop through the specified hierarchy range
        for hierarchy in range(1,hierarchy_range):
            # Average pooling with kernel size 3^k x 3^k
            win_size = 3 ** hierarchy
            avg_pool = torch.nn.AvgPool2d(win_size, stride=1, padding=win_size /
            pooled_map = avg_pool(feature_map)
            # Append the pooled feature map to the list
            pooled_feature_maps=+pooled_map
        return pooled_feature_maps

    def compute_batched_2d_correlation_maps(self, pts_list, feature_map1, featur
        """
        Computes 2D correlation maps between selected points in one feature map

        This method takes two feature maps and a list of points. It extracts fea
        at specified points, normalizes them, and then computes a batched 2D cor
        The output is a set of correlation maps, each corresponding to a point i
        feature vector correlates across the spatial dimensions of the second fe

        Parameters:
        - pts_list (list of tuples): List of points (y, x) for which the correla
        - feature_map1 (torch.Tensor): The first feature map tensor of shape (1,
        - feature_map2 (torch.Tensor): The second feature map tensor of shape (1
                                       and H2, W2 do not necessarily need to be eq

        Returns:
        - torch.Tensor: A tensor of shape (NumPoints, H2, W2) where each slice c
                        for each point in `pts_list`.
        Notes:
```

```
        The function assumes that the first dimension of feature_map1 and fe
        This method uses batch matrix multiplication and vector normalizatio
        Running this method on a GPU is recommended due to its computational
    """
    # Convert the input tensors to float16
    feature_map1 = feature_map1.to(dtype=torch.float16)
    feature_map2 = feature_map2.to(dtype=torch.float16)
    _, C, H, W = feature_map2.shape

    # Flatten feature_map2 for batch matrix multiplication
    feature_map2_flat = feature_map2.view(C, H*W)

    # Prepare a batch of point features
    points_indices = torch.tensor(pts_list)
    point_features = feature_map1[0, :, points_indices[:, 0], points_indices

    # Normalize the point features and feature_map2_flat
    point_features_norm = torch.norm(point_features, dim=1, keepdim=True)
    normalized_point_features = point_features / point_features_norm

    feature_map2_norm = torch.norm(feature_map2_flat, dim=0, keepdim=True)
    normalized_feature_map2 = feature_map2_flat / feature_map2_norm

    # Compute the correlation map for each point
    correlation_maps = torch.mm(normalized_point_features, normalized_featur

    # Reshape the correlation maps to the desired output shape (NumPoints, H
    correlation_maps = correlation_maps.view(-1, H, W)

    # Cleanup if needed
    torch.cuda.empty_cache()

    return correlation_maps

def compute_correlation_map_max_locations(self, pts_list, feature_map1, feat
    """
    Compute the maximum locations in the batched correlation maps between tw

    Parameters:
    - pts_list (list of tuples): List of points for which the correlation ma
    - feature_map1, feature_map2 (torch.Tensor): The input feature maps.

    Returns:
    - torch.Tensor: Tensor of maximum locations for each point.
    - torch.Tensor: Tensor of maximum values for each point.
    """
    enhanced_feature_map1 = self.compute_pooled_and_combining_feature_maps(f
    enhanced_feature_map2 = self.compute_pooled_and_combining_feature_maps(f
    # Compute the batched correlation maps
    batched_correlation_maps = self.compute_batched_2d_correlation_maps(pts_

    M,H2, W2 = batched_correlation_maps.shape
    #print(batched_correlation_maps.shape)

    # Find the maximum values and their locations along the last two dimensi
    max_values, max_indices_flat = torch.max(batched_correlation_maps.view(1

    x, y = zip(*[self.unravel_index(idx.item(), (H2, W2)) for idx in max_ind
    x = torch.tensor(x, device = 'cuda').view(M)
    y = torch.tensor(y, device = 'cuda').view(M)
```

```
        # Stack the coordinates to get a 2xHxW tensor
        max_locations = torch.stack((x, y)).t()

        return max_locations, max_values

    def feature_upsampling(self,ft):
        """
        Upsample the feature to match the specified image size.

        Parameters:
        - ft (torch.Tensor): Feature tensor to be upsampled.

        Returns:
        - tuple: Upsampled source and target feature maps.
        """
        with torch.no_grad():
            num_channel = ft.size(1)
            src_ft = ft[0].unsqueeze(0)
            src_ft = nn.Upsample(size=(self.img_size, self.img_size), mode='bili
            gc.collect()
            torch.cuda.empty_cache()
            trg_ft = nn.Upsample(size=(self.img_size, self.img_size), mode='bili
        return src_ft,trg_ft

    def feature_maps(self,feature_map1,feature_map2,iccl):
        """
        Processes feature maps to extract points that meet the inverse consisten

        This method computes the maximum locations of correlation between featur
        checks for inverse consistency between the mapped points. It filters the
        specified inverse consistency criteria limit (iccl), keeping only those
        distance between the original point and its double-mapped location is wi

        Parameters:
        - feature_map1 (torch.Tensor): The first feature map, used as the base f
        - feature_map2 (torch.Tensor): The second feature map, used for reverse
        - iccl (float): The maximum allowed distance (inverse consistency criter
                        its double-mapped location to be considered consistent.

        Returns:
        tuple of (list, list, list):
        - pnts (list of tuples): The points from the original feature map that m
        - rmaxs (list of floats): The maximum correlation values at these points
        - rspts (list of tuples): The corresponding points in the second feature
                                  with the points in `pnts`.
        """
        pnts,rmaxs,rspts=[],[],[]
        pts = [(int(y), int(x)) for x, y in self.pts]
        max_indices_ST, max_values_ST = self.compute_correlation_map_max_locatio
        x_prime_y_prime = max_indices_ST
        max_indices_TS, max_values_TS = self.compute_correlation_map_max_locatio
        x_prime_prime_y_prime_prime = max_indices_TS
        for i, (pt, max_idx) in enumerate(zip(self.pts, x_prime_prime_y_prime_pr
            # Calculate the distance between the point and the max correlation i
            if np.sqrt((pt[1] - max_idx.cpu()[0]) ** 2 + (pt[0] - max_idx.cpu()[
                pnts.append((int(pt[0]), int(pt[1])))
                rmaxs.append(max_values_ST[i].cpu().item())  # Assuming max_valu
                rspts.append((x_prime_y_prime[i][1].cpu().item(), x_prime_y_prim
        return pnts, rmaxs, rspts
```

In [10]:
```python
def compute_boundary(image, mean_intensity):
    """
    Compute the boundary of an image based on its mean intensity.

    Parameters:
    - image (numpy.array): The input grayscale image.
    - mean_intensity (float): Average intensity of the image to define boundarie

    Returns:
    - tuple: upper, lower, left, and right boundaries of the image region with i
    """
    # Compute the upper, lower, left, and right boundary
    upper_boundary = next((i for i, row in enumerate(image) if np.mean(row) > me
    lower_boundary = next((i for i, row in enumerate(image[::-1]) if np.mean(row

    left_boundary = next((i for i, col in enumerate(image.T) if np.mean(col) > m
    right_boundary = next((i for i, col in enumerate(image.T[::-1]) if np.mean(c

    return upper_boundary, image.shape[0]-lower_boundary, left_boundary, image.s

def is_within_boundary(kp, boundaries):
    """
    Check if a keypoint is within the specified boundaries.

    Parameters:
    - kp (cv2.KeyPoint): The keypoint to check.
    - boundaries (tuple): Tuple of (upper, lower, left, right) boundaries.

    Returns:
    - bool: True if the keypoint is within the boundaries, False otherwise.
    """
    upper, lower, left, right = boundaries
    return left <= kp.pt[0] <= right and upper <= kp.pt[1] <= lower

def SIFT_top_n_keypoints(image_path, N=250, img_shape=256, max_dist=25):
    """
    Detect top N keypoints in the given image using SIFT, considering constraint

    Parameters:
    - image_path (str): Path to the input image.
    - N (int): Number of keypoints to select. Defaults to 250.
    - img_shape (int): The size to which the image should be resized. Defaults t
    - max_dist (int): Minimum distance between selected keypoints. Defaults to 2

    Returns:
    - list: List of keypoints' positions in the form (x, y).
    """
    # Load image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image = cv2.resize(image, (img_shape, img_shape))

    # Initialize SIFT detector
    sift = cv2.SIFT_create()

    # Detect keypoints and compute descriptors
    keypoints, descriptors = sift.detectAndCompute(image, None)

    # Sort keypoints based on response (strength of the keypoint)
    keypoints = sorted(keypoints, key=lambda x: -x.response)
```

```
    # Determine the intensity threshold
    mean_intensity = np.mean(image)
    boundaries = compute_boundary(image, mean_intensity)

    # Select top N keypoints
    selected_keypoints = []
    for keypoint in keypoints:
        # Check if the keypoint is within the boundary
        if is_within_boundary(keypoint, boundaries):
            # Check if the pixel intensity at the keypoint is greater than the t
            if image[int(keypoint.pt[1]), int(keypoint.pt[0])] > mean_intensity:
                # Check if the keypoint is far from existing selected keypoints
                if all(cv2.norm(np.array(keypoint.pt) - np.array(kp.pt)) > max_d
                    selected_keypoints.append(keypoint)

            # Break if N keypoints are selected
            if len(selected_keypoints) == N:
                break

    return [kp.pt for kp in selected_keypoints]

def select_random_points(img, num_points=100, img_size=1200,offset=0.01,window_s
    """
    Selects a specified number of random points from an image, ensuring that eac
    meeting a defined intensity threshold within the image. The image is resized
    are chosen randomly, with each potential point undergoing validation against

    Parameters:
    - img (str): Path to the image file.
    - num_points (int, optional): The number of random points to select. Default
    - img_size (int, optional): The size to which the image is resized (assumed
    - offset (float, optional): Proportional offset to exclude points near the e
                                the image dimensions. Defaults to 0.01.
    - window_size (int, optional): Size of the square window used to check pixel
                                   Defaults to 51.
    - max_attempts_per_point (int, optional): The maximum number of attempts all
                                              that meets the criteria. Defaults to

    Returns:
    - list of tuples: A list where each tuple represents the (y, x) coordinates

    Notes:
        The function converts the image to grayscale and resizes it to img_size
        points near the image boundary by applying a boundary offset calculated
        Each point must be centered in a window (defined by 'window_size') where
        greater than or equal to 5. If the function fails to find a suitable poi
        for any location, it stops and returns the points found up to that momen
    """

    image = cv2.resize(cv2.imread(img, cv2.IMREAD_GRAYSCALE), (img_size, img_siz
    h, w = image.shape
    boundary_offset = int(offset * h)
    pts = []
    window_offset = window_size // 2  # Calculate the offset from the center of

    while len(pts) < num_points:
        attempts = 0
        while attempts < max_attempts_per_point:
            x = random.randint(boundary_offset + window_offset, h - boundary_off
```

```
            y = random.randint(boundary_offset + window_offset, w - boundary_off

            # Define the window boundaries
            x_lower = x - window_offset
            x_upper = x + window_offset + 1
            y_lower = y - window_offset
            y_upper = y + window_offset + 1

            # Check that no pixel in the window has an intensity less than 10
            if np.all(image[x_lower:x_upper, y_lower:y_upper] >= 5):
                pts.append((y, x))
                break  # Successfully found a point, break the inner loop
            attempts += 1  # Increment attempts

        if attempts == max_attempts_per_point:
            print("Maximum attempts reached, unable to find sufficient points wi
            break  # Break outer loop if max attempts is reached without finding

    return pts
```

In [11]:

```
def CLAHE_plot_cond(image,disp_clip):
    """
    Conditionally applies CLAHE to an image based on the provided clipping limit

    Parameters:
        image (np.array): The input image as a NumPy array, typically grayscale.
        disp_clip (float or str): The clip limit for CLAHE. If it is '0.0' (as a

    Returns:
        np.array: The image after applying CLAHE if `disp_clip` is not '0.0'; ot
    """
    if float(disp_clip)!=0.0:
        image = clahe(image,float(disp_clip))
    return image

def outliers_plot_condition(landmark_errors,cond):
    """
    Filters out specific outlier values from a list of landmark errors based on

    This function examines each error in the list of landmark errors and removes
    in this case, the value 10000, if the condition specified by the 'cond' para
    is False, the list is returned unchanged. This functionality can be useful f
    data before further analysis or visualization.

    Parameters:
    - landmark_errors (list of int or float): A list containing numerical values
                                              in landmarks detection.
    - cond (bool): A condition that determines whether the filtering of outliers
                 True, outliers are removed; if False, the list is returned as i

    Returns:
    - list of int or float: A list of landmark errors with specified outliers re
    """
    if cond:
        landmark_errors =[x for x in landmark_errors if x!=10000]
    return landmark_errors

def clahe(imag, clip):
    """
    Apply Contrast Limited Adaptive Histogram Equalization (CLAHE) to an image.
```

```
    This function converts an image to grayscale, applies CLAHE to enhance the i
    and then converts it back to RGB. It uses OpenCV for the CLAHE operation and
    conversions.

    Parameters:
    - imag (np.array): The input image array. Expected to be in format suitable
    - clip (float): The clipping limit for the CLAHE algorithm, which controls t
                  Higher values increase contrast.

    Returns:
    - np.array: The contrast-enhanced image in RGB format.

    Notes:
          The tile grid size for CLAHE is set to (8, 8). Adjustments to this par
          the granularity of the histogram equalization.
    """
    clahe = cv2.createCLAHE(clipLimit=clip, tileGridSize=(8, 8))
    imag = Image.fromarray(np.uint8(imag))
    imag = imag.convert('L')
    img = np.asarray(imag)
    image_equalized = clahe.apply(img)
    image_equalized_img = Image.fromarray(np.uint8(image_equalized))
    image_equalized = image_equalized_img.convert('RGB')
    image_equalized = np.asarray(image_equalized)
    return image_equalized

def compute_plot_Flori21_AUC(landmark_errors):
    """
    Function to compute and plot the success rate curve and calculate the AUC fo

    Parameters:
    - landmark_errors: List of landmark errors including outliers.
    """
    landmark_errors_sorted = sorted(landmark_errors) # includes all outliers as
    # Initialize lists for thresholds and success rates
    thresholds = list(range(101)) # 0 to 100
    success_rates = []
    # Calculate success rate for each threshold
    for threshold in thresholds:
        successful_count = sum([1 for error in landmark_errors_sorted if error <
        success_rate = successful_count / len(landmark_errors_sorted)
        success_rates.append(success_rate * 100) # convert to percentage
    # Plot the curve
    plt.plot(thresholds, success_rates, label="Success Rate Curve")
    plt.xlabel("Threshold")
    plt.ylabel("Success Rate (%)")
    plt.title("Success Rate vs. Threshold")
    plt.legend()
    plt.grid(True)
    plt.show()
    # Compute AUC
    auc = np.sum(success_rates) / 10000 # normalize to 0-1
    print("AUC:", auc)

def plot_landmark_errors(landmark_errors,rpth,chr='All',disable_outliers=False):
    """
    Plots a graph of landmark errors over successive iterations to provide a vis
    across samples. This function is designed to help in the assessment of regis
    or computer vision tasks by plotting each landmark error against its iterati
```

```
    displays the average landmark error across all iterations.

    Parameters:
    - landmark_errors (list of float): A list containing numerical errors for ea
                                    Outliers (e.g., errors set to 10000) are au
    - rpth (str): Path where the resulting plot image will be saved.
    - chr (str, optional): Characteristic or description to include in the plot
                         used. Defaults to 'All'.
    - disable_outliers (bool, optional): If set to True, disables the automatic
                                      data. Defaults to False.

    Returns:
    - None: This function does not return any value but saves the plot to the sp

    Notes:
        This plot is useful for tracking improvements or deteriorations in landm
        It automatically filters out error values set to 10000, considering them
        is set to True.
        The function saves the plot in the directory specified by `rpth` and nam
    """
    landmark_errors=outliers_plot_condition(landmark_errors,disable_outliers)
    samples = list(range(0, len(landmark_errors)))
    avg_error = sum(landmark_errors) / len(landmark_errors)
    plt.figure(figsize=(12, 7))
    plt.plot(samples, landmark_errors, marker='o', linestyle='-', color='#2C3E50
    plt.axhline(y=avg_error, color='#E74C3C', linestyle='--', label=f"Average Er
    plt.title("Mean Landmark Error for the entire Database Housing {} images".fo
    plt.xlabel("Iteration Number", fontsize=14)
    plt.ylabel("Landmark Error", fontsize=14)
    plt.xticks(samples, [f"Case {i}" for i in samples], rotation=45)
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.legend(fontsize=12)
    plt.tight_layout()
    plt.savefig(os.path.join(rpth,'Landmark_Error_Plot.png'))
    plt.show();

def image_point_correspondences(images,img_size,landmarks1,landmarks2,rpth,num,s
    """
    Displays and compares point correspondences between two images using given l

    This function visualizes two images side-by-side with their respective landm
    landmarks across the images is marked with the same color for easy identific
    is designed to handle visualization for studies involving image registration
    matching is crucial.

    Parameters:
    - images (list of str): File paths to the two images (source and target imag
    - img_size (int): The size to which images should be resized, specified as w
    - landmarks1 (list of tuples): Landmark points on the first image (source im
    - landmarks2 (list of tuples): Corresponding landmark points on the second i
    - rpth (str): Path where the resultant visualization should be saved.
    - num (int): An identifier number used to differentiate the output file name
    - snum (str): Stage number or identifier to categorize the process stage.
    - disp_size (int, optional): The display size to which the image will be res
    - disp_clip (float, optional): Enabling the image to be enhanced using CLAHE

    Returns:
    - None: This function directly displays the image using matplotlib and saves

    Notes:
```

```
        The function uses OpenCV for reading and resizing images. It employs a C
        Matplotlib is used for visualizing the images and landmarks. The color m
        of landmarks; if there are more than 15 landmarks, a cyclic colormap is
        This function is particularly useful for visualizing transformations and
        similar fields where point correspondence is critical.
    """
    image1 = CLAHE_plot_cond(cv2.cvtColor(cv2.resize(cv2.imread(images[0]),(disp
    image2 = CLAHE_plot_cond(cv2.cvtColor(cv2.resize(cv2.imread(images[1]),(disp
    landmarks1 = coordinates_rescaling(landmarks1,img_size,img_size,disp_size)
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,disp_size)
    assert len(landmarks1) == len(landmarks2), f"points lengths are incompatible
    num_points = len(landmarks1)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
    fig.suptitle("Stage-{} Point Correspondences".format(snum), fontsize=14, fon
    ax1.set_title('Fixed Image')
    ax2.set_title('Moving Image')
    ax1.axis('off')
    ax2.axis('off')
    ax1.imshow(image1)
    ax2.imshow(image2)
    if num_points > 15:
        cmap = plt.get_cmap('tab20')
    else:
        cmap = ListedColormap(["red", "yellow", "blue", "lime", "magenta", "indi
                               "maroon", "black", "white", "chocolate", "gray",
    colors = np.array([cmap(x) for x in range(len(landmarks1))])
    radius1, radius2 = 4, 1
    for point1, point2, color in zip(landmarks1, landmarks2, colors):
        x1, y1 = point1
        circ1_1 = plt.Circle((x1, y1), radius1, facecolor=color, edgecolor='whit
        circ1_2 = plt.Circle((x1, y1), radius2, facecolor=color, edgecolor='whit
        ax1.add_patch(circ1_1)
        ax1.add_patch(circ1_2)
        x2, y2 = point2
        circ2_1 = plt.Circle((x2, y2), radius1, facecolor=color, edgecolor='whit
        circ2_2 = plt.Circle((x2, y2), radius2, facecolor=color, edgecolor='whit
        ax2.add_patch(circ2_1)
        ax2.add_patch(circ2_2)
    plt.figtext(0.5, 0.115, "Note: {0} point correspondences were identified by
    plt.savefig(os.path.join(rpth,'Stage'+str(snum)+'Point_Correspondences'+str(
    plt.show();

def original_image_point_correspondences(images,orig_moving_image_pth,img_size,
    """
    Visualizes and saves point correspondences across three images (fixed, movin
    to aid in assessing the effectiveness of image registration processes. The f
    contrast using CLAHE for enhanced visualization and overlays landmark points

    Parameters:
    - images (list of np.array): List containing three images representing fixed
    - orig_moving_image_pth (str): Path to the original moving image, used to up
    - img_size (tuple): Original dimensions (width, height) of the images prior
    - landmarks1 (list of tuples): Coordinates of landmarks in the fixed image.
    - landmarks2 (list of tuples): Coordinates of landmarks in the original movi
    - landmarks3 (list of tuples): Coordinates of landmarks in the transformed i
    - rpth (str): Directory path where the result images will be saved.
    - num (int): Identifier to differentiate output filenames.
    - disp_size (int, optional): Target size (one dimension) for scaling images
    - disp_clip (float, optional): Enabling the image to be enhanced using CLAHE
```

```
    Raises:
    - AssertionError: If the number of landmarks in any list does not match the

    Notes:
        The images are resized to `disp_size` for display.
        Landmarks are also rescaled to match the display size.
        A colormap is applied to distinguish between different landmarks; a larg
    """
    assert len(landmarks1) == len(landmarks2) == len(landmarks3), "All landmarks
    images[1]=cv2.imread(orig_moving_image_pth) # replacing the deformed image w
    num_points = len(landmarks1)
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Final Registration Results by Composing Transformations Estima

    ax1.set_title('Fixed Image')
    ax2.set_title('Moving Image')
    ax3.set_title('Deformed Image')

    ax1.axis('off')
    ax2.axis('off')
    ax3.axis('off')

    ax1.imshow(CLAHE_plot_cond(cv2.cvtColor(cv2.resize(images[0],(disp_size,disp
    ax2.imshow(CLAHE_plot_cond(cv2.cvtColor(cv2.resize(images[1],(disp_size,disp
    ax3.imshow(CLAHE_plot_cond(cv2.cvtColor(cv2.resize(images[2].astype(np.uint8

    landmarks1 = coordinates_rescaling(landmarks1,img_size,img_size,disp_size)
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,disp_size)
    landmarks3 = coordinates_rescaling(landmarks3,img_size,img_size,disp_size)

    if num_points > 15:
        cmap = plt.get_cmap('tab20')
    else:
        cmap = ListedColormap(["red", "yellow", "blue", "lime", "magenta", "indi
                               "maroon", "black", "white", "chocolate", "gray",

    colors = np.array([cmap(x) for x in range(num_points)])
    radius1, radius2 = 4, 1

    for point1, point2, point3, color in zip(landmarks1, landmarks2, landmarks3,
        # Landmarks for Image 1
        x1, y1 = point1
        circ1_1 = plt.Circle((x1, y1), radius1, facecolor=color, edgecolor='whit
        circ1_2 = plt.Circle((x1, y1), radius2, facecolor=color, edgecolor='whit
        ax1.add_patch(circ1_1)
        ax1.add_patch(circ1_2)

        # Landmarks for Image 2
        x2, y2 = point2
        circ2_1 = plt.Circle((x2, y2), radius1, facecolor=color, edgecolor='whit
        circ2_2 = plt.Circle((x2, y2), radius2, facecolor=color, edgecolor='whit
        ax2.add_patch(circ2_1)
        ax2.add_patch(circ2_2)

        # Landmarks for Image 3
        x3, y3 = point3
        circ3_1 = plt.Circle((x3, y3), radius1, facecolor=color, edgecolor='whit
        circ3_2 = plt.Circle((x3, y3), radius2, facecolor=color, edgecolor='whit
        ax3.add_patch(circ3_1)
        ax3.add_patch(circ3_2)
```

```
    plt.savefig(os.path.join(rpth, 'Final_Registration_Results_for_case' + str(r
    plt.show();
```

In [12]:
```
def transform_points_affine(moving_points, affine_matrix):
    """
    Transform the moving points using the given affine matrix.

    Parameters:
    - moving_points: List of (x, y) tuples
    - affine_matrix: (3x3) affine matrix

    Returns:
    - List of (x, y) tuples representing transformed points
    """
    points_array = np.array(moving_points)
    homogeneous_points = np.hstack([points_array, np.ones((len(moving_points), 1
    transformed_points = np.dot(homogeneous_points, affine_matrix.T)
    return [tuple(point) for point in transformed_points[:, :2]]

def transform_points_homography(moving_points, homography_matrix):
    """
    Transform the moving points using the given homography matrix.

    Parameters:
    - moving_points: List of (x, y) tuples
    - homography_matrix: (3x3) homography matrix

    Returns:
    - List of (x, y) tuples representing transformed points
    """
    points_array = np.array(moving_points)
    homogeneous_points = np.hstack([points_array, np.ones((len(moving_points), 1
    transformed_points = np.dot(homogeneous_points, homography_matrix.T)
    transformed_points /= transformed_points[:, 2][:, np.newaxis]  # Normalize b
    return [tuple(point[:2]) for point in transformed_points]

def transform_points_third_order_polynomial(moving_points, coefficients):
    """
    Transform the moving points using the given third-order polynomial coefficie

    Parameters:
    - moving_points: List of (x, y) tuples
    - coefficients: Array of 20 coefficients for the third-order polynomial tran

    Returns:
    - List of (x, y) tuples representing transformed points
    """
    if len(coefficients) != 20:
        raise ValueError("Coefficients should have a shape of (20,).")

    # Extract the coefficients
    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, \
    a11, a12, a13, a14, a15, a16, a17, a18, a19, a20 = coefficients

    transformed_points = []
    for x, y in moving_points:
        # Compute new x' and y' for each point using third-order polynomial
        x_prime = (a1*x**3 + a2*x**2*y + a3*x*y**2 + a4*y**3 +
                   a5*x**2 + a6*x*y + a7*y**2 + a8*x + a9*y + a10)
```

```
        y_prime = (a11*x**3 + a12*x**2*y + a13*x*y**2 + a14*y**3 +
                   a15*x**2 + a16*x*y + a17*y**2 + a18*x + a19*y + a20)
        transformed_points.append((x_prime, y_prime))

    return transformed_points

def transform_points_quadratic(points, coefficients):
    """
    Applies a quadratic transformation to a set of 2D points based on the provid
    typically used in image processing and computer vision tasks to deform point

    Parameters:
    - points (list of tuples): A list of points, where each point is represented
    - coefficients (list): A list of 12 coefficients for the quadratic transform

    Returns:
    - list: A list of tuples representing the deformed points.

    Raises:
    - ValueError: If the number of coefficients is not equal to 12, as the quadr

    Notes:
        This function uses a quadratic transformation defined as:
        x' = a1*x + a2*y + a3*x*y + a4*x^2 + a5*y^2 + a6
        y' = a7*x + a8*y + a9*x*y + a10*x^2 + a11*y^2 + a12
        where `x, y` are the original coordinates and `x', y'` are the transform
        The coefficients must be specified in the order [a1, a2, a3, a4, a5, a6,
    """
    if len(coefficients) != 12:
        raise ValueError("Coefficients should have a shape of (12,).")

    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12 = coefficients

    deformed = []
    for x, y in points:
        x_prime = a1*x + a2*y + a3*x*y + a4*x**2 + a5*y**2 + a6
        y_prime = a7*x + a8*y + a9*x*y + a10*x**2 + a11*y**2 + a12
        deformed.append((x_prime, y_prime))

    return deformed

def compute_landmark_error_fixed_space(polynomial_matrix,fixed_points,moving_poi
    """
    Compute the landmark error between fixed points and transformed moving point

    Parameters:
    - fixed_points: List of (x, y) tuples in the fixed image.
    - moving_points: List of (x, y) tuples in the moving image.
    - polynomial_matrix: (3x3) matrix used to transform points using a third-ord
    - image_size: The original size of the images.
    - new_image_size: The size of the images after rescaling.

    Returns:
    - mle: Mean Landmark Error.
    """
    transformed_points = transform_points_third_order_polynomial(moving_points,
    transformed_points = coordinates_rescaling_high_scale(transformed_points,new
    errors = np.linalg.norm(np.array(fixed_points) - transformed_points, axis=1)
    mle = np.mean(errors)
    return mle
```

```
def compute_landmark_error(fixed_points,fixed_image_size,moving_points,moving_im
    """
    Calculates the mean landmark error between fixed points and transformed movi
    after rescaling to a new image size. This function is primarily used in imag
    to measure the accuracy of image registration by quantifying the displacemen

    Parameters:
    - fixed_points (list of tuples): Coordinates of landmark points in the fixed
    - fixed_image_size (tuple): The original size (width, height) of the fixed i
    - moving_points (list of tuples): Coordinates of landmark points in the movi
    - moving_image_size (tuple): The original size (width, height) of the moving
    - new_image_size (int): The size to which both sets of points will be resize

    Returns:
    - float: The mean landmark error calculated as the average Euclidean distanc
          landmarks after rescaling to the new image size.

    Notes:
        The function first rescales the coordinates of both fixed and moving poi
        It then calculates the Euclidean distance between the corresponding resc
        This metric is useful for evaluating the precision of image registration
    """
    rescaled_fixed_points = coordinates_rescaling_high_scale(fixed_points,new_im
    rescaled_moving_points = coordinates_rescaling_high_scale(moving_points,new_
    errors = np.linalg.norm(np.array(rescaled_fixed_points) - rescaled_moving_po
    mle = np.mean(errors)
    return mle

def compute_third_order_polynomial_matrix(landmarks1, landmarks2):
    """
    Compute coefficients for the third-order polynomial transformation.

    Parameters:
    - landmarks1 (list): List of (x, y) tuples of landmarks in the first image.
    - landmarks2 (list): List of (x, y) tuples of landmarks in the second image.

    Returns:
    - np.array: Coefficients of the third-order polynomial transformation.
    """
    if len(landmarks1) != len(landmarks2) or len(landmarks1) < 10:
        raise ValueError("Both landmarks should have the same number of points,

    A = []
    B = []

    for (x, y), (x_prime, y_prime) in zip(landmarks1, landmarks2):
        # For x'
        A.append([x**3, x**2 * y, x * y**2, y**3, x**2, x * y, y**2, x, y, 1, 0,
        # For y'
        A.append([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, x**3, x**2 * y, x * y**2, y**3,

        B.extend([x_prime, y_prime])

    A = np.array(A)
    B = np.array(B)

    # Solve the linear system
    coefficients, _, _, _ = np.linalg.lstsq(A, B, rcond=None)
```

```
    return coefficients  # The shape of coefficients is (20,)

def compute_quadratic_matrix(landmarks1, landmarks2):
    """
    Compute the quadratic matrix using provided landmarks.

    Parameters:
    - landmarks1: List of (x, y) tuples from the source image.
    - landmarks2: List of (x, y) tuples from the target image.

    Returns:
    - 3x3 homography matrix.
    """
    if len(landmarks1) != len(landmarks2) or len(landmarks1) < 6:
        raise ValueError("Both landmarks should have the same number of points,

    A = []
    B = []

    for (x, y), (x_prime, y_prime) in zip(landmarks1, landmarks2):
        A.append([x, y, x*y, x*x, y*y, 1, 0, 0, 0, 0, 0, 0])
        A.append([0, 0, 0, 0, 0, 0, x, y, x*y, x*x, y*y, 1])

        B.append(x_prime)
        B.append(y_prime)

    A = np.array(A)
    B = np.array(B)

    # Solve the linear system
    coefficients, _, _, _ = np.linalg.lstsq(A, B, rcond=None)

    return coefficients

def compute_homography_matrix(landmarks1, landmarks2):
    """
    Compute the homography matrix using provided landmarks.

    Parameters:
    - landmarks1: List of (x, y) tuples from the source image.
    - landmarks2: List of (x, y) tuples from the target image.

    Returns:
    - 3x3 homography matrix.
    """
    homography_matrix, _ = cv2.findHomography(np.array(landmarks1), np.array(lar
    return homography_matrix

def transform_points_third_order_polynomial_matrix(landmarks1, landmarks2,img_si
    """
    Computes a third-order polynomial transformation matrix based on rescaled la
    to another. This transformation is typically used for tasks like geometric t
    alignment or registration of image features is necessary.

    Parameters:
    - landmarks1 (list of tuples): List of original landmark points in the sourc
    - landmarks2 (list of tuples): List of corresponding landmark points in the
                                   The points in landmarks2 should correspond one-
    - img_size (int): Original size of the images from which the landmarks were
                      rescale points for accurate computation of the transformatio
```

```
    - new_img_size (int): New size to which the points will be rescaled before c
                          This should reflect the size of the image space into whi

    Returns:
    - numpy.ndarray: A transformation matrix which can be used to map the points
                     to the space defined by landmarks2. The matrix is represe
                     corresponding to the terms of a third-order polynomial.

    Notes:
        Ensure that the number of points in landmarks1 and landmarks2 are equal
        This function involves rescaling coordinates, calculating a transformati
        tasks where geometric transformations are necessary for alignment and re
    """
    landmarks1 = coordinates_rescaling(landmarks1,img_size,img_size,new_img_size
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,new_img_size
    third_order_polynomial_matrix  = compute_third_order_polynomial_matrix(landm
    return third_order_polynomial_matrix

def transform_points_quadratic_matrix(landmarks1, landmarks2,img_size,new_img_si
    """
    Computes a quadratic transformation matrix based on rescaled landmarks from

    This function rescales the input landmarks from their original dimensions (i
    It then calculates a quadratic transformation matrix that describes how poin
    can be transformed to align with the second set (landmarks2). This matrix co
    to images or coordinates.

    Parameters:
    - landmarks1 (list of tuples): List of (x, y) tuples representing original l
    - landmarks2 (list of tuples): List of (x, y) tuples representing target lan
                                   corresponding to landmarks1.
    - img_size (int): The original size (width and height, assumed square) of th
    - new_img_size (int): The new size (width and height, assumed square) to whi
                          before computing the transformation matrix.

    Returns:
    - numpy.ndarray: A matrix that contains the coefficients of the quadratic tr
                     to transform points from the source image to the target i

    Notes:
        Ensure that the number of points in landmarks1 and landmarks2 are equal
        This function is essential in image processing tasks where precise trans
    """
    landmarks1 = coordinates_rescaling(landmarks1,img_size,img_size,new_img_size
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,new_img_size
    quadratic_matrix = compute_quadratic_matrix(landmarks1, landmarks2)
    return quadratic_matrix


def warp_image_third_order_polynomial(image, coefficients):
    """
    Applies a third-order polynomial transformation to an image using provided c

    Parameters:
    - image (numpy.ndarray): The image to deform, provided as a numpy array. The
                             two-dimensional (grayscale image) or three-dimensiona
    - coefficients (list or array): An array of 20 coefficients for the third-or

    Raises:
    - ValueError: If the number of coefficients provided is not 20, an error is
```

```
                of exactly 20 coefficients to perform the transformation.

    Returns:
    - numpy.ndarray: The deformed image as a numpy array of the same shape as th

    Notes:
        The deformation is defined by a polynomial transformation that adjusts t
        based on the polynomial defined by the coefficients.
        This function supports both grayscale and color images. For color images
        to each color channel independently.
        The transformation involves calculating new pixel positions and mapping
        to these new positions using spline interpolation of order 1.
    """
    if len(coefficients) != 20:
        raise ValueError("Coefficients should have a shape of (20,).")

    # Extract the coefficients
    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, \
    a11, a12, a13, a14, a15, a16, a17, a18, a19, a20 = coefficients

    # Check if the image is grayscale or colored
    if len(image.shape) == 2:
        height, width = image.shape
        output = np.zeros((height, width))
        channels = 1
        image = image[:, :, np.newaxis]  # add an additional dimension for consi
    else:
        height, width, channels = image.shape
        output = np.zeros((height, width, channels))

    # Generate the coordinates
    coordinates = np.indices((height, width))
    x_coords = coordinates[1]
    y_coords = coordinates[0]

    # Compute new x' and y' for every x and y using third-order polynomial
    x_prime = (a1*x_coords**3 + a2*x_coords**2*y_coords + a3*x_coords*y_coords**
               a5*x_coords**2 + a6*x_coords*y_coords + a7*y_coords**2 + a8*x_coo
    y_prime = (a11*x_coords**3 + a12*x_coords**2*y_coords + a13*x_coords*y_coord
               a15*x_coords**2 + a16*x_coords*y_coords + a17*y_coords**2 + a18*x

    # Map the old image pixels to the new deformed positions
    for c in range(channels):  # for each channel
        output[:, :, c] = map_coordinates(image[:, :, c], [y_prime, x_prime], or

    if channels == 1:
        return output[:, :, 0]  # return as 2D grayscale image
    else:
        return output

def warp_image_quadratic_matrix(image, coefficients):
    """
    Applies a quadratic transformation to deform an image using provided coeffic

    Parameters:
    - image (numpy.ndarray): The image to deform, represented as a numpy array.
                             either two-dimensional (grayscale image) or three-dim
    - coefficients (list or array): A list or array of 12 coefficients defining

    Raises:
```

```
    - ValueError: If the number of coefficients provided is not equal to 12, rai
                that exactly 12 coefficients are required for the transformation

    Returns:
    - numpy.ndarray: The deformed image as a numpy array of the same shape as th

    Notes:
        The deformation involves calculating new pixel coordinates using the qua
        by the coefficients and then mapping the original pixel values to these
        The function checks if the image is in grayscale or color and processes
        The mapping of pixels uses spline interpolation of order 1 for accuracy
        the transformed coordinates with zeros.
    """
    if len(coefficients) != 12:
        raise ValueError("Coefficients should have a shape of (12,).")

    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12 = coefficients

    # Check if the image is grayscale or colored
    if len(image.shape) == 2:
        height, width = image.shape
        output = np.zeros((height, width))
        channels = 1
        image = image[:, :, np.newaxis]  # add an additional dimension for consi
    else:
        height, width, channels = image.shape
        output = np.zeros((height, width, channels))

    # Generate the coordinates
    coordinates = np.indices((height, width))
    x_coords = coordinates[1]
    y_coords = coordinates[0]

    # Compute new x' and y' for every x and y
    x_prime = a1*x_coords + a2*y_coords + a3*x_coords*y_coords + a4*x_coords**2
    y_prime = a7*x_coords + a8*y_coords + a9*x_coords*y_coords + a10*x_coords**2

    # Map the old image pixels to the new deformed positions
    for c in range(channels):  # for each channel
        output[:, :, c] = map_coordinates(image[:, :, c], [y_prime, x_prime], or

    if channels == 1:
        return output[:, :, 0]  # return as 2D grayscale image
    else:
        return output


def compute_third_order_polynomial_matrix_and_plot(images, img_size, landmarks1,
    """
    Computes a third-order polynomial transformation matrix based on landmark co
    between two images and applies this transformation to align one image with a
    also displays and saves the original and transformed images, enhancing their
    visibility.

    Parameters:
    - images (list of str): Paths to the source and target images.
    - img_size (int): The dimensions (height and width) to which the images shou
    - landmarks1 (list of tuples): Coordinates of landmarks in the source image.
    - landmarks2 (list of tuples): Corresponding coordinates of landmarks in the
    - rpth (str): Path to the directory where the resultant images will be saved
```

```
    - num (int): An identifier number for differentiating the output file names.
    - snum (int): Stage number for referencing in output.
    - disp_clip (float, optional): Clipping limit for the CLAHE algorithm, used

    Raises:
    - ValueError: If the list of landmarks from the source image is empty.

    Returns:
    tuple: Contains three elements:
        - imags (list of np.array): The original fixed and moving images along w
        - imgs (list of str): Paths to the saved output images.
        - coefficients (np.array): Coefficients of the third-order polynomial us

    Notes:
        This function is suited for complex registration tasks where finer contr
        The transformation matrix is applied to the target image to align it wit
        The images are displayed and saved with enhanced contrast to aid in visu
    """
    imags,imgs = [],[]
    img1 = cv2.resize(cv2.imread(images[0]), (img_size, img_size))
    img2 = cv2.resize(cv2.imread(images[1]), (img_size, img_size))

    imags.append(img1)
    imags.append(img2)

     # Check if the list is not empty
    if not landmarks1:  raise ValueError("Input list cannot be empty")

    # Check and delte the temporary folder if it exists
    if os.path.exists(os.path.join(os.getcwd(),'temp_dir')): shutil.rmtree(os.pa

    # Compute the third-order polynomial transformation matrix
    coefficients = compute_third_order_polynomial_matrix(landmarks1, landmarks2)
    coefficients_for_transform = compute_third_order_polynomial_matrix(landmarks

    # Apply the transformation using third-order polynomial
    transformed_image = warp_image_third_order_polynomial(img2, coefficients_for
    imags.append(transformed_image)

    # Display and save the images
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Stage-{} Results: Registration Using Third Order Polynomial Tr

    axs[0].imshow(CLAHE_plot_cond(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB),disp_cli
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(CLAHE_plot_cond(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB),disp_cli
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(CLAHE_plot_cond(cv2.cvtColor(transformed_image.astype(np.uint8
    axs[2].set_title('Deformed Image')
    axs[2].axis('off')

    plt.show();

    # saving intermediary results for better visualization
    imgs.append(os.path.join(rpth, 'Deformed_Image_' + str(num) + '_.png'))
    imgs.append(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'))
```

```
    cv2.imwrite(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'), img1)
    cv2.imwrite(os.path.join(rpth, 'Moving_' + str(num) + '_.png'), img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transforme
    return imags,imgs,coefficients

def compute_affine_matrix_and_plot(images,img_size,landmarks1, landmarks2,rpth,n
    """
    Computes an affine transformation matrix based on provided landmarks from tw
    this transformation to visually compare the source, target, and transformed

    This function computes the affine transformation matrix that best maps the s
    with the target image using landmark correspondences. It then applies this t
    the source image and displays the original (source and target) and transform
    The images are enhanced using CLAHE for better visibility and are saved to t

    Parameters:
    - images (list of str): File paths for the source and target images.
    - img_size (int): The size (width and height) to which the images will be re
    - landmarks1 (list of tuples): Landmark points (x, y) on the source image.
    - landmarks2 (list of tuples): Corresponding landmark points (x, y) on the t
    - rpth (str): Directory path where the resultant images will be saved.
    - num (int): Identifier number used to differentiate the output file names.
    - snum (int): Stage number for referencing in output.
    - disp_clip (float, optional): Clipping limit for the CLAHE algorithm, used

    Returns:
    - tuple: Contains two items:
        - imgs (list of str): Paths to the saved images.
        - affine_matrix (numpy.ndarray): The computed affine transformation matr

    Raises:
    - ValueError: If the landmarks list is empty, indicating insufficient data t

    Notes:
        The affine transformation matrix is computed using a least squares metho
        This function is useful for tasks in image registration where visual com
        Enhanced contrast is used to aid in the visual assessment of image regis
    """
    imgs,imags=[],[]
    img1 = cv2.resize(cv2.imread(images[0]),(img_size,img_size))
    img2 = cv2.resize(cv2.imread(images[1]),(img_size,img_size))

    imags.append(img1)
    imags.append(img2)

    # Check if the list is not empty
    if not landmarks1:  raise ValueError("Input list cannot be empty")

    # Check and delte the temporary folder if it exists
    if os.path.exists(os.path.join(os.getcwd(),'temp_dir')): shutil.rmtree(os.pa

    # Create the array with the specified format
    A = np.array([[xs, ys, 1] for xs, ys in landmarks1])

    X = np.array([xt for xt, yt in landmarks2])
    Y = np.array([yt for xt, yt in landmarks2])

    # Solve for the variables x1, y1, and z1
    sol1 = np.dot(np.dot(np.linalg.inv(np.dot(A.T,A)),A.T),X)
    sol2 = np.dot(np.dot(np.linalg.inv(np.dot(A.T,A)),A.T),Y)
```

```
    # Extract the variables
    x1, y1, z1 = sol1
    x2, y2, z2 = sol2
    affine_matrix = np.array([[x1,y1,z1],
                              [x2,y2,z2],
                              [0, 0, 1]])
    print("Affine Matrix:")
    print(affine_matrix)

    # Ensure the affine matrix is of type float32
    affine_matrix = affine_matrix.astype(np.float32)

    # Use only the top two rows for cv2.warpAffine
    affine_for_warp = affine_matrix[:2]

    # Apply the affine transformation using cv2.warpAffine
    transformed_image = cv2.warpAffine(img2, affine_for_warp, (img2.shape[1], im
    imags.append(transformed_image)

    # Display and save the images
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Stage-{} Results: Registration Using Affine Transformation".fo

    axs[0].imshow(CLAHE_plot_cond(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB),disp_cli
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(CLAHE_plot_cond(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB),disp_cli
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(CLAHE_plot_cond(cv2.cvtColor(transformed_image.astype(np.uint8
    axs[2].set_title('Deformed Image')
    axs[2].axis('off')

    plt.show();

    # saving intermediary results for better visualization
    imgs.append(os.path.join(rpth, 'Deformed_Image_' + str(num) + '_.png'))
    imgs.append(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'))
    cv2.imwrite(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'), img1)
    cv2.imwrite(os.path.join(rpth, 'Moving_' + str(num) + '_.png'), img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transforme
    return imgs,imags,affine_matrix

def compute_quadratic_matrix_and_plot(images,img_size,landmarks1, landmarks2,rpt
    """
    Computes a quadratic transformation matrix from source to target landmarks a
    to the source image. The transformed source image is displayed alongside the
    and all images are saved to disk.

    This function takes pairs of corresponding landmarks from the source and tar
    transformation matrix. This matrix is then used to warp the source image to
    result, along with the original images, is displayed and saved for compariso

    Parameters:
    - images (list of str): File paths for the source and target images.
    - img_size (int): The size (width and height) to which the images will be re
    - landmarks1 (list of tuples): Landmark points (x, y) on the source image.
    - landmarks2 (list of tuples): Corresponding landmark points (x, y) on the t
```

```
    - rpth (str): Directory path where the resultant images will be saved.
    - num (int): Identifier number used to differentiate the output file names.
    - cll (float, optional): Clipping limit for the CLAHE algorithm used in cont
    - snum (int): Stage number used for displaying in the title of the plot.
    - disp_clip (float, optional): Clipping limit for the CLAHE algorithm, used

    Returns:
    - tuple: Contains three items:
        - imgs (list of str): File paths where the output images are saved.
        - imags (list of np.array): List containing the numpy arrays of the orig
        - quadratic_matrix (numpy.ndarray): The computed quadratic transformatio

    Raises:
    - AssertionError: If the number of points in `landmarks1` and `landmarks2` a
                    number of points is required for matrix computation.

    Notes:
        The function uses OpenCV for image processing tasks such as reading, res
        The quadratic transformation matrix is computed using a least squares me
        Matplotlib is used for visualizing the before and after effects of the t
        This function is particularly useful in applications such as image regis
    """
    imgs,imags=[],[]
    img1 = cv2.resize(cv2.imread(images[0]),(img_size,img_size))
    img2 = cv2.resize(cv2.imread(images[1]),(img_size,img_size))

    imags.append(img1)
    imags.append(img2)

    # Check if the list is not empty
    if not landmarks1:  raise ValueError("Input list cannot be empty")

    # Check and delte the temporary folder if it exists
    if os.path.exists(os.path.join(os.getcwd(),'temp_dir')): shutil.rmtree(os.pa

    # Ensure the quadratic matrix is of type float32
    quadratic_matrix = compute_quadratic_matrix(landmarks1, landmarks2)

    quadratic_matrix_for_image_deformed = compute_quadratic_matrix(landmarks2, l

    print("quadratic Matrix:")
    print(quadratic_matrix)

    # Apply the quadratic transformation using cv2.warpquadratic
    transformed_image =  warp_image_quadratic_matrix(img2, quadratic_matrix_for_
    transformed_image = cv2.resize(transformed_image,  (img2.shape[1], img2.shap
    imags.append(transformed_image)

    # Display and save the images
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Stage-{} Results: Registration Using Quadratic Transformation"

    axs[0].imshow(CLAHE_plot_cond(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB),disp_cli
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(CLAHE_plot_cond(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB),disp_cli
    axs[1].set_title('Moving Image')
    axs[1].axis('off')
```

```
    axs[2].imshow(CLAHE_plot_cond(cv2.cvtColor(transformed_image.astype(np.uint8
    axs[2].set_title('Deformed Image')
    axs[2].axis('off')

    plt.show();

    # saving intermediary results for better visualization
    imgs.append(os.path.join(rpth, 'Deformed_Image_' + str(num) + '_.png'))
    imgs.append(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'))
    cv2.imwrite(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'), img1)
    cv2.imwrite(os.path.join(rpth, 'Moving_' + str(num) + '_.png'), img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transforme
    return imgs,imags,quadratic_matrix

def compute_homography_matrix_and_plot(images, img_size, landmarks1, landmarks2,
    """
    Computes the homography transformation matrix based on landmark correspondenc
    and applies this transformation to the source image. The function displays t
    target images along with the transformed source image. It also saves these i

    Parameters:
    - images (list of str): Paths to the source and target images.
    - img_size (int): The size to which both images will be resized.
    - landmarks1 (list of tuples): Landmark points (x, y) from the source image.
    - landmarks2 (list of tuples): Corresponding landmark points (x, y) from the
    - rpth (str): The directory path where the resultant images will be saved.
    - num (int): An identifier number used to differentiate the output file name
    - snum (int): Stage number used for displaying in the title of the plot.
    - disp_clip (float, optional): Clipping limit for the CLAHE algorithm, used

    Returns:
    - tuple: A tuple containing the paths to the saved images, a list of image a
           and the computed homography matrix.

    Raises:
    - ValueError: If the list of landmarks is empty, indicating that there are n

    Notes:
        The function uses OpenCV for image reading, resizing, and applying the h
        Matplotlib is used for displaying the images.
        Ensure the landmarks are accurately defined as their correspondence dire
        Homography transformations are particularly useful for applications in i
    """
    imgs,imags=[],[]
    img1 = cv2.resize(cv2.imread(images[0]),(img_size,img_size))
    img2 = cv2.resize(cv2.imread(images[1]),(img_size,img_size))

    imags.append(img1)
    imags.append(img2)

    # Check if the list is not empty
    if not landmarks1:  raise ValueError("Input list cannot be empty")

    # Check and delte the temporary folder if it exists
    if os.path.exists(os.path.join(os.getcwd(),'temp_dir')): shutil.rmtree(os.pa

    # Compute homography matrix
    homography_matrix = compute_homography_matrix(landmarks1, landmarks2)

    print("Homography Matrix:")
```

```
    print(homography_matrix)

    # Ensure the affine matrix is of type float32
    homography_matrix = homography_matrix.astype(np.float32)

    # Apply the homography transformation using cv2.warpPerspective
    transformed_image=cv2.warpPerspective(img2, homography_matrix, (img2.shape[1
    imags.append(transformed_image)

    # Display and save the images
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Stage-{} Results: Registration Using Homography Transformation

    axs[0].imshow(CLAHE_plot_cond(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB),disp_cli
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(CLAHE_plot_cond(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB),disp_cli
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(CLAHE_plot_cond(cv2.cvtColor(transformed_image.astype(np.uint8
    axs[2].set_title('Deformed Image')
    axs[2].axis('off')

    plt.show();

    # saving intermediary results for better visualization
    imgs.append(os.path.join(rpth, 'Deformed_Image_' + str(num) + '_.png'))
    imgs.append(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'))
    cv2.imwrite(os.path.join(rpth, 'Fixed_' + str(num) + '_.png'), img1)
    cv2.imwrite(os.path.join(rpth, 'Moving_' + str(num) + '_.png'), img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transforme
    return imgs,imags,homography_matrix
```

In [13]:

```
def landmark_error(point, transformed_point):
    """
    Computes the Euclidean distance between the original point and the transform

    Parameters:
    - point (tuple): Original point (x, y).
    - transformed_point (tuple): Transformed point (x, y).

    Returns:
    - float: Euclidean distance.
    """
    return np.linalg.norm(np.array(point) - np.array(transformed_point))

def estimate_affine_transformation(points):
    """
    Estimates the affine transformation matrix using point correspondences.

    Parameters:
    - points (np.array): Array of point correspondences.

    Returns:
    - np.array: Affine transformation matrix.
    """
    src_pts = np.float32([point[0] for point in points])
    dst_pts = np.float32([point[1] for point in points])
```

```
    affine_matrix, _ = cv2.estimateAffinePartial2D(src_pts, dst_pts)
    return affine_matrix

def estimate_homography_matrix(points):
    """
    Estimates the homography matrix given a set of point correspondences.

    Parameters:
    - points: A list of tuples, where each tuple contains two (x, y) tuples.
              The first tuple in each pair is from the first set of points (set1
              and the second tuple is the corresponding point in the second set

    Returns:
    - homography_matrix: The estimated (3x3) homography matrix.
    """
    import numpy as np
    import cv2

    # Separate the points into two sets
    set1 = [point[0] for point in points]
    set2 = [point[1] for point in points]

    # Convert to numpy arrays
    set1 = np.array(set1, dtype=np.float32)
    set2 = np.array(set2, dtype=np.float32)

    # Estimate the homography matrix
    homography_matrix, _ = cv2.findHomography(set1, set2, cv2.RANSAC)

    return homography_matrix

def remove_outliers_based_on_error_affine(set1, set2, threshold=20):
    """
    Filters out outlier point pairs from two sets of points by applying an affin
    removing pairs that have an error greater than a specified threshold. The fu
    an affine transformation matrix based on all given point pairs. Each point i
    transformed using this matrix, and the error is calculated as the Euclidean
    transformed point and the corresponding point in the second set. Points with
    threshold are considered outliers and are excluded from the results.

    Parameters:
    - set1 (list of tuples): A list of (x, y) tuples representing coordinates of
    - set2 (list of tuples): A list of (x, y) tuples representing corresponding
                             second image. The indices in `set1` and `set2` must c
    - threshold (float, optional): The maximum allowed error distance between th
                                   points for them to be considered inliers. Defau

    Returns:
    - tuple of lists: Returns two lists (updated_set1, updated_set2) containing
                      `set1` and `set2` respectively.
    Notes:
        It is critical that `set1` and `set2` are of equal length and that the p
        as any misalignment could result in incorrect calculations and poor resu
        This function is typically used in image processing and computer vision
        transformation of point sets between images is required, particularly in
    """
    points = list(zip(set1, set2))
    affine_matrix = estimate_affine_transformation(points)
    updated_set1 = []
    updated_set2 = []
```

```
    for point1, point2 in zip(set1, set2):
        transformed_point = transform_points_affine([point1], affine_matrix)[0]
        error = landmark_error(point2, transformed_point)

        if error <= threshold:
            updated_set1.append(point1)
            updated_set2.append(point2)

    return updated_set1, updated_set2

def remove_outliers_based_on_error_homography(set1, set2, threshold=20):
    """
    Filters out outlier point pairs from two sets of points by applying a homogr
    and removing pairs that have an error greater than a specified threshold. Th
    a homography transformation matrix based on all given point pairs. Each poin
    transformed using this matrix, and the error is calculated as the Euclidean
    transformed point and the corresponding point in the second set. Points with
    threshold are considered outliers and are excluded from the results.

    Parameters:
    - set1 (list of tuples): A list of (x, y) tuples representing coordinates of
    - set2 (list of tuples): A list of (x, y) tuples representing corresponding
                            second image. The indices in `set1` and `set2` must c
    - threshold (float, optional): The maximum allowed error distance between th
                                  points for them to be considered inliers. Defau

    Returns:
    - tuple of lists: Returns two lists (updated_set1, updated_set2) containing
                        `set1` and `set2` respectively.
    Notes:
        Ensure that `set1` and `set2` are of equal length and that the points co
        as any misalignment could result in incorrect calculations and poor resu
        This function is typically used in image processing and computer vision
        alignment and transformation of point sets between images are required,
        like panorama stitching and object tracking.
    """
    points = list(zip(set1, set2))
    homography_matrix = estimate_homography_matrix(points)
    updated_set1 = []
    updated_set2 = []

    for point1, point2 in zip(set1, set2):
        transformed_point = transform_points_homography([point1], homography_mat
        error = landmark_error(point2, transformed_point)

        if error <= threshold:
            updated_set1.append(point1)
            updated_set2.append(point2)

    return updated_set1, updated_set2

def filter_outlier_cond(computed,original,criteria='affine', thresh=20):
    """
    Filters out outliers based on a specified condition.

    This function processes two sets of points (computed and original) and filte

    Parameters:
    - computed (list of tuples): List of computed points as (x, y) coordinates.
```

```
    - original (list of tuples): List of original points as (x, y) coordinates t
    - criteria (str, optional): The criteria to use for filtering outliers. Opti
    - thresh (int, optional): Threshold value used in the outlier removal proces

    Returns:
    - list: A list containing the filtered computed points after outlier removal
    - list: A list containing the filtered original points after outlier removal

    Raises:
    - AssertionError: If the length of the computed points is not 3.

    Notes:
        If 'homography' is chosen as the criteria, the function estimates a homo
        If 'affine' is chosen, it removes outliers based on affine transformatio
    """
    assert len(computed) >= 3
    if criteria=='homography':
        computed,original = estimate_homography_matrix(computed,original,thresh)
    else:
        computed,original = remove_outliers_based_on_error_affine(computed,origi
    return computed,original
```

In [14]:

```
def main_initialization(images,N,img_size,max_dist,offset,window_size,clip):
    """
    Initializes image processing by applying CLAHE if specified, extracting keyp
    and computing the Discrete Fourier Transform (DFT) for the given images.

    Parameters:
        - images (list of str): List of image file paths that need processing.
        - N (int): Number of keypoints to detect or random points to select.
        - img_size (tuple of int): The dimensions (width, height) to which image
        - max_dist (float): Maximum distance between keypoints for the SIFT algo
        - offset (float): Offset used in the selection of random points.
        - window_size (int): Size of the window used in random point selection.
        - clip (float): Clipping limit for the CLAHE algorithm; if greater than

    Returns:
        - tuple:
            - images (list of np.array): The list of images after processing, po
            - pts(list of tuples): the list of detected points after applying SI
            - dft (np.array): The result of the Discrete Fourier Transform appli

    Notes:
        The function begins by extracting SIFT keypoints from the first image an
        It then applies CLAHE if the clipping limit is specified and computes th
    """
    pts = SIFT_top_n_keypoints(images[0],N,img_size,max_dist)
    pts = pts+select_random_points(images[0],N,img_size,offset,window_size)
    if clip > 0:
        images = CLAHE_Images(images, clip = clip)
    dft = DFT(images,img_size,pts)
    return images,pts,dft

def CLAHE_Images(imags,clip):
    """
    Applies Contrast Limited Adaptive Histogram Equalization (CLAHE) to a list o
    their contrast. This method is particularly useful for improving the visibil
    that suffer from poor contrast.

    Parameters:
```

```
    - imags (list of str): List of paths to the image files that need contrast e
    - clip (float): Clip limit for the CLAHE algorithm, which sets the threshold
                 The higher the clip limit, the more aggressive the contrast en

    Returns:
    - list of str: Returns a list of paths to the saved CLAHE-processed images.
                saved with a "CLAHE_" prefix in its filename to distinguish it

    Notes:
        This function uses OpenCV's `createCLAHE` method to apply the CLAHE algo
        first converted to grayscale as CLAHE is typically applied to single-cha
        visualization of detail.
        The images are processed in-place and saved in the same directory as the
        prefixed to their original filenames.
        It is recommended to adjust the `clip` parameter based on the specific r
        content and the desired level of contrast enhancement.
    """
    imgs=[]
    img_dir = os.path.join(os.getcwd(),'temp_dir')
    clahe = cv2.createCLAHE(clipLimit=clip, tileGridSize=(8, 8))
    os.makedirs(os.path.join(os.getcwd(),'temp_dir') ,exist_ok=True)
    for img in imags:
      fn,_ = os.path.splitext(os.path.basename(img))
      ifn = os.path.join(img_dir,'CLAHE'+'_'+str(fn)+'.png')
      imag = cv2.imread(img)
      imag = Image.fromarray(np.uint8(imag))
      imag = imag.convert('L')
      img = np.asarray(imag)
      image_equalized = clahe.apply(img)
      image_equalized_img = Image.fromarray(np.uint8(image_equalized))
      image_equalized = image_equalized_img.convert('RGB')
      image_equalized = np.asarray(image_equalized)
      cv2.imwrite(ifn,image_equalized);
      imgs.append(ifn)
    return imgs

def Feature_padding(feature_maps, size):
    """
    Pad feature maps to a uniform size using bilinear interpolation.

    This function adjusts the size of each feature map in the input list to a sp

    Parameters:
    - feature_maps (list of tensors): A list of feature map tensors to be resize
    - size (tuple): The target size for the feature maps as (height, width).

    Returns:
    - list: A list of uniformly sized feature maps.
    """
    uniform_feature_maps=[]
    for feature in feature_maps:
        uniform_feature_maps.append(F.interpolate(feature, size=size, mode='bili
    return uniform_feature_maps

def multi_resolution_features(orig_images,img_size,N,clip,offset,window_size,max
    """
    Generate multi-resolution features from images using SIFT, and Random Points

    This function processes images to generate feature maps at multiple resoluti
```

```
    Parameters:
    - orig_images (list of str): List of paths to the images to be processed.
    - img_size (int): The size of the images for processing.
    - N (int): The number of keypoints to be used in SIFT.
    - clip (float): The clip limit for CLAHE.
    - max_dist (float): Maximum distance for keypoint selection in SIFT.
    - timestep (float): Timestep parameter for Diffusion Model initialization.
    - up_ft_indices (list): Indices for feature upsampling in the Diffusion Mode
    - multi_ch (bool): Flag to indicate multi-channel mode.
    - multi_img_size (int): The size of the images for multi-resolution processi
    - multi_iter (int): Number of iterations for multi-resolution processing.

    Returns:
    - tuple: A tuple of source and target feature tensors.
    """
    if multi_ch:
        src_fts,trg_fts =[],[]
        for i in range(multi_iter):
            images,pts,dft = main_initialization(orig_images,N,multi_img_size*(i
            src_ft1,trg_ft1 = dft.feature_upsampling(RetinaRegNet_Intialization(
            src_fts.append(src_ft1)
            trg_fts.append(trg_ft1)
        src_fts = Feature_padding(src_fts,(img_size,img_size))
        trg_fts = Feature_padding(trg_fts,(img_size,img_size))
        src_ft = torch.cat(src_fts, dim=1)
        trg_ft = torch.cat(trg_fts, dim=1)
    else:
        images,pts,dft = main_initialization(orig_images,N,img_size,max_dist,off
        src_ft,trg_ft = dft.feature_upsampling(RetinaRegNet_Intialization(images
    return src_ft,trg_ft

def landmarks_condition_check(orig_images, img_size, pts, t, uft, landmarks1, la
    """
    Iteratively attempts to improve image registration quality by enhancing imag
    until certain quality conditions are met or a maximum number of attempts is
    for image contrast enhancement and uses various feature transformation and s
    of landmark correspondences between two images.

    Parameters:
    - orig_images (list of str): Paths to the original images to be processed.
    - img_size (int): Size of the images to be processed, assumed to be square.
    - pts (list): List of all sampled feature keypoints in the image
    - t (float): Threshold parameter for initializing the Diffusion Model.
    - uft (float): Parameter for extracting diffusion features from the diffusio
    - landmarks1 (list of tuples): Initial landmarks as (x, y) coordinates in th
    - landmarks2 (list of tuples): Target landmarks as (x, y) coordinates in the
    - max_tries (int, optional): Maximum number of attempts to improve image reg
    - num (int, optional): Minimum required number of landmarks. Defaults to 100
    - iccl (float, optional): Inverse consistency criteria limit used in landmar
    - outlier_cond (str, optional): Condition used to determine outliers. Defaul
    - thresh (float, optional): Threshold used for filtering outliers. Defaults

    Returns:
    - tuple: Depending on the success of the registration process, this function
        The original images and the best set of landmarks found, or
        The original images and a set of default landmarks if conditions are not

    Raises:
    - AssertionError: If the number of initial and target landmarks do not match
```

```
    Notes:
        This function is particularly useful in medical imaging or computer visi
        registration is crucial for further analysis.
        The effectiveness of the registration process depends heavily on the qua
        CLAHE and other image processing techniques may not always produce the d
        are of poor quality or the initial landmarks are inaccurately defined.
    """
    imgs,lim,land_marks1, land_marks2,list_landmarks_2, list_sim_scores,list_lan
    tries, ch, = 0, 0
    assert len(landmarks1) == len(landmarks2), f"Points lengths are incompatible
    landmarks2,landmarks1 = filter_outlier_cond(landmarks2,landmarks1,outlier_cc
    list_landmarks_1.append(landmarks1)
    imgs.append(orig_images)
    list_landmarks_2.append(landmarks2)
    if len(landmarks2) < num:
        print("Image Registration Unsuccessful for Original Set of Images")
        while len(land_marks2) < num and tries< max_tries:
            print("Executing Trial", tries + 1)
            dft = DFT(orig_images, img_size, pts)
            src_ft,trg_ft = dft.feature_upsampling(RetinaRegNet_Intialization(or
            land_marks1,sim_score, land_marks2 = dft.feature_maps(src_ft,trg_ft,
            del src_ft
            del trg_ft
            torch.cuda.empty_cache()
            gc.collect()
            land_marks2,land_marks1 = filter_outlier_cond(land_marks2,land_marks
            list_landmarks_1.append(land_marks1)
            imgs.append(images)
            list_landmarks_2.append(land_marks2)
            list_sim_scores.append(np.mean(sim_score))
            tries += 1
        for i in range(len(list_landmarks_2)):
            lim.append(len(list_landmarks_2[i]))
        idx = np.argmax(np.array(lim))
        return orig_images,list_landmarks_1[idx],list_landmarks_2[idx]
    else:
        return orig_images, landmarks1, landmarks2
```

In [15]:

```
def folder_structure(path):
    """
    Creates a directory structure for storing image registration results.

    Parameters:
    - path (str): Base path for the directory.
    """
    os.makedirs(os.path.join(path+'_'+'Image_Registration_Results'), exist_ok=Tr

def subject_organization(nfn,fls):
    """
    Organize subjects based on file naming conventions.

    Parameters:
    - nfn (list): List of subject names.
    - fls (list): List of filenames.

    Returns:
    - dict: Dictionary with subjects as keys and their corresponding files as va
    """
    result_lists = {f'Subject_{i + 1}': [] for i in range(len(nfn))}
    for i in fls:
```

```
        if '_'.join(map(str,i.split('.')[0].split('_')[-2:])) in nfn:
            result_lists[str('_'.join(map(str,i.split('.')[0].split('_')[-2:])))
        else:
            continue
    return result_lists

def elements_replication(fixed, temp):
    """
    Replicate elements of a list based on another list's elements.

    Parameters:
    - fixed (list): List containing elements to replicate.
    - temp (list): List containing numbers indicating how many times to replicat

    Returns:
    - list: List with replicated elements.
    """
    fxd = []
    for i in range(len(fixed)):
        replicated_sublist = [fixed[i][0]] * temp[i]
        fxd.append(replicated_sublist)
    return fxd

def data_organizing(pth,nfn,fnn,result_lists):
    """
    Organize data based on filenames and subject names.

    Parameters:
    - pth (str): Base path to the dataset.
    - nfn (list): List of subject names.
    - fnn (list): List of file identifiers.
    - result_lists (dict): Dictionary with subjects as keys and their correspond

    Returns:
    - tuple: Lists containing paths to fixed images, moving images, and point fi
    """
    fixed,moving,pnts,temp=[],[],[],[]
    for i in nfn:
        a,b,c=[],[],[]
        for j in range(len(result_lists[i])):
            if result_lists[i][j].split('_')[1].startswith(str(fnn[1][0])):
                b.append(os.path.join(pth,str(i),fnn[1],result_lists[i][j]))
            elif result_lists[i][j].startswith(str(fnn[2][0])):
                a.append(os.path.join(pth,str(i),fnn[2],result_lists[i][j]))
            else:
                c.append(os.path.join(pth,str(i),fnn[0],result_lists[i][j]))
        temp.append(len(b))
        fixed.append(sorted(a))
        moving.append(sorted(b))
        pnts.append(sorted(c))
    fixed = elements_replication(fixed,temp)
    return fixed,moving,pnts

def text_points_extraction(pnts):
    """
    Extract point coordinates from a text file.

    Parameters:
    - pnts (str): Path to the text file containing point coordinates.
```

```
    Returns:
    - tuple: Lists of fixed points and moving points.
    """
    fixed_pnts = []
    moving_pnts = []
    with open(pnts, 'r') as file:
        for line in file:
            points = [float(coord) for coord in line.strip().split(',')]
            fps = tuple(points[:2])
            lps = tuple(points[2:])
            fixed_pnts.append(fps)
            moving_pnts.append(lps)
    return fixed_pnts,moving_pnts

def info_extraction(fixed,moving, pnts):
    """
    Extract information from given lists to pair up images and their points.

    Parameters:
    - fixed (list): List of fixed image paths.
    - moving (list): List of moving image paths.
    - pnts (list): List of point file paths.

    Returns:
    - tuple: Lists of image pairs, fixed points, and moving points.
    """
    images,fixed_points,moving_points=[],[],[]
    assert len(fixed) == len(moving), f"Some Imgaes do not have a pair: {len(fix
    for i in range(len(fixed)):
        for j in range(len(moving)):
            if i==j:
                for k in range(len(fixed[i])):
                    for l in range(len(moving[j])):
                        if k==l:
                            images.append([moving[j][l],fixed[i][k]])
                            p1,p2 = text_points_extraction(pnts[j][l])
                            fixed_points.append(p1)
                            moving_points.append(p2)
                        else:
                            continue
            else:
                continue
    return images,fixed_points,moving_points

def coordinates_rescaling_high_scale(pnts,H,W,img_shape):
    """
    Rescale a list of coordinates based on given height and width ratios.

    Parameters:
    - pnts (list of tuples): List of (x, y) coordinates to be rescaled.
    - H (int): Original height.
    - W (int): Original width.
    - img_shape (int): Desired image dimension (assumes square shape).

    Returns:
    - list of tuples: List of rescaled (x, y) coordinates.
    """
    scaled_points=[]
    for row in pnts:
        a = (row[0]/W)*img_shape[1]
```

```
        b = (row[1]/H)*img_shape[0]
        scaled_points.append((a,b))
    return scaled_points

def coordinates_rescaling(pnts,H,W,img_shape):
    """
    Rescale a list of coordinates based on given height and width ratios.

    Parameters:
    - pnts (list of tuples): List of (x, y) coordinates to be rescaled.
    - H (int): Original height.
    - W (int): Original width.
    - img_shape (int): Desired image dimension (assumes square shape).

    Returns:
    - list of tuples: List of rescaled (x, y) coordinates.
    """
    scaled_points=[]
    for row in pnts:
        a = (row[0]/W)*img_shape
        b = (row[1]/H)*img_shape
        scaled_points.append((a,b))
    return scaled_points

def coordinates_processing(image1,image2,fpnts,mpnts,img_shape=256):
    """
    Process and rescale coordinates for two images.

    Parameters:
    - image1 (str): Path to the first image.
    - image2 (str): Path to the second image.
    - fpnts (list of tuples): List of (x, y) coordinates related to the first im
    - mpnts (list of tuples): List of (x, y) coordinates related to the second i
    - img_shape (int, optional): Desired image dimension for rescaling. Default

    Returns:
    - tuple: A tuple containing:
        - Tuple: Height and Width of the second image.
        - Tuple: Height and Width of the first image.
        - int: Maximum of the heights and widths of both images.
        - list of tuples: Scaled coordinates for the first image.
        - list of tuples: Scaled coordinates for the second image.
        - list of tuples: Scaled original coordinates for the second image.
    """
    H1,W1,C1 = cv2.imread(image1).shape
    H2,W2,C2 = cv2.imread(image2).shape
    scaled_moving_points = coordinates_rescaling(mpnts,H1,W1,img_shape)
    scaled_fixed_points = coordinates_rescaling(fpnts,H2,W2,img_shape)
    scaled_original_moving_points = coordinates_rescaling(mpnts,H1,W1,max(max(H1
    return (H2,W2),(H1,W1),max(max(H1,W1),max(H2,W2)),scaled_fixed_points,scaled

def feature_scaling(images,fixed_points,moving_points,img_shape):
    """
    Apply feature scaling to given images and their associated points.

    Parameters:
    - images (list): List of tuples containing image paths for fixed and moving
    - fixed_points (list): List of fixed points corresponding to each image.
    - moving_points (list): List of moving points corresponding to each image.
    - img_shape (int): Desired image dimension for rescaling.
```

```
    Returns:
    - tuple: A tuple containing:
        - list: Sizes of fixed images.
        - list: Sizes of moving images.
        - list: Maximum of the heights and widths of the images.
        - list: Fixed points after scaling.
        - list: Moving points after scaling.
        - list: Scaled moving points.
    """
    fixed_image_size,moving_image_size,max_image_size,fixed_pointss,moving_point
    for i in range(len(images)):
        fhs,mhss,mhs,fpnts,mpnts,scmpnts = coordinates_processing(images[i][0],i
        fixed_image_size.append(fhs)
        moving_image_size.append(mhss)
        max_image_size.append(mhs)
        fixed_pointss.append(fpnts)
        moving_pointss.append(mpnts)
        scaled_moving_points.append(scmpnts)
    return fixed_image_size,moving_image_size,max_image_size,fixed_pointss,movin

def data_preprocessing(path):
    """
    Preprocess the dataset from a given path by organizing and extracting releva

    Parameters:
    - path (str): Path to the dataset directory.

    Returns:
    - tuple: A tuple containing:
        - list: Extracted images from the dataset.
        - list: Fixed points associated with the images.
        - list: Moving points associated with the images.
    """
    fls,nfn,fnn=[],[],[]
    pth = os.path.join(os.getcwd(),path,'data')
    for i in sorted(os.listdir(pth)):
        nfn.append(i)
        for j in os.listdir(os.path.join(os.getcwd(),path,'data',i)):
            fnn.append(j)
            for k in os.listdir(os.path.join(os.getcwd(),path,'data',i,j)):
                if k!='.ipynb_checkpoints':
                    fls.append(k)
                else:
                    continue
    folder_structure(path)
    result= subject_organization(nfn,fls)
    fixed,moving,pnts = data_organizing(pth,nfn,np.unique(fnn),result)
    images,fixed_points,moving_points=info_extraction(fixed,moving,pnts)
    return images,fixed_points,moving_points
```

In [16]:

```
def RetinaRegNet_Intialization(filelist,img_size = 256,timestep = 75,up_ft_index
    """
    Initialize RetinaRegNet by processing a list of image files.

    Parameters:
    - filelist (list of str): List of paths to image files for feature extractio
    - img_size (int, optional): Desired size for resizing images. Default is 256
    - timestep (int, optional): Time step for the intializing the diffusion mode
    - up_ft_index (int, optional): Index for the extracting diffusion features f
```

```
    Returns:
    - ft (torch.Tensor): A tensor containing the Diffusion features of the image

    Notes:
        The function uses the SDFeaturizer from the 'stabilityai/stable-diffusio
        from each image. After processing all images, the extracted features are
        To avoid memory issues, the function cleans up resources after processin
    """
    ft = []
    imglist = []
    dfm = SDFeaturizer(sd_id='stabilityai/stable-diffusion-2-1')
    for filename in filelist:
        img = Image.open(filename).convert('RGB')
        img = img.resize((img_size, img_size))
        imglist.append(img)
        img_tensor = (PILToTensor()(img) / 255.0 - 0.5) * 2
        ft.append(dfm.forward(img_tensor,
                              timestep,
                              up_ft_index,
                              prompt='FLoRI21',
                              ensemble_size=8))
    ft = torch.cat(ft, dim=0)

    del dfm
    torch.cuda.empty_cache()
    gc.collect()
    return ft
```

In [17]:

```
def main(orig_images,rpth,ifn,stage_num,img_size=256,up_ft_indices = 1,timestep
    """
    Perform image registration and point correspondence using a series of proces

    Parameters:
    - images (list): A list of input images for registration.
    - rpth (str): Path to save the resulting registered images.
    - ifn (str): File name prefix for the saved images.
    - stage_num (int): Stage number for referencing in plots and outputs.
    - img_size (int, optional): Size of the input images (default is 256).
    - up_ft_indices (int, optional): Up-sampling factor for feature indices (def
    - timestep (int, optional): Time step for feature extraction (default is 75)
    - N (int, optional): Number of keypoints to extract (default is 50).
    - offset (float, optional): Offset parameter for feature extraction (default
    - window_size (int, optional): Size of the window for feature extraction (de
    - max_dist (int, optional): Maximum distance for feature matching (default i
    - iccl (int, optional): ICC level for feature matching (default is 3).
    - outlier_cond (str, optional): Condition for outlier removal (default is 'a
    - thresh (int, optional): Threshold value for outlier removal (default is 20
    - max_tries (int, optional): Maximum number of attempts for matching feature
    - num (int, optional): Number of iterations for matching features (default i
    - clip (float, optional): Clip parameter for image enhancement (default is 1
    - disp_clip (float, optional): Clip parameter for enhancing quality of image
    - multi_ch (bool, optional): Flag indicating whether to use multi-channel pr
    - multi_iter (int, optional): Number of iterations for multi-channel process
    - multi_img_size (int, optional): Size of images for multi-channel processin

    Returns:
    - original (list): List of original image points.
    - computed (list): List of computed image points after registration.
```

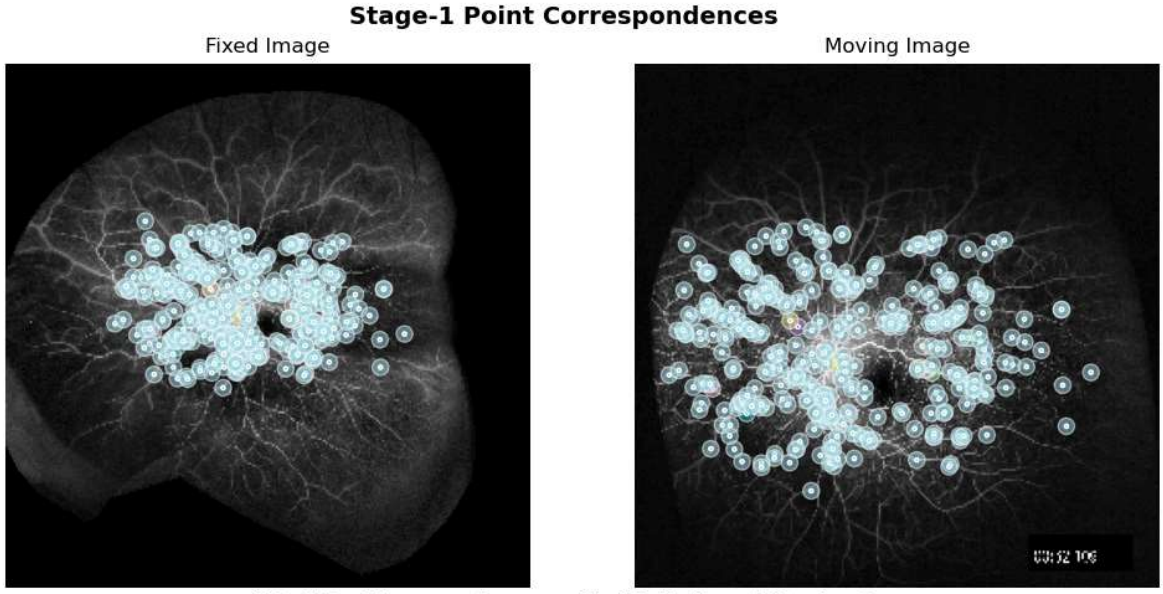
```
    Note:
        This function performs various processing steps including feature extrac
        outlier removal, and image registration.
        It saves the resulting registered images in the specified directory.
        If the image registration is unsuccessful, empty lists are returned for
    """
    images,pts,dft = main_initialization(orig_images,N,img_size,max_dist,offset,
    src_ft,trg_ft = multi_resolution_features(orig_images,img_size,N,clip,offset
    pnts,rmaxs, rspts = dft.feature_maps(src_ft,trg_ft,iccl)
    del src_ft
    del trg_ft
    torch.cuda.empty_cache()
    gc.collect()
    images,original,computed = landmarks_condition_check(images, img_size, pts,
    if len(computed)!=0:
        image_point_correspondences(images[::-1],img_size,computed,original,rpth
        return original,computed
    else:
        print("Image Registration is Unsuccessful for the presented Images due t
        return [],[]
    torch.cuda.empty_cache()
```

In [18]:
```
img_size= 1024
images,fixed_points,moving_points = data_preprocessing('FLoRI21_DataPort')
fixed_image_size,moving_image_size,max_image_size,scaled_fixed_points,scaled_mov
```

In [19]:
```
landmark_errors=[]
for i in range(len(images)):
    print("Case {}".format(i))
    print("Loading Fixed Images {0} Moving Image{1} to the framework".format(ima
    original_low_res,computed_low_res = main(images[i],os.path.join(os.getcwd(),
    imags,imgs,homography_matrix_low_res = compute_homography_matrix_and_plot(im
    if len(homography_matrix_low_res) !=0:
        transformed_points_hom = transform_points_homography(scaled_moving_point
        transformed_points_high_res_hom =  coordinates_rescaling(transformed_poi
        original_low_res,computed_low_res = main(imags,os.path.join(os.getcwd(),
        imag,imgs,polynomial_matrix_low_res = compute_third_order_polynomial_mat
        if len(polynomial_matrix_low_res) !=0:
            ## rescaled version for dispaly purposes
            transformed_points_poly = transform_points_third_order_polynomial(tr
            original_image_point_correspondences(imag,images[i][0],img_size, sca
            ### Original Version for computation of errors
            polynomial_matrix = transform_points_third_order_polynomial_matrix(c
            bef_error = compute_landmark_error(fixed_points[i],fixed_image_size[
            aft_error = compute_landmark_error_fixed_space(polynomial_matrix,fix
            print("Mean Landmark Error for Case {0} Before Registration is {1} p
            print("Mean Landmark Error for Case {0} After Registration is {1} pi
            landmark_errors.append(aft_error)
        else:
            landmark_errors.append(10000)
    else:
        landmark_errors.append(10000)
```

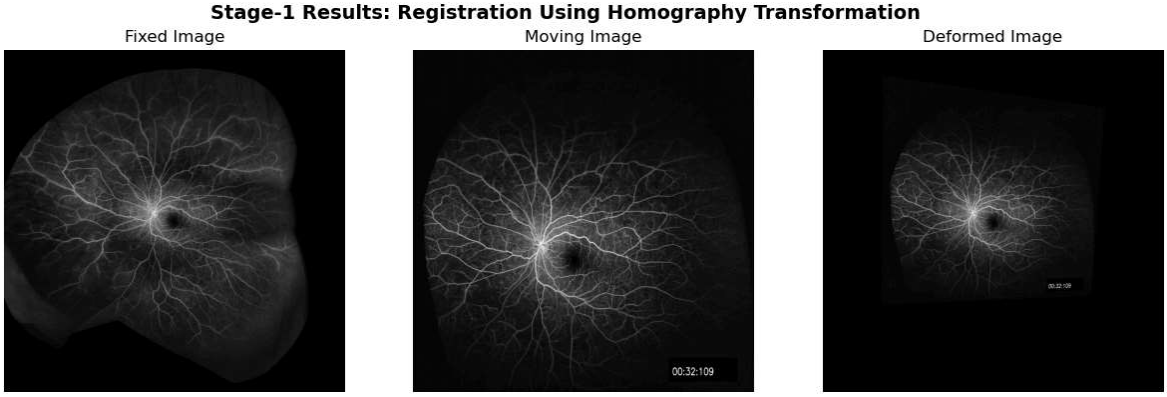
```
Case 0
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_1/Montag
e/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_1/FA/Raw_FA_1_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

```
/blue/weishao/vi.sivaraman/conda/envs/VBS_HRC/lib/python3.11/site-packages/torch/
nn/modules/conv.py:459: UserWarning: Applied workaround for CuDNN issue, install
nvrtc.so (Triggered internally at ../aten/src/ATen/native/cudnn/Conv_v8.cpp:80.)
  return F.conv2d(input, weight, bias, self.stride,
/scratch/local/29752099/ipykernel_108069/3422066467.py:101: UserWarning: To copy
construct from a tensor, it is recommended to use sourceTensor.clone().detach() o
r sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(so
urceTensor).
  points_indices = torch.tensor(pts_list)
```

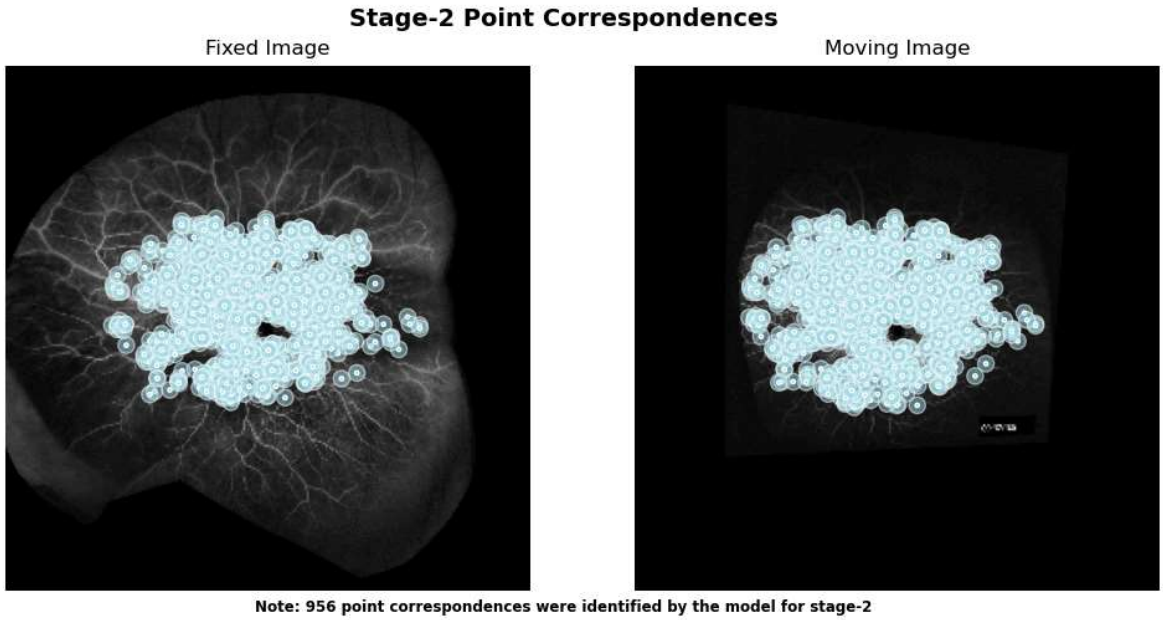


Note: 352 point correspondences were identified by the model for stage-1

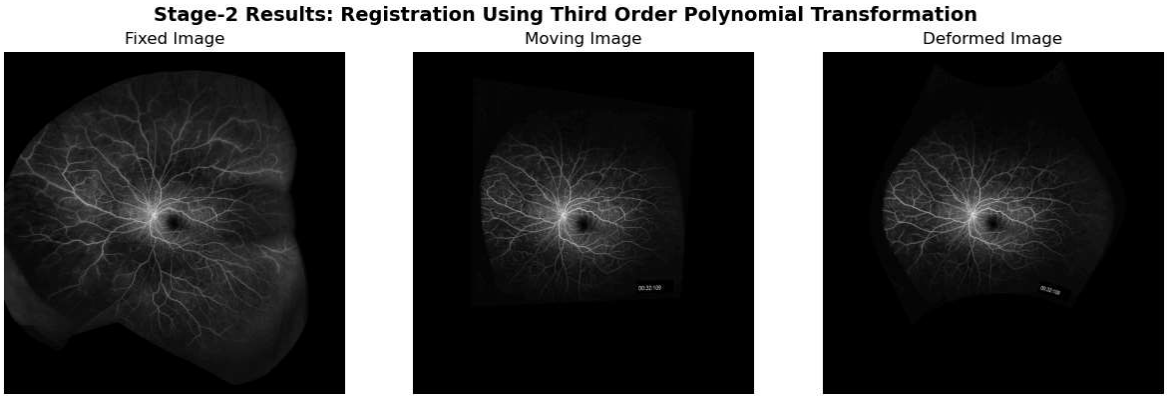
```
Homography Matrix:
[[8.36809377e-01 6.10898814e-03 1.81061464e+02]
 [1.33628243e-01 7.22583570e-01 7.63164084e+01]
 [2.21178941e-04 6.87989316e-05 1.00000000e+00]]
```

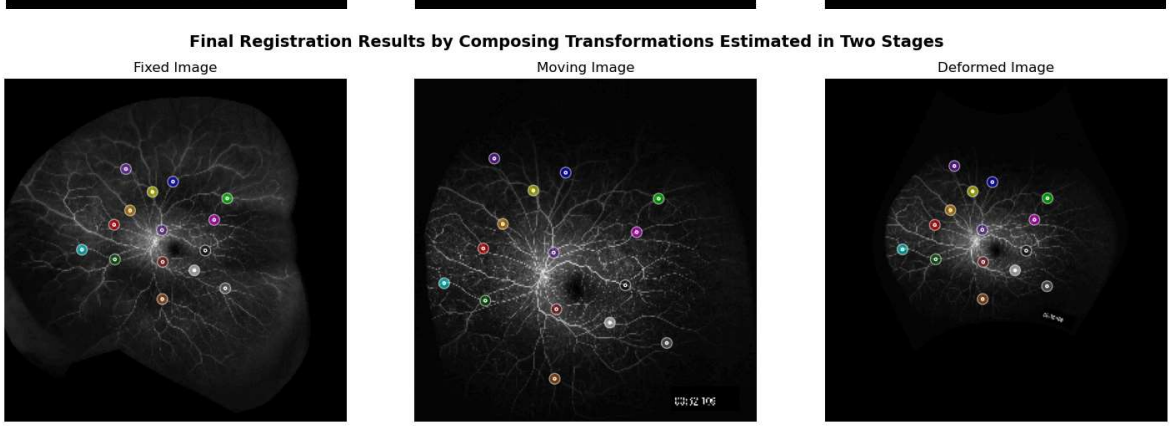



```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 956 point correspondences were identified by the model for stage-2

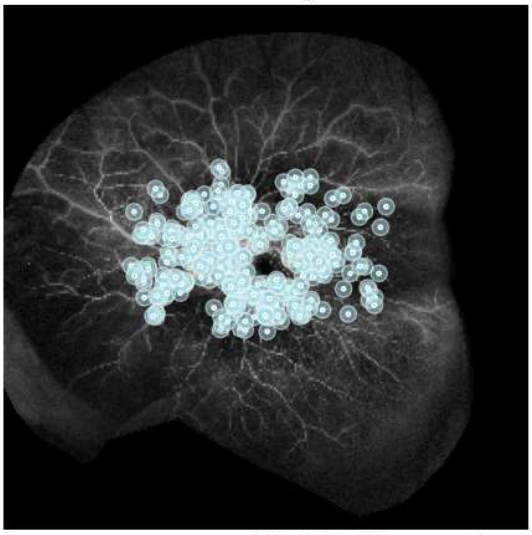




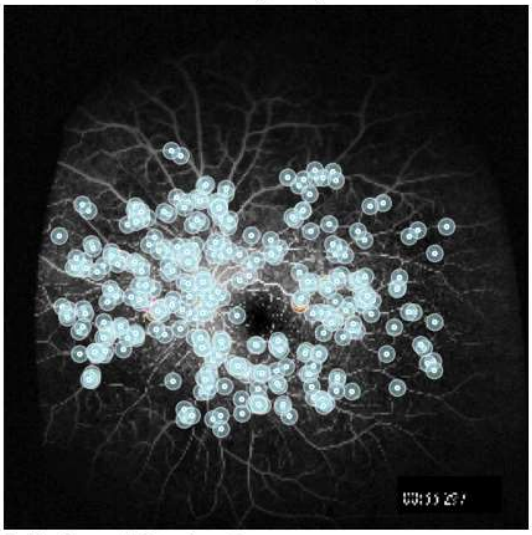

```
Mean Landmark Error for Case 0 Before Registration is 660.5727742589274 pixels
Mean Landmark Error for Case 0 After Registration is 14.677732620402322 pixels
Case 1
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_1/Montag
e/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_1/FA/Raw_FA_2_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image

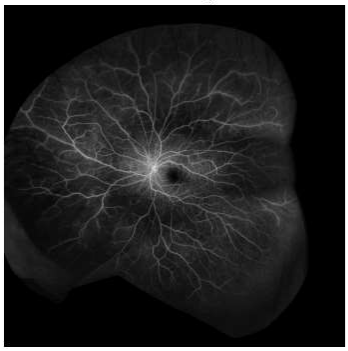


Note: 363 point correspondences were identified by the model for stage-1

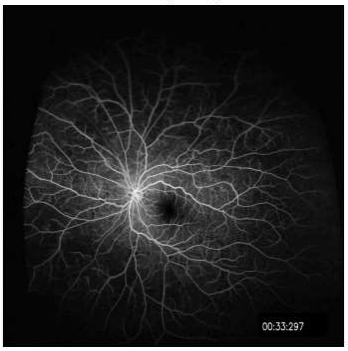
```
Homography Matrix:
[[ 7.53857977e-01 -2.43700933e-03  1.81877662e+02]
 [ 5.97363039e-02  6.55288326e-01  1.26952992e+02]
 [ 1.49194532e-04  1.93116884e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

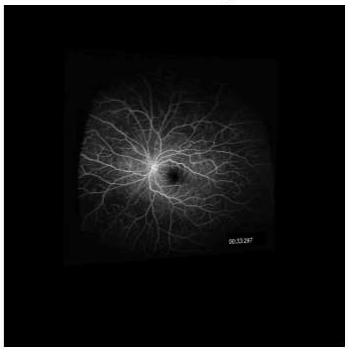
Fixed Image



Moving Image



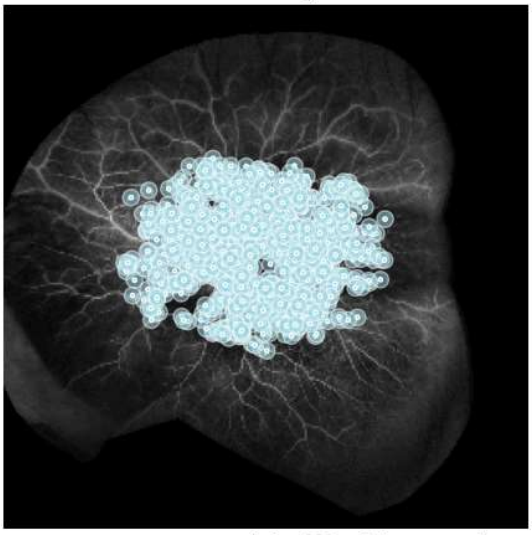
Deformed Image



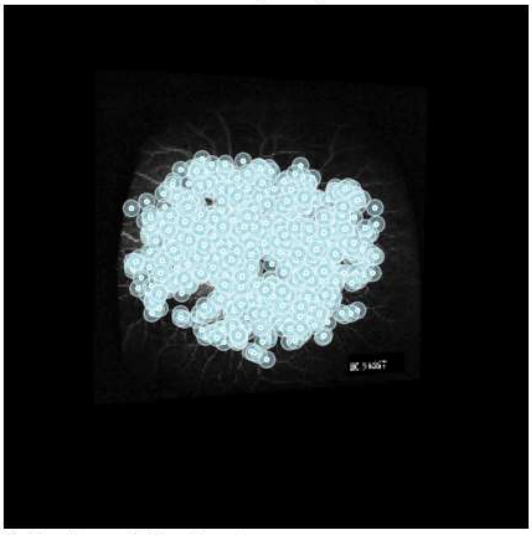
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

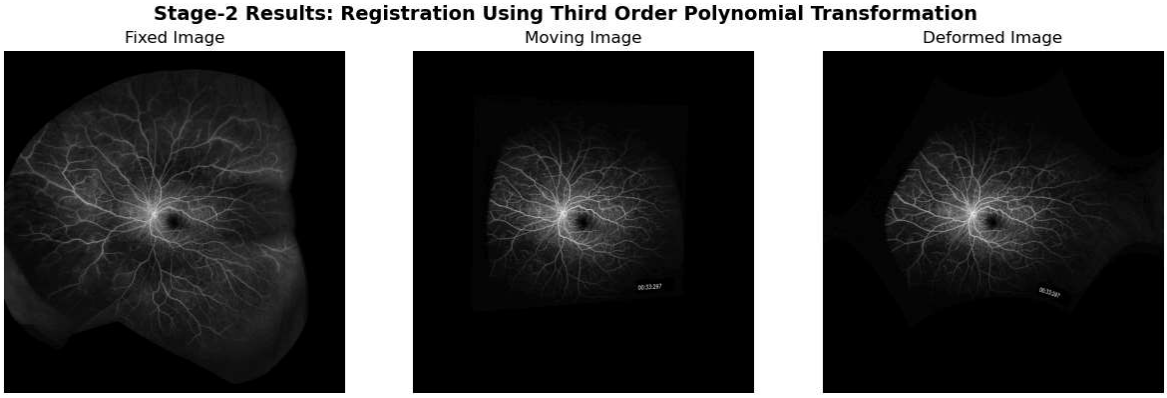
Fixed Image

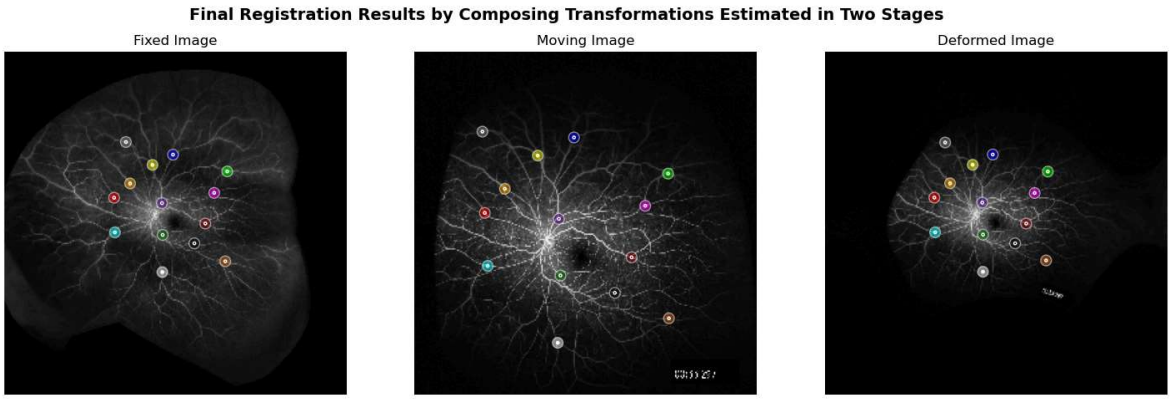


Moving Image

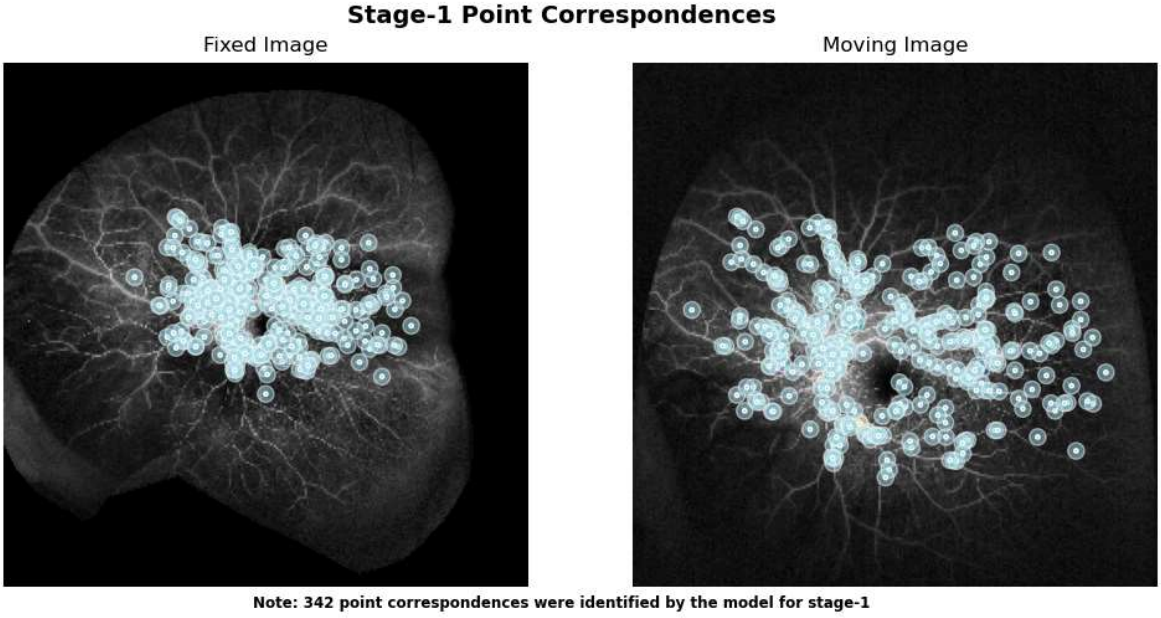


Note: 1009 point correspondences were identified by the model for stage-2



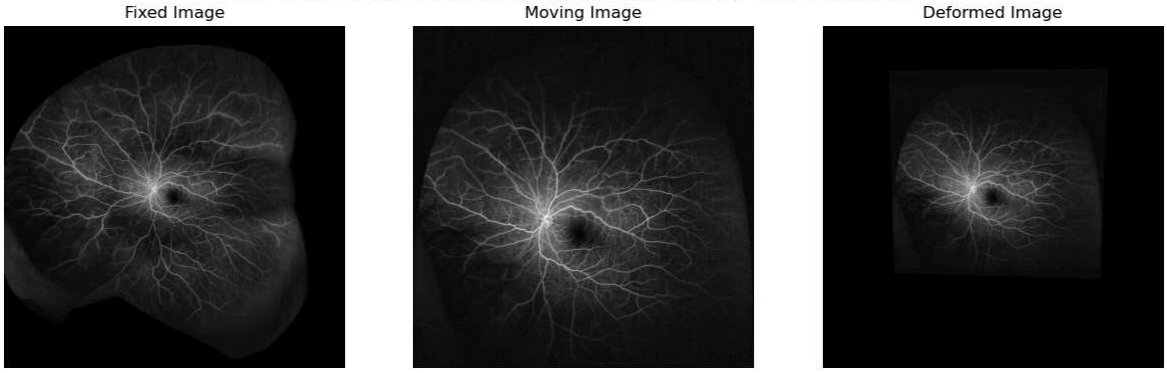



```
Mean Landmark Error for Case 1 Before Registration is 673.4817677455603 pixels
Mean Landmark Error for Case 1 After Registration is 18.284216684942585 pixels
Case 2
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_1/Montag
e/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_1/FA/Raw_FA_3_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



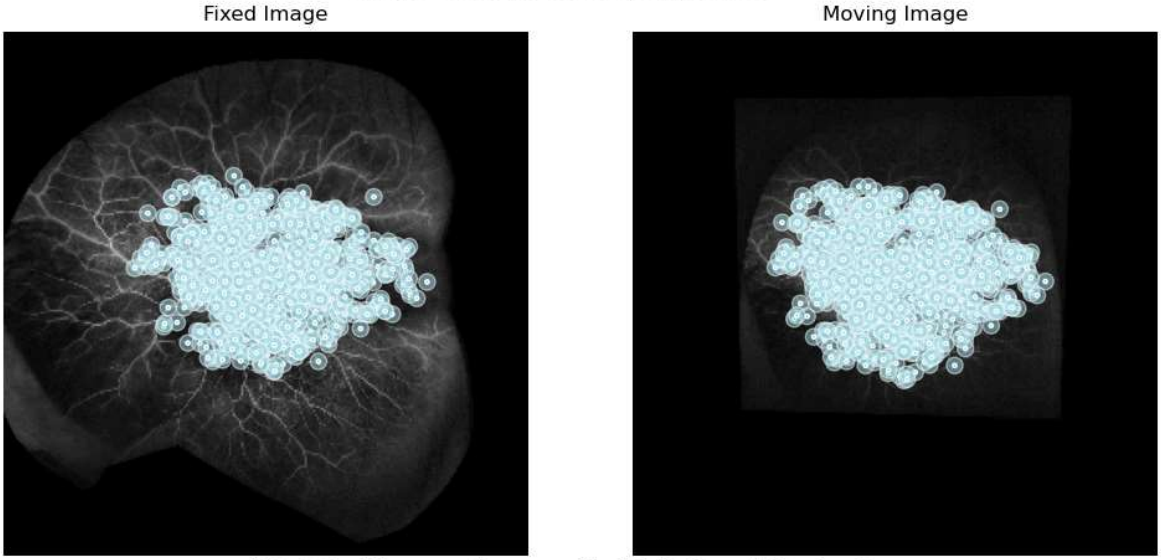

```
Homography Matrix:
[[ 6.11366068e-01  2.60387885e-02  1.97897922e+02]
 [-9.89625748e-03  6.34208445e-01  1.32945616e+02]
 [-3.75186097e-05  5.68504244e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation



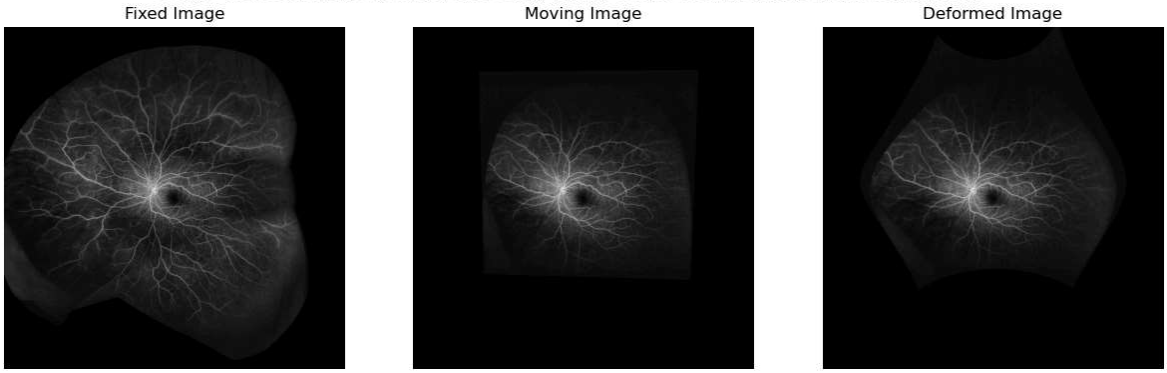

```
Loading pipeline components...:   0%|          | 0/6 [00:01<?, ?it/s]
```

Stage-2 Point Correspondences

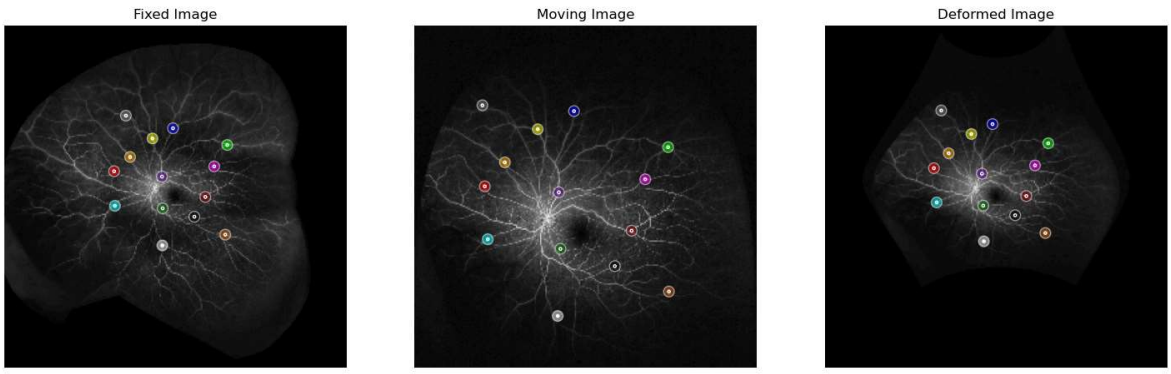


Note: 801 point correspondences were identified by the model for stage-2

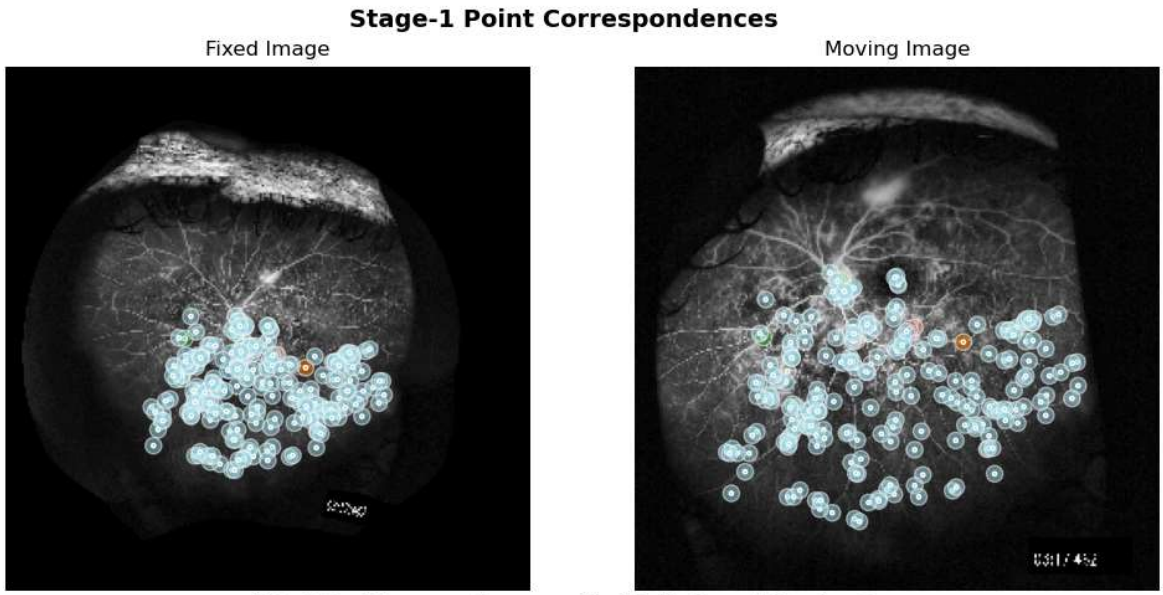
Stage-2 Results: Registration Using Third Order Polynomial Transformation



Final Registration Results by Composing Transformations Estimated in Two Stages

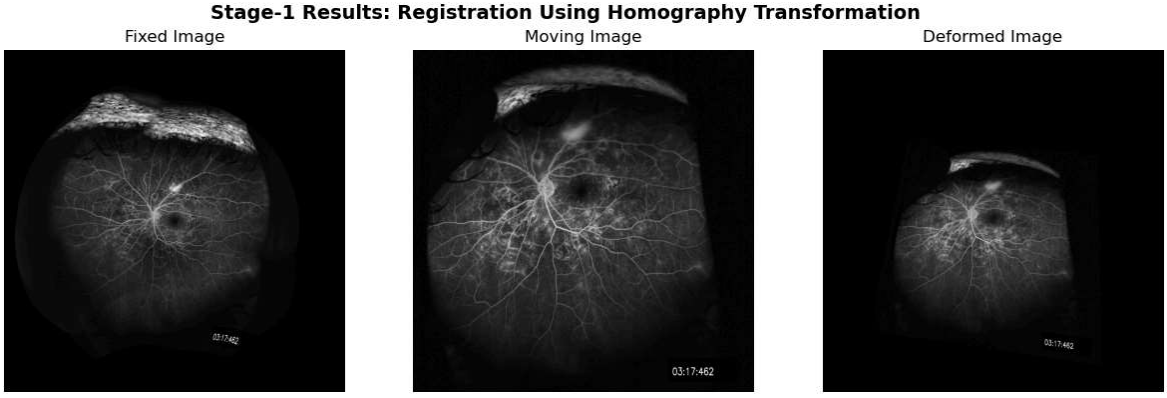


```
Mean Landmark Error for Case 2 Before Registration is 673.4817677455603 pixels
Mean Landmark Error for Case 2 After Registration is 39.83396702843833 pixels
Case 3
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montag
e/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_2/FA/Raw_FA_1_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

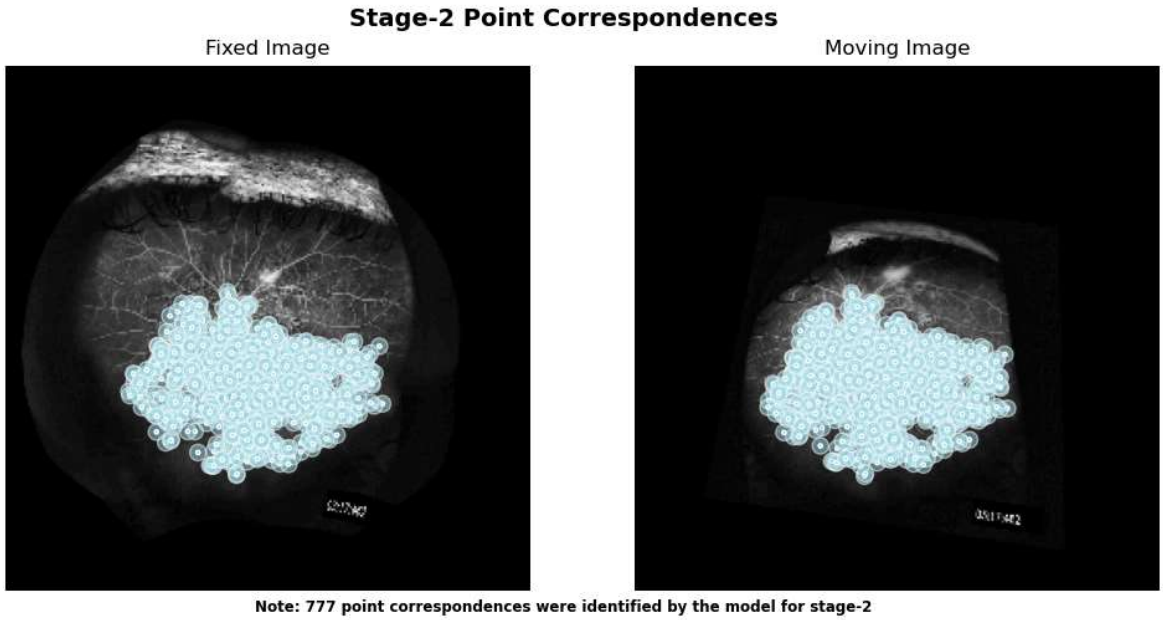


Note: 228 point correspondences were identified by the model for stage-1

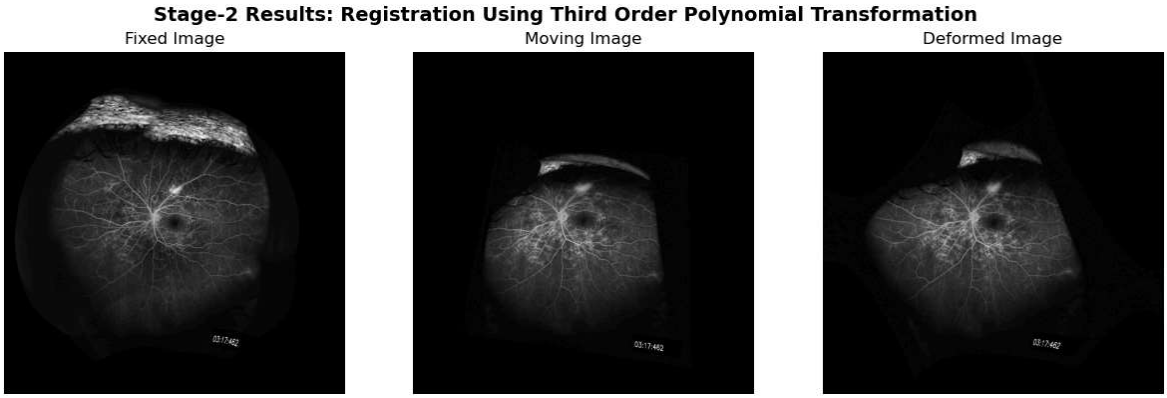
```
Homography Matrix:
[[ 5.26050417e-01 -1.43197641e-01  2.56878525e+02]
 [ 4.73274307e-02  4.19401174e-01  2.54028905e+02]
 [-3.71295995e-05 -1.83559018e-04  1.00000000e+00]]
```

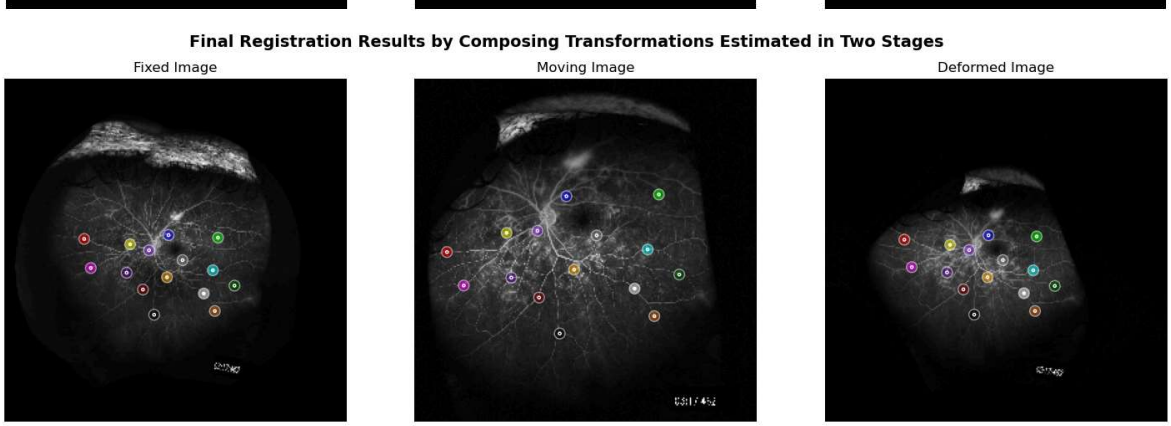



```
Maximum attempts reached, unable to find sufficient points with the specified cri
teria.
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 777 point correspondences were identified by the model for stage-2

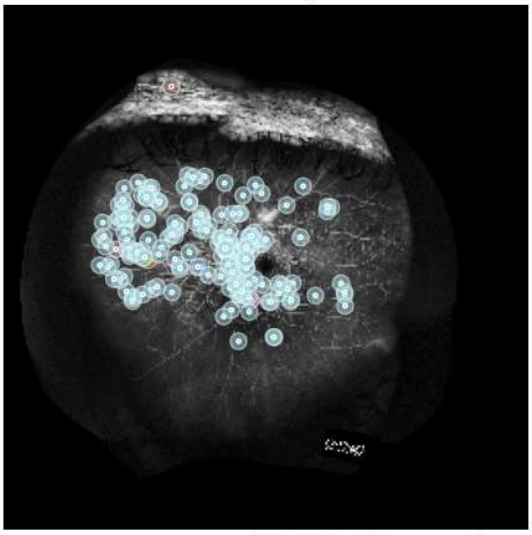




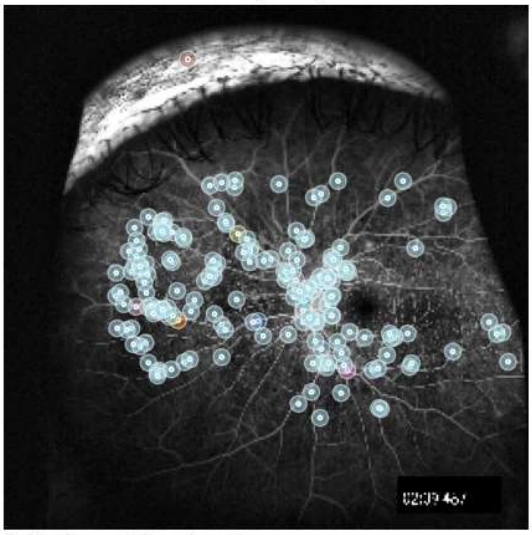

```
Mean Landmark Error for Case 3 Before Registration is 1013.6612836814534 pixels
Mean Landmark Error for Case 3 After Registration is 9.464472795521617 pixels
Case 4
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montag
e/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_2/FA/Raw_FA_2_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image



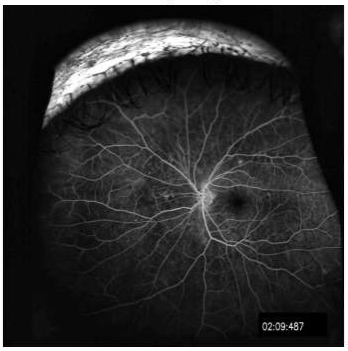
Note: 183 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 8.33782044e-01 -8.81644524e-02  7.77449723e+01]
 [ 2.12662586e-01  7.46925236e-01  1.43827514e+01]
 [ 2.54815840e-04  1.37338773e-05  1.00000000e+00]]
```

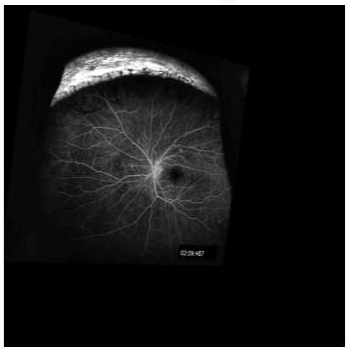
Stage-1 Results: Registration Using Homography Transformation

Fixed Image

Moving Image



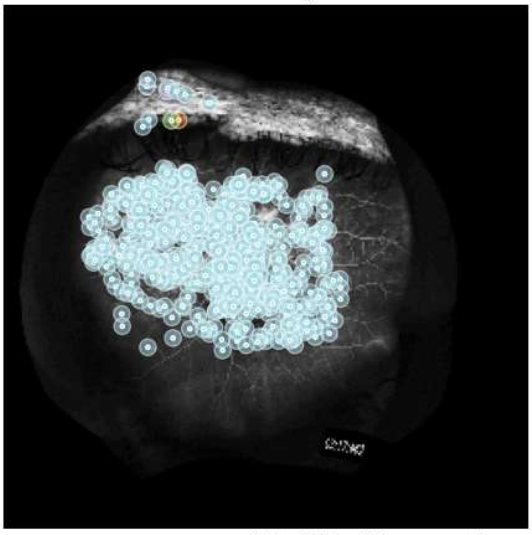
Deformed Image



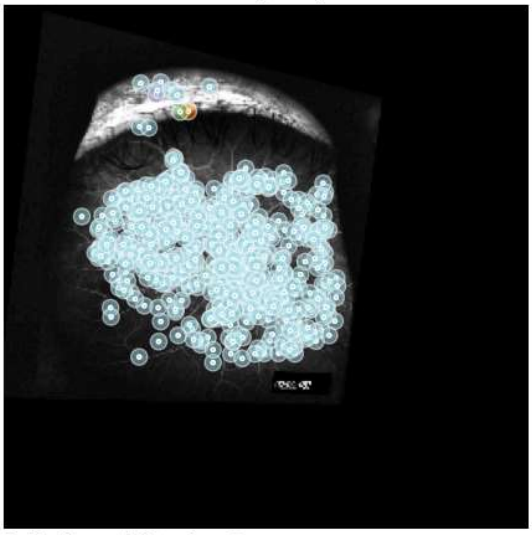
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

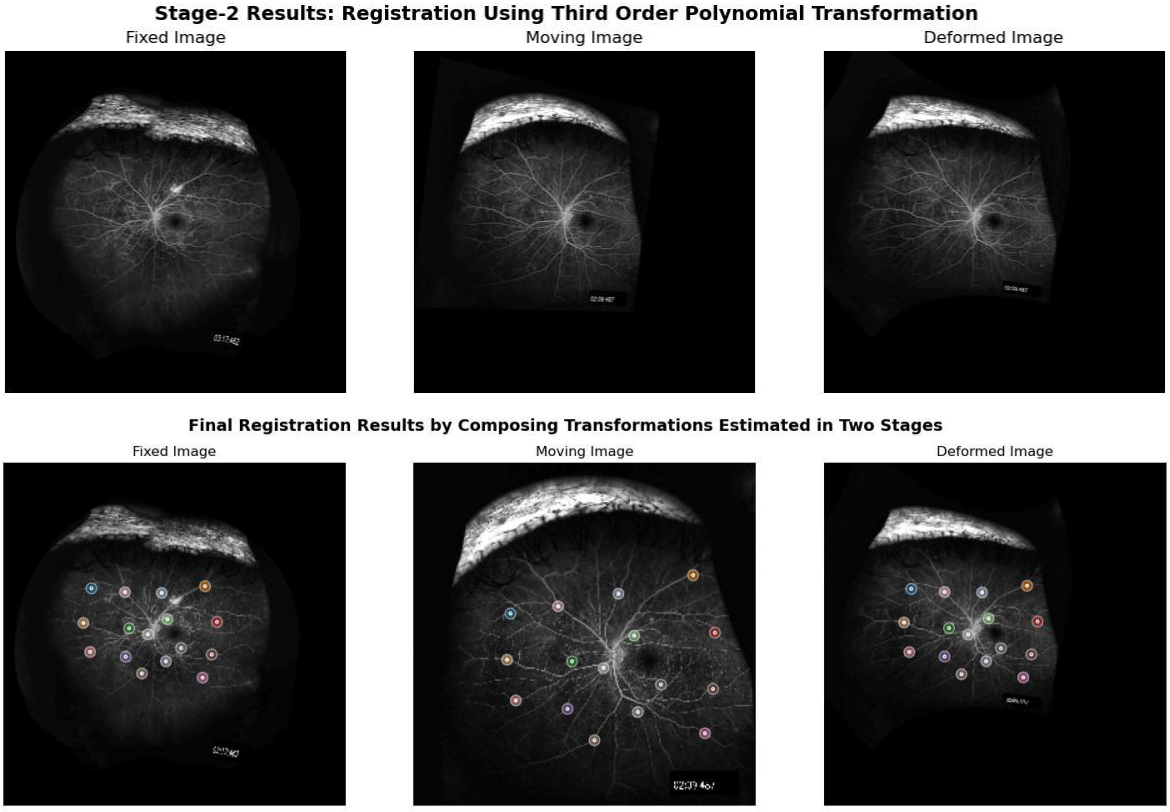
Fixed Image



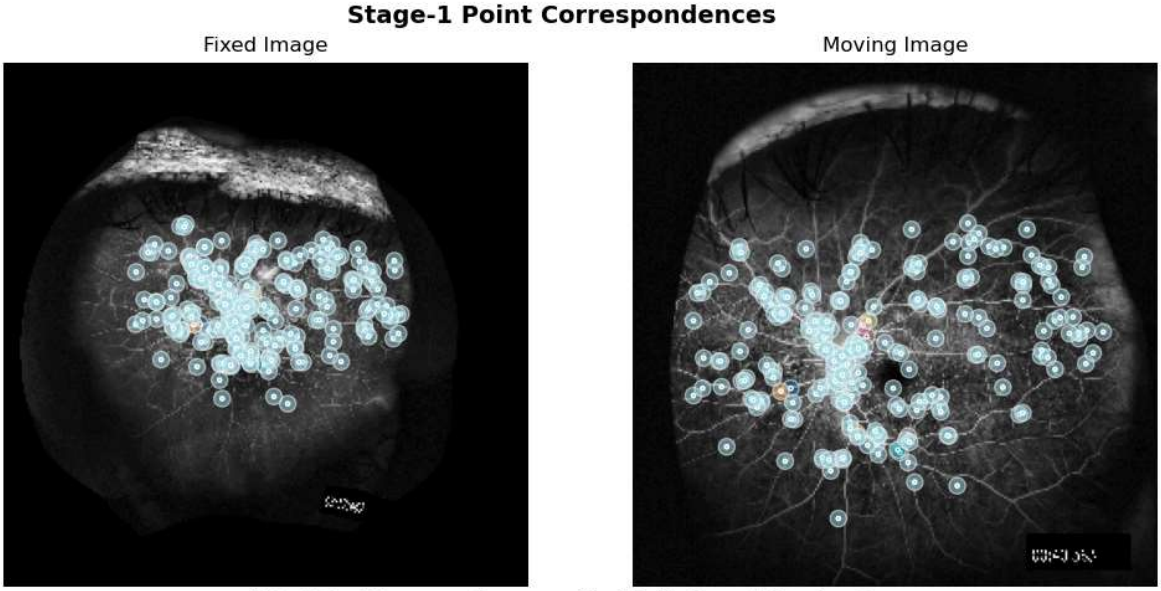
Moving Image



Note: 607 point correspondences were identified by the model for stage-2



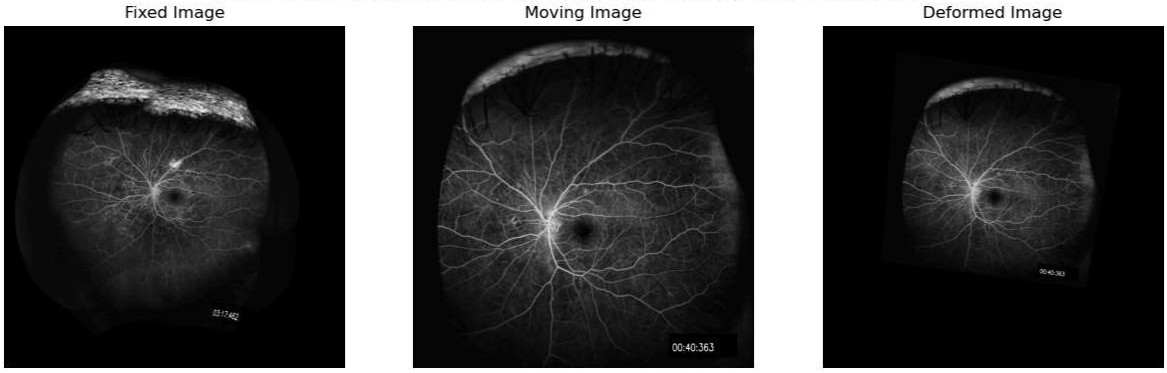

```
Mean Landmark Error for Case 4 Before Registration is 776.2780167748374 pixels
Mean Landmark Error for Case 4 After Registration is 9.332517886329612 pixels
Case 5
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montag
e/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_2/FA/Raw_FA_3_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 231 point correspondences were identified by the model for stage-1

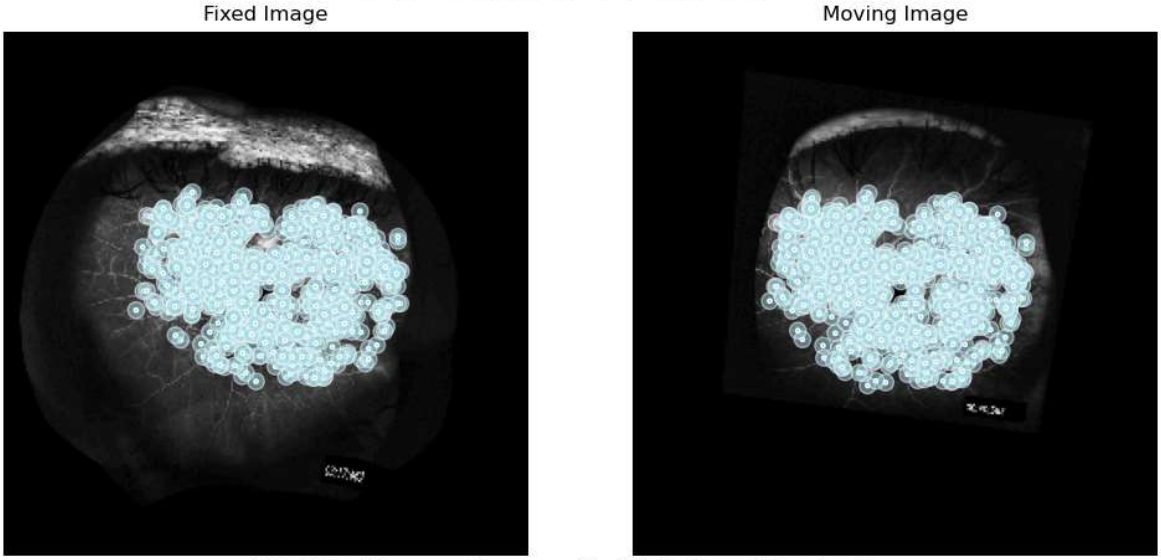
```
Homography Matrix:
[[ 6.79602697e-01 -2.78666439e-02  2.20820370e+02]
 [ 1.02506723e-01  6.84144689e-01  7.65696230e+01]
 [ 1.98921049e-05  9.92629113e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation



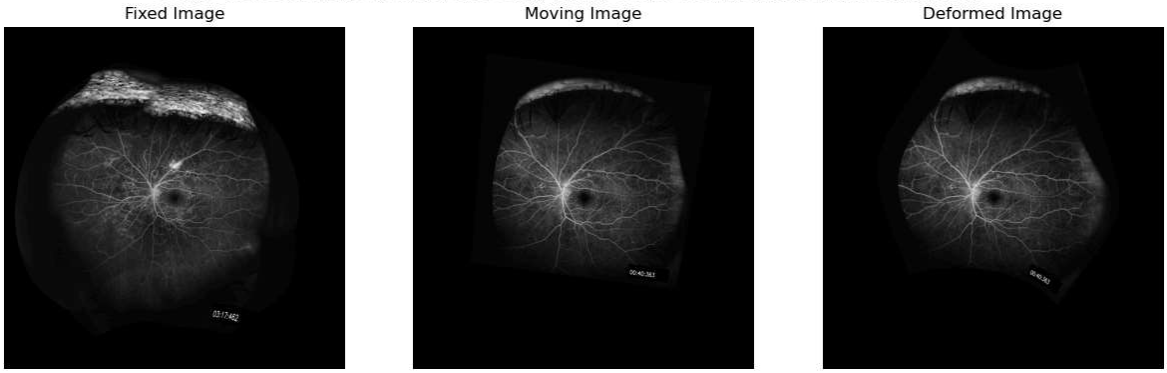

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences



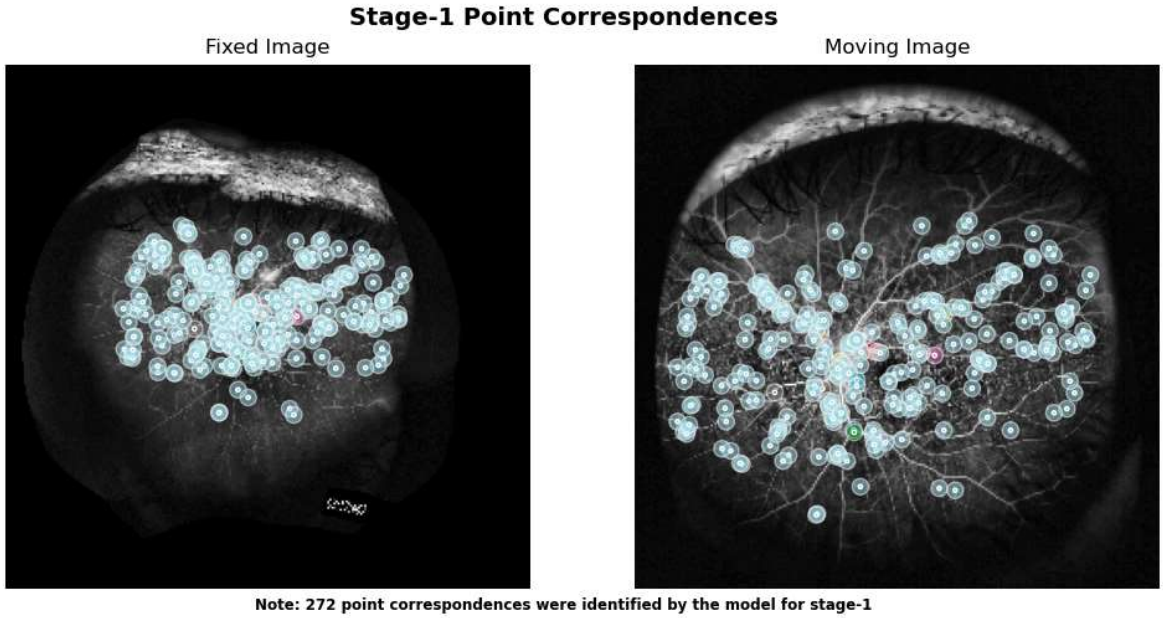
Note: 795 point correspondences were identified by the model for stage-2

Stage-2 Results: Registration Using Third Order Polynomial Transformation



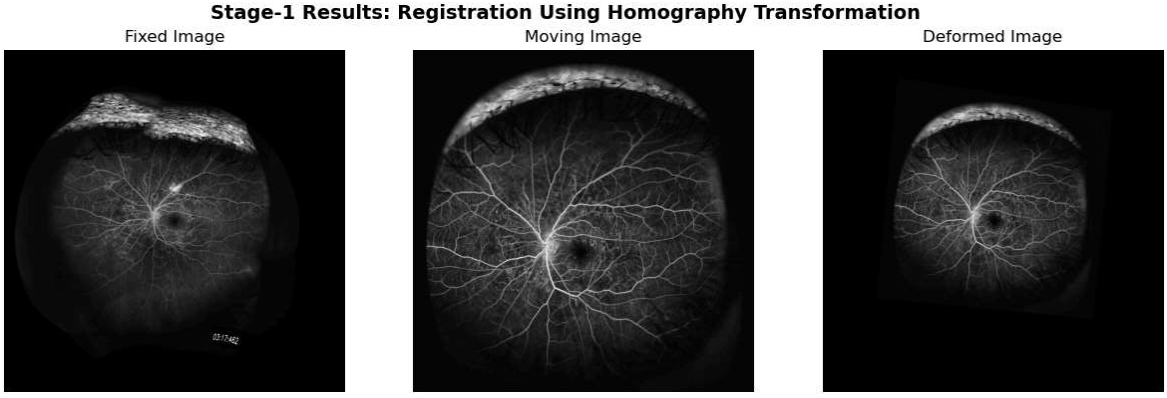
Final Registration Results by Composing Transformations Estimated in Two Stages

```
Mean Landmark Error for Case 5 Before Registration is 674.9261885180194 pixels
Mean Landmark Error for Case 5 After Registration is 12.727586390119303 pixels
Case 6
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montag
e/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_2/FA/Raw_FA_4_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

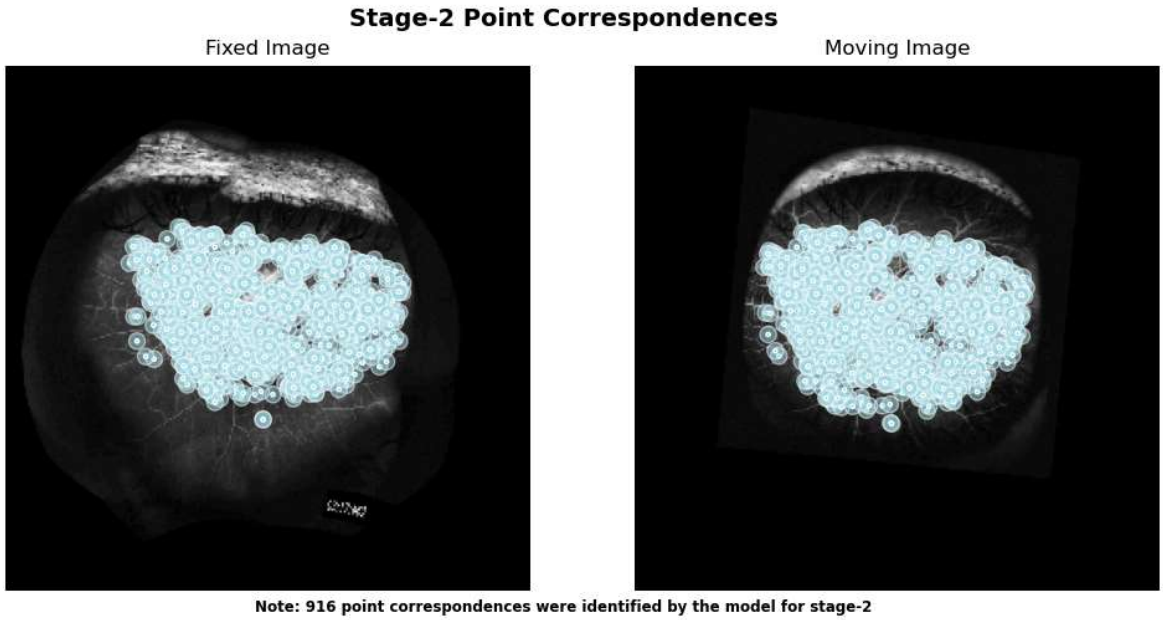


Note: 272 point correspondences were identified by the model for stage-1

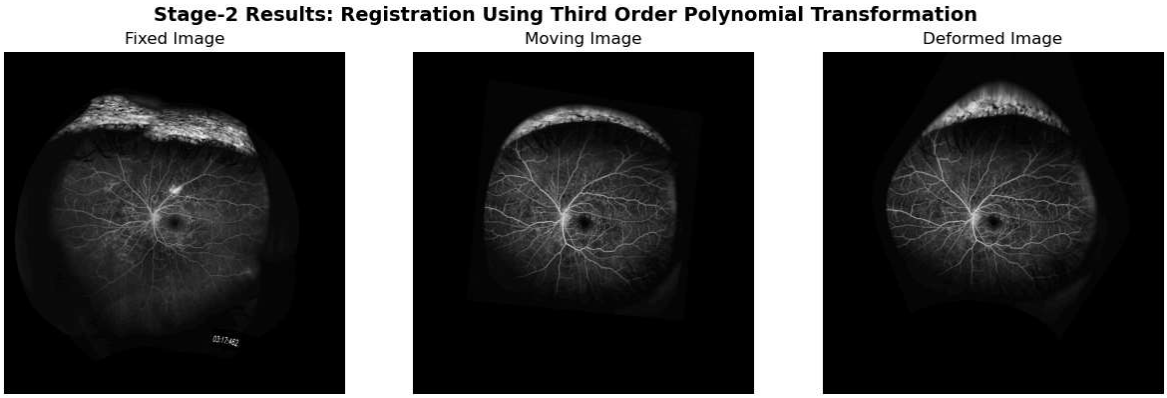
```
Homography Matrix:
[[ 6.81603351e-01 -6.26797428e-02  2.24259458e+02]
 [ 1.08331862e-01  6.41612516e-01  8.31877180e+01]
 [ 5.89664257e-05 -3.47483109e-06  1.00000000e+00]]
```

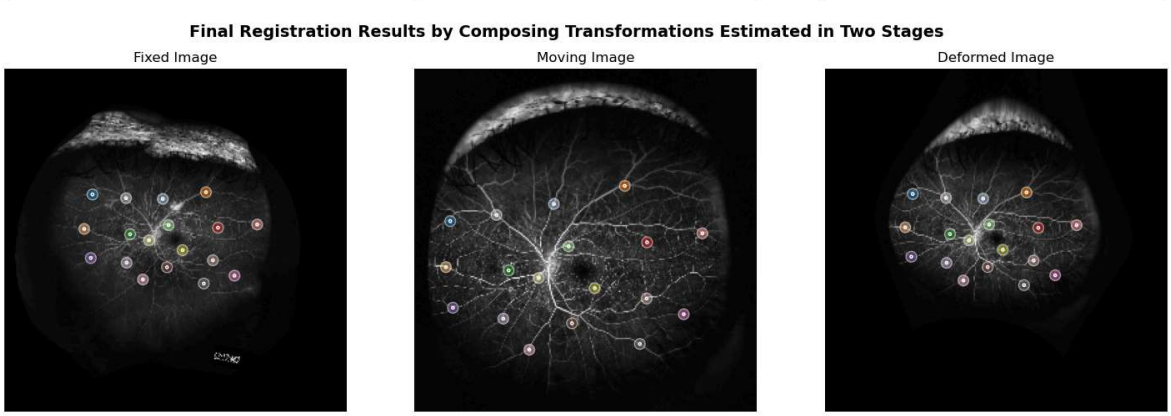



```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 916 point correspondences were identified by the model for stage-2

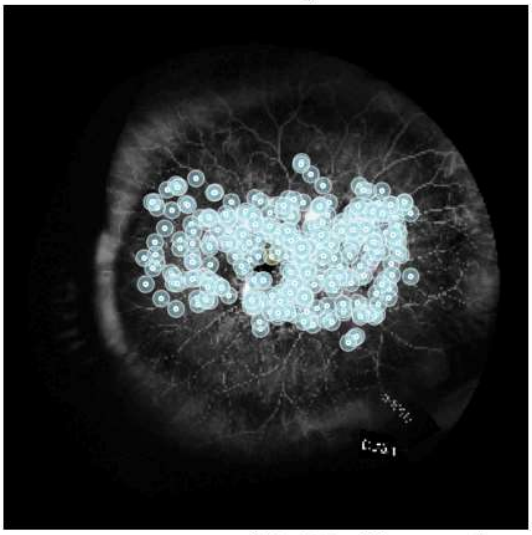




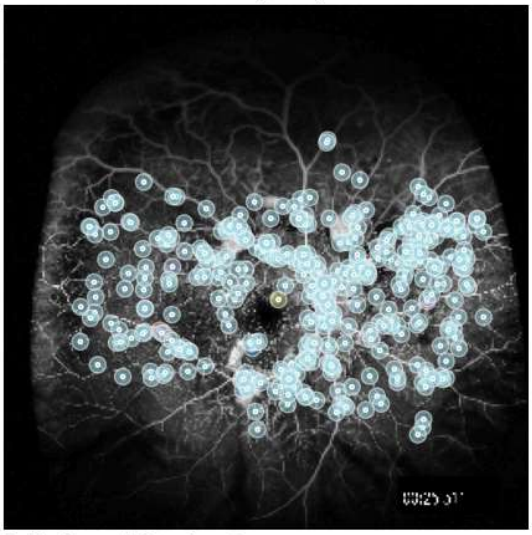

```
Mean Landmark Error for Case 6 Before Registration is 683.3470707329728 pixels
Mean Landmark Error for Case 6 After Registration is 11.879233154926741 pixels
Case 7
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_3/Montag
e/Montage_Subject_3.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_3/FA/Raw_FA_1_Subject_3.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image

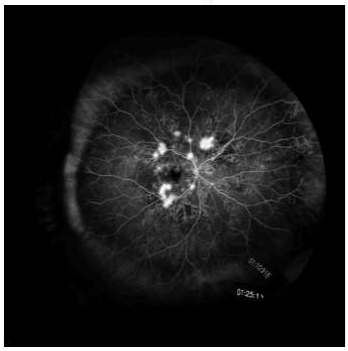


Note: 389 point correspondences were identified by the model for stage-1

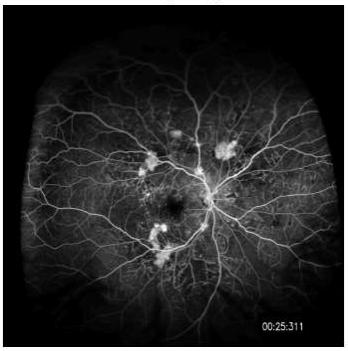
```
Homography Matrix:
[[ 5.80497854e-01  4.90399855e-02  1.96209646e+02]
 [-2.73551844e-02  6.80128081e-01  1.25618619e+02]
 [-9.50744308e-05  1.21124220e-04  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

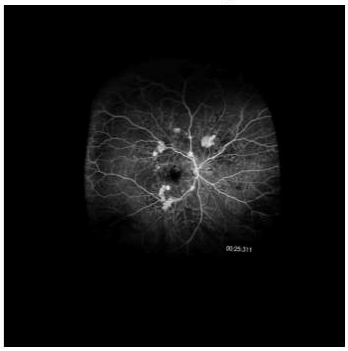
Fixed Image



Moving Image



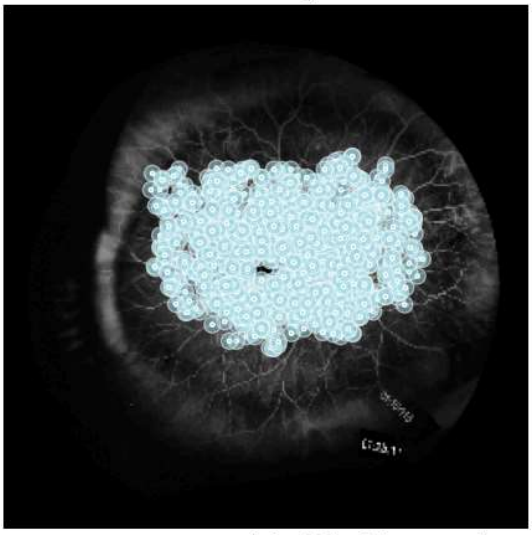
Deformed Image



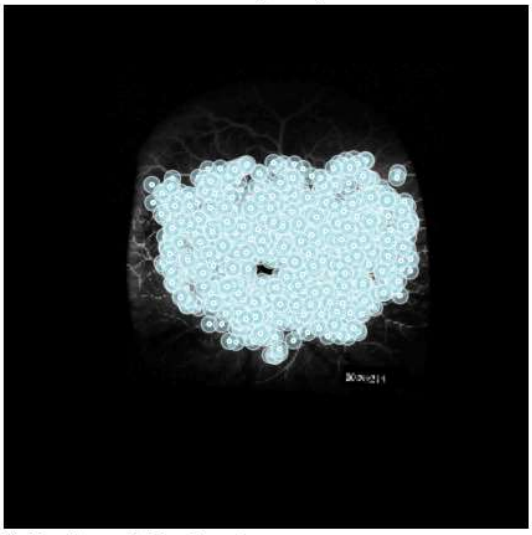
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

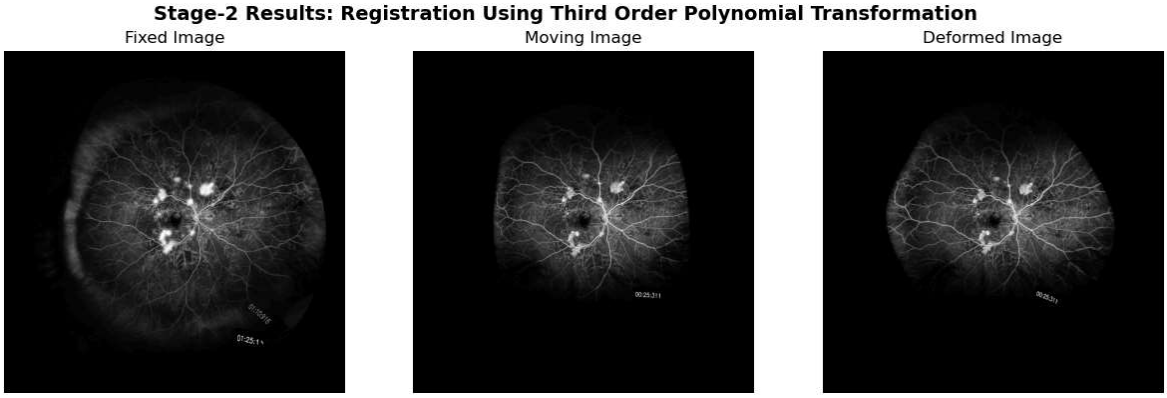
Fixed Image

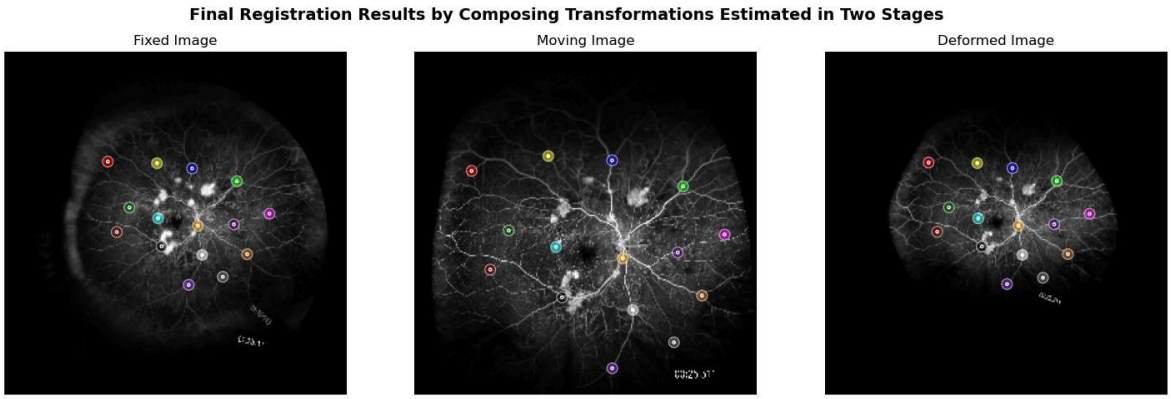


Moving Image

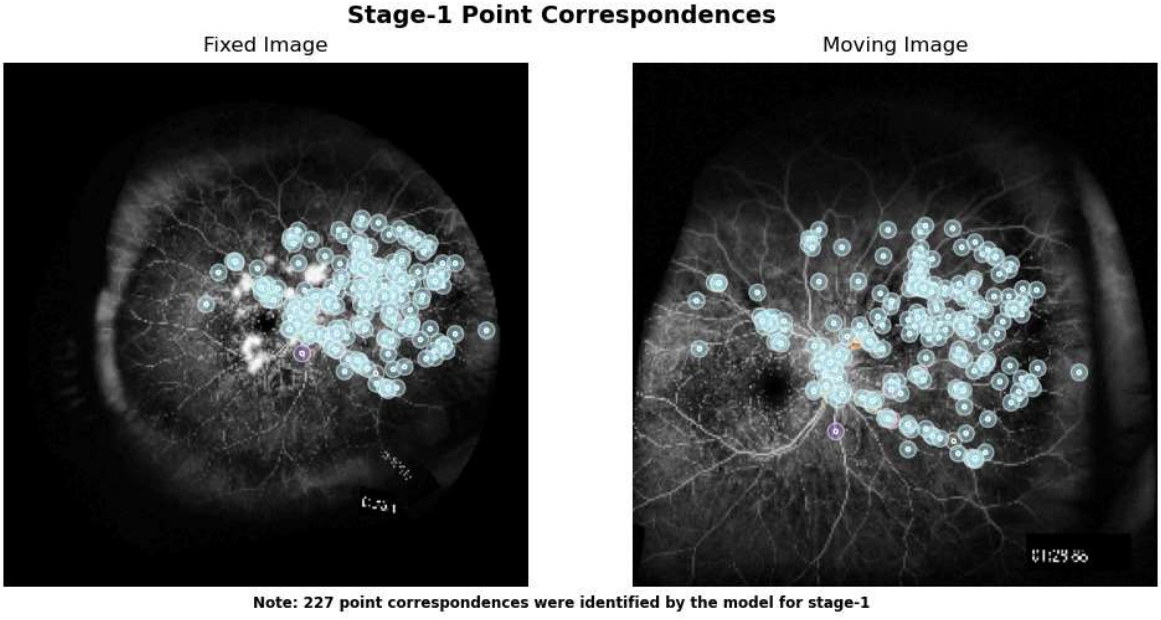


Note: 1181 point correspondences were identified by the model for stage-2



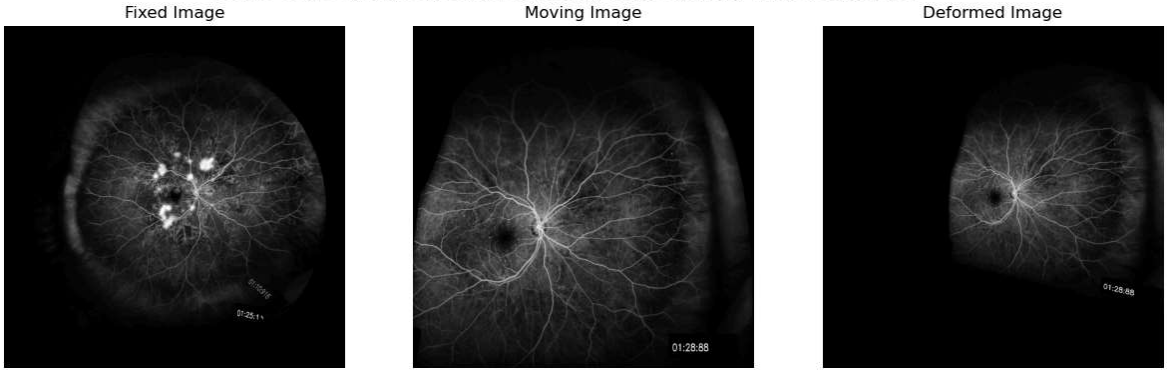



```
Mean Landmark Error for Case 7 Before Registration is 666.06809535934 pixels
Mean Landmark Error for Case 7 After Registration is 19.94079277437867 pixels
Case 8
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_3/Montag
e/Montage_Subject_3.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_3/FA/Raw_FA_2_Subject_3.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



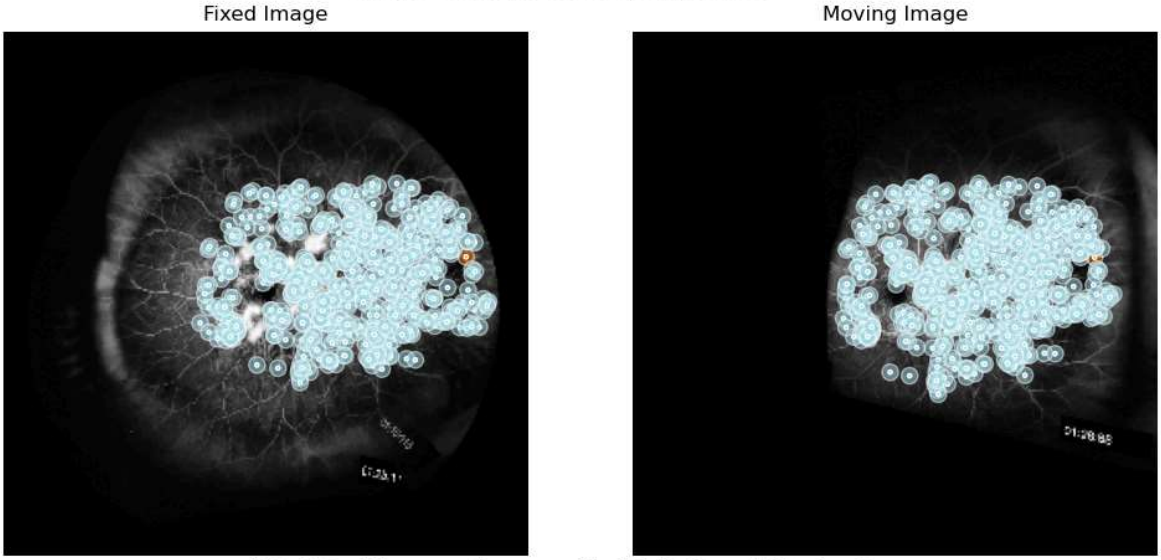

```
Homography Matrix:
[[ 3.45128136e-01  5.29852018e-02  3.71510261e+02]
 [-1.40561930e-01  6.41307887e-01  1.31068718e+02]
 [-3.73971151e-04  1.16199174e-04  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation



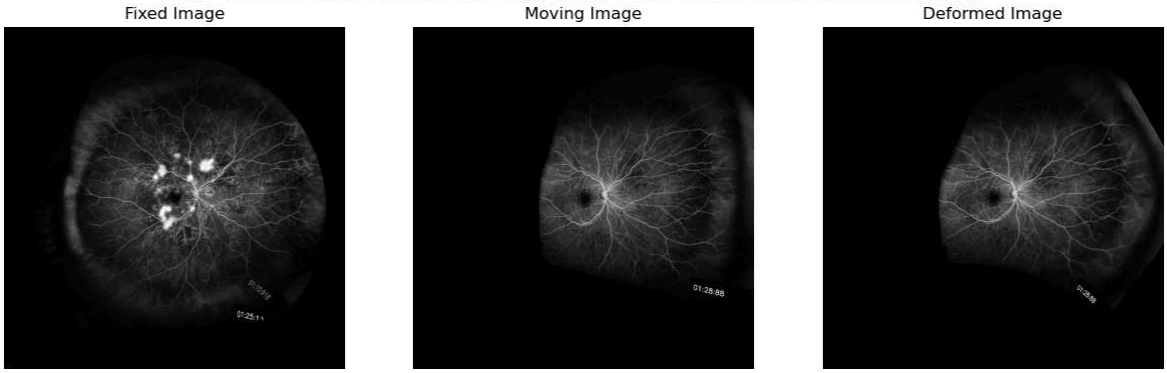

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences



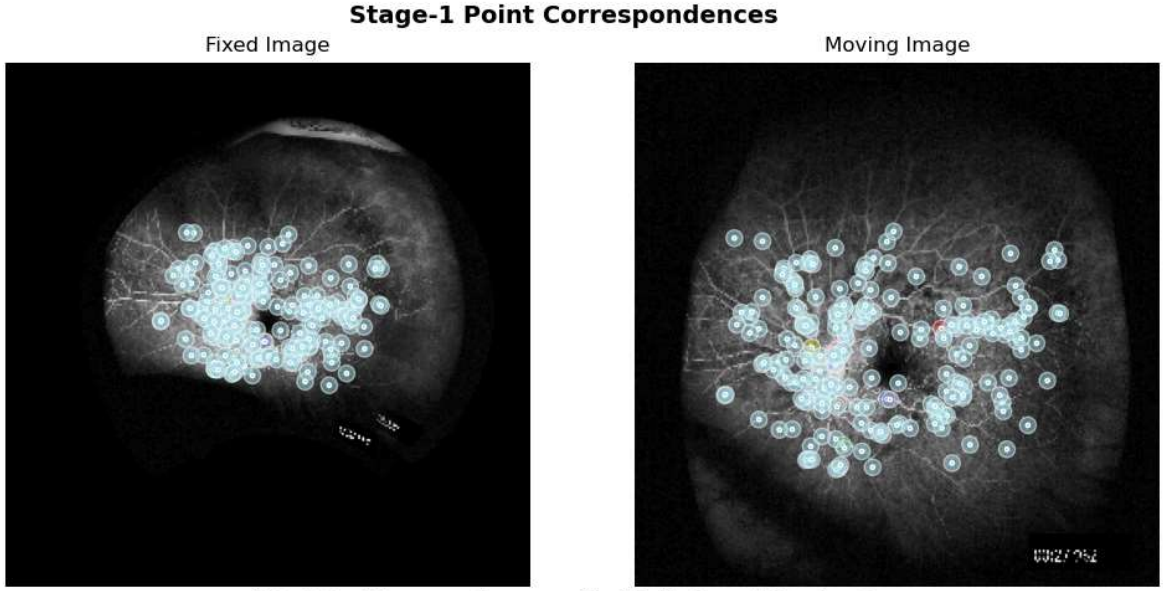
Note: 603 point correspondences were identified by the model for stage-2

Stage-2 Results: Registration Using Third Order Polynomial Transformation



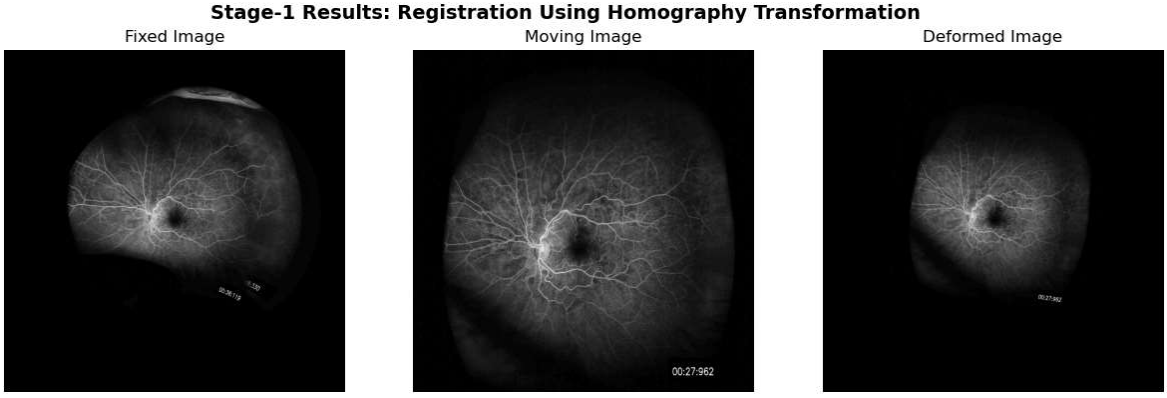
Final Registration Results by Composing Transformations Estimated in Two Stages

```
Mean Landmark Error for Case 8 Before Registration is 987.7230206941801 pixels
Mean Landmark Error for Case 8 After Registration is 11.764332809221116 pixels
Case 9
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_4/Montag
e/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_4/FA/Raw_FA_1_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

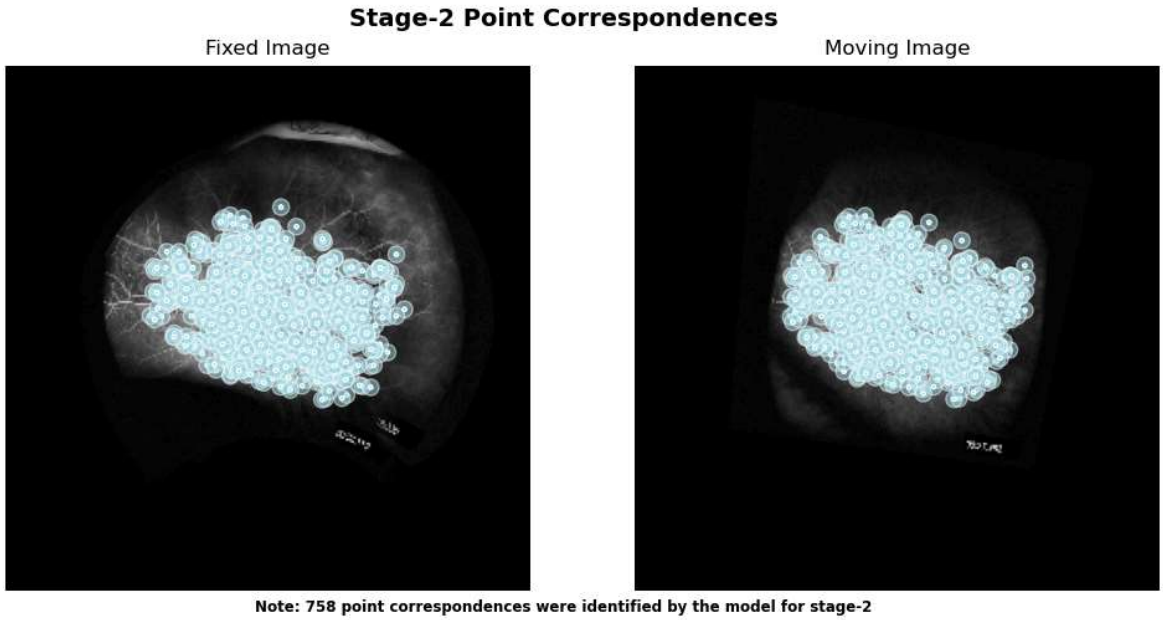


Note: 221 point correspondences were identified by the model for stage-1

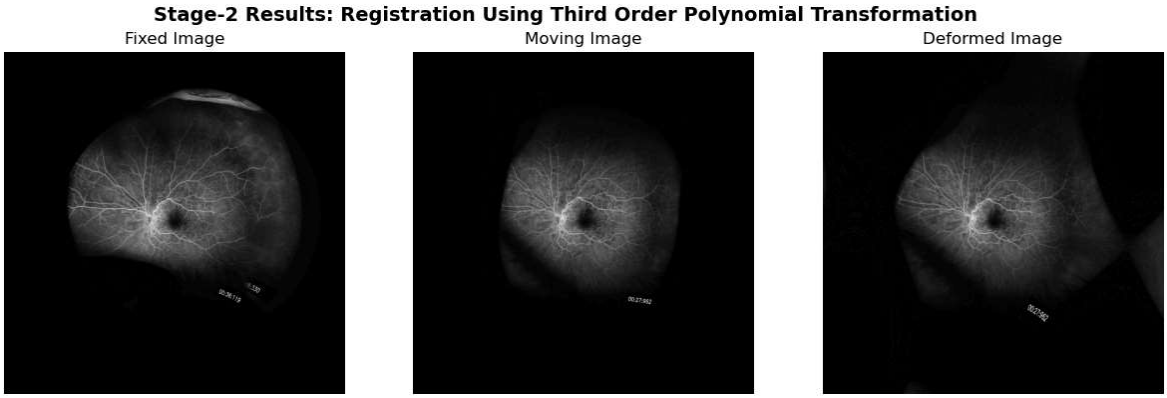
```
Homography Matrix:
[[ 7.10642871e-01 -2.78103223e-02  2.36098502e+02]
 [ 1.47714757e-01  7.22010210e-01  6.14857378e+01]
 [ 7.29534915e-05  1.28044048e-04  1.00000000e+00]]
```

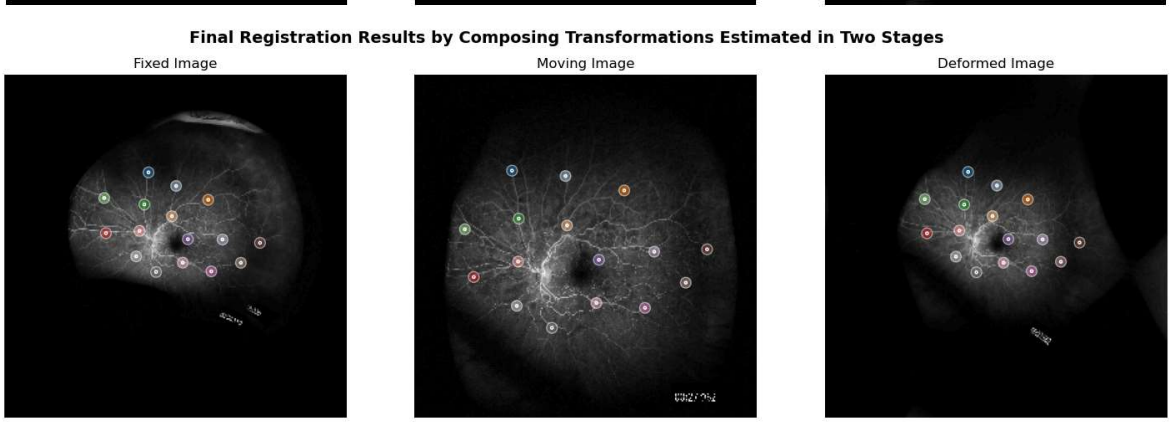



```
Maximum attempts reached, unable to find sufficient points with the specified cri
teria.
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 758 point correspondences were identified by the model for stage-2

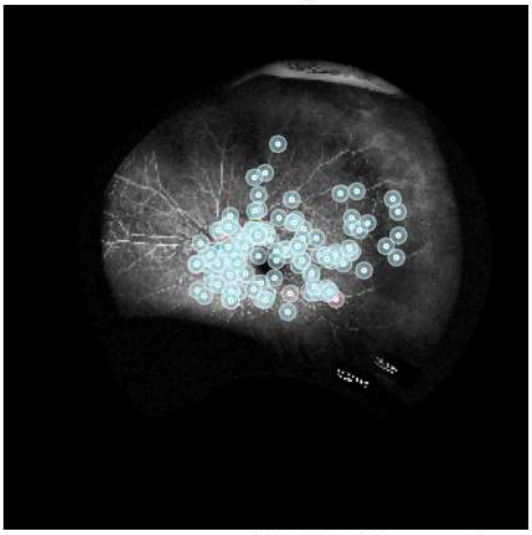




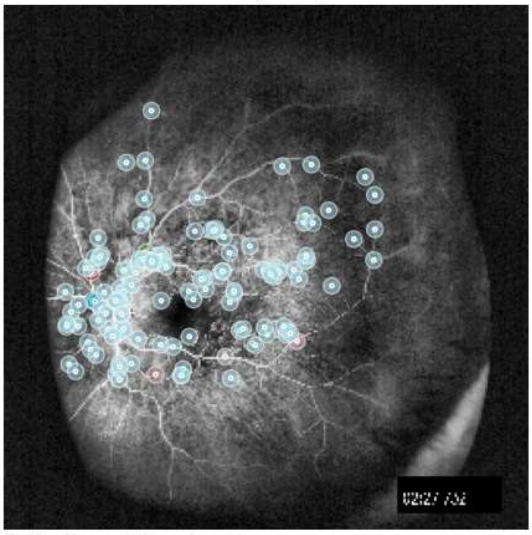

```
Mean Landmark Error for Case 9 Before Registration is 696.7319535892544 pixels
Mean Landmark Error for Case 9 After Registration is 13.073068079260198 pixels
Case 10
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_4/Montag
e/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_4/FA/Raw_FA_2_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image



Note: 139 point correspondences were identified by the model for stage-1

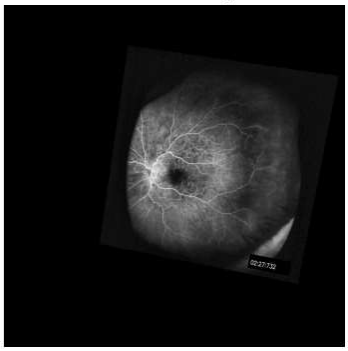
```
Homography Matrix:
[[ 5.64307502e-01 -6.96105647e-02  3.72079358e+02]
 [ 7.86509895e-02  6.15316632e-01  1.20433929e+02]
 [-5.25361358e-05  4.78357495e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

Fixed Image

Moving Image

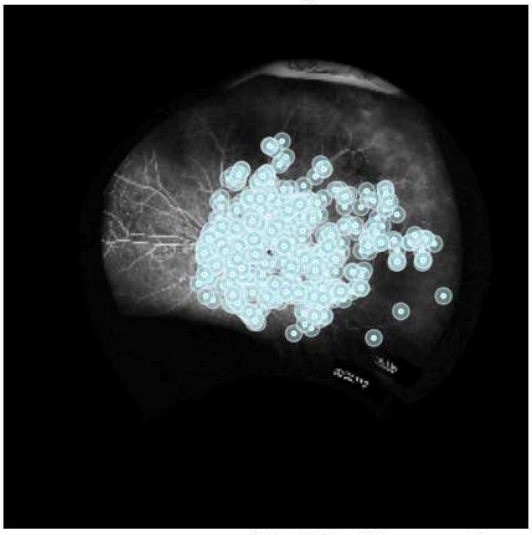
Deformed Image



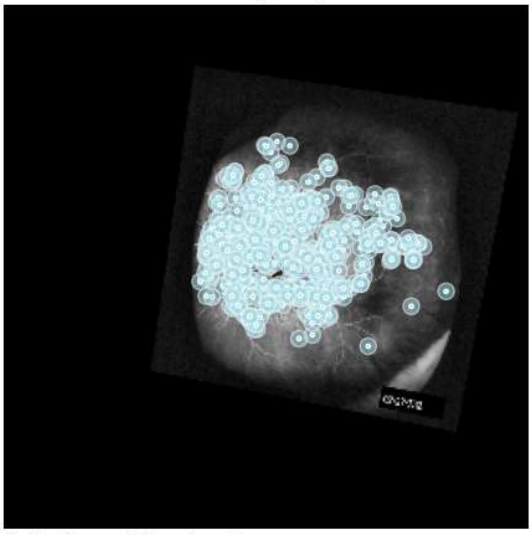
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

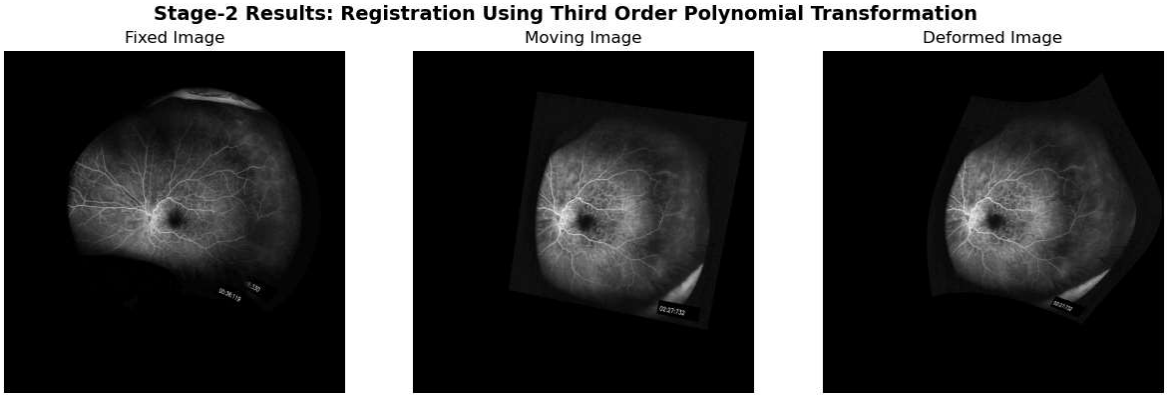
Fixed Image

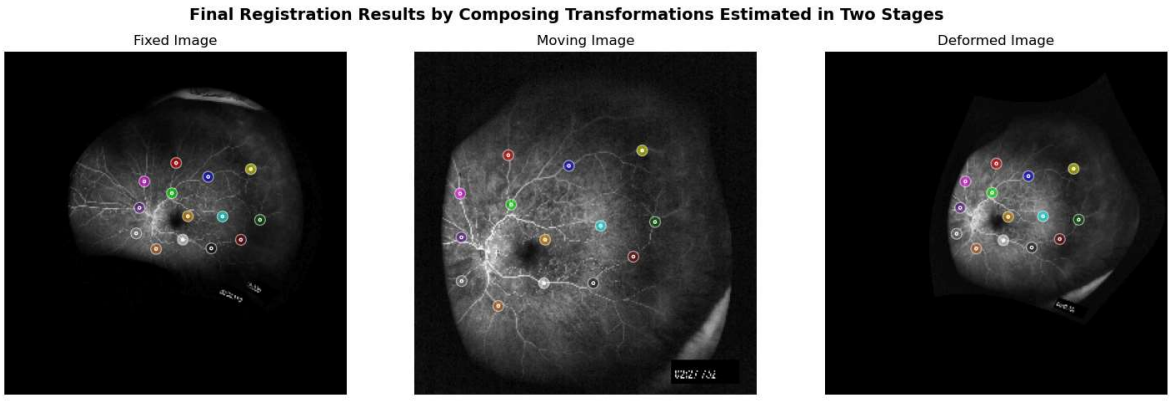


Moving Image

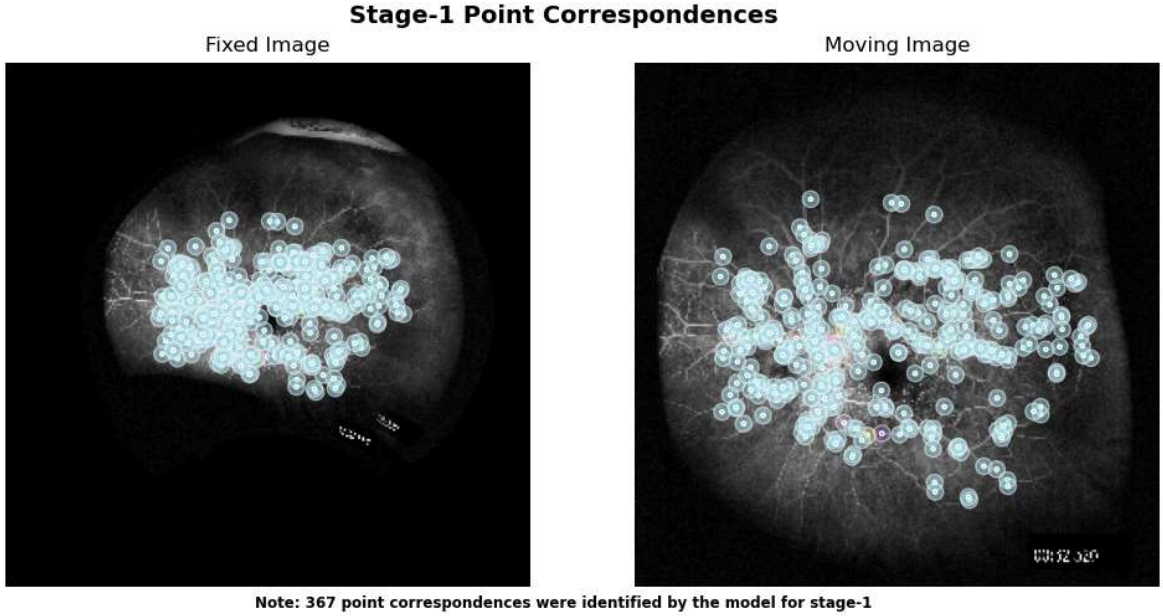


Note: 533 point correspondences were identified by the model for stage-2





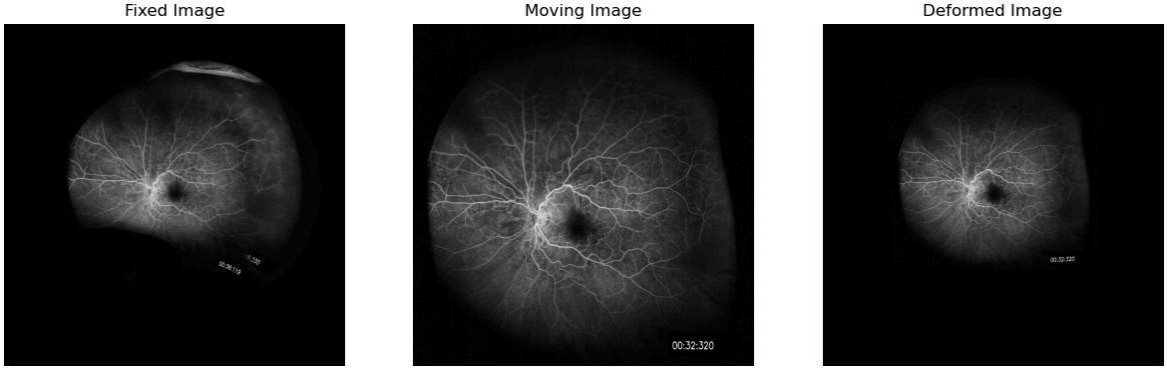

```
Mean Landmark Error for Case 10 Before Registration is 980.0529717789981 pixels
Mean Landmark Error for Case 10 After Registration is 10.480616100582798 pixels
Case 11
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_4/Montag
e/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_4/FA/Raw_FA_3_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```




```
Homography Matrix:
[[6.94839675e-01 4.10184991e-02 1.86583986e+02]
 [3.90762148e-03 6.83031000e-01 1.32761576e+02]
 [5.29213644e-05 8.50532938e-05 1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

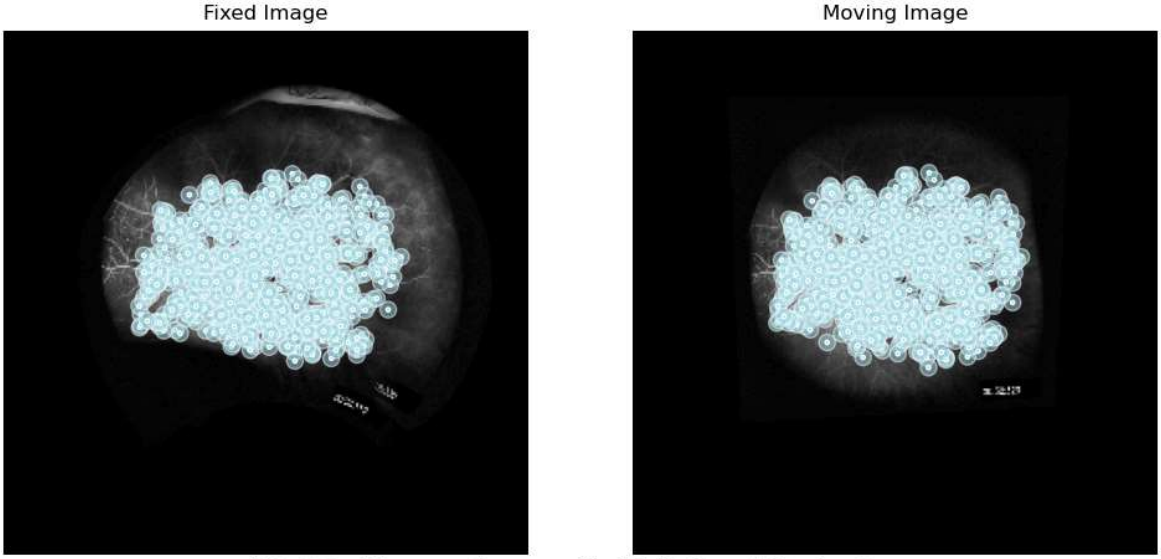
Fixed Image | Moving Image | Deformed Image



```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

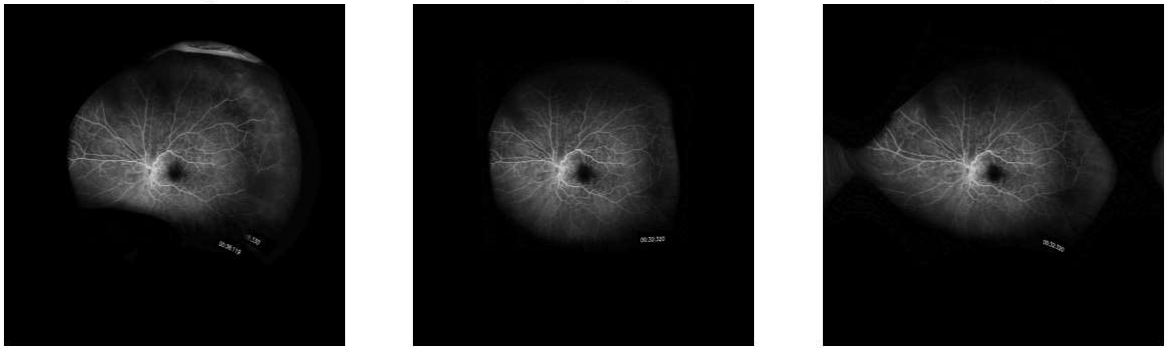
Fixed Image | Moving Image



Note: 829 point correspondences were identified by the model for stage-2

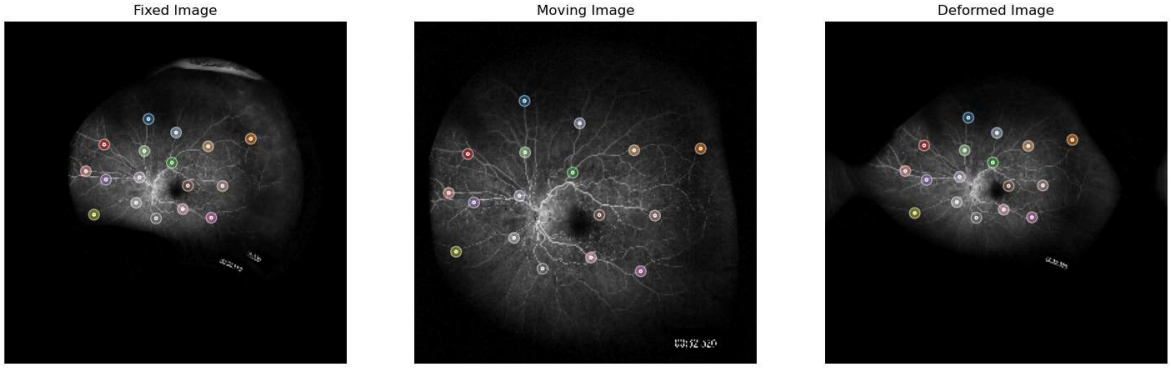
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image

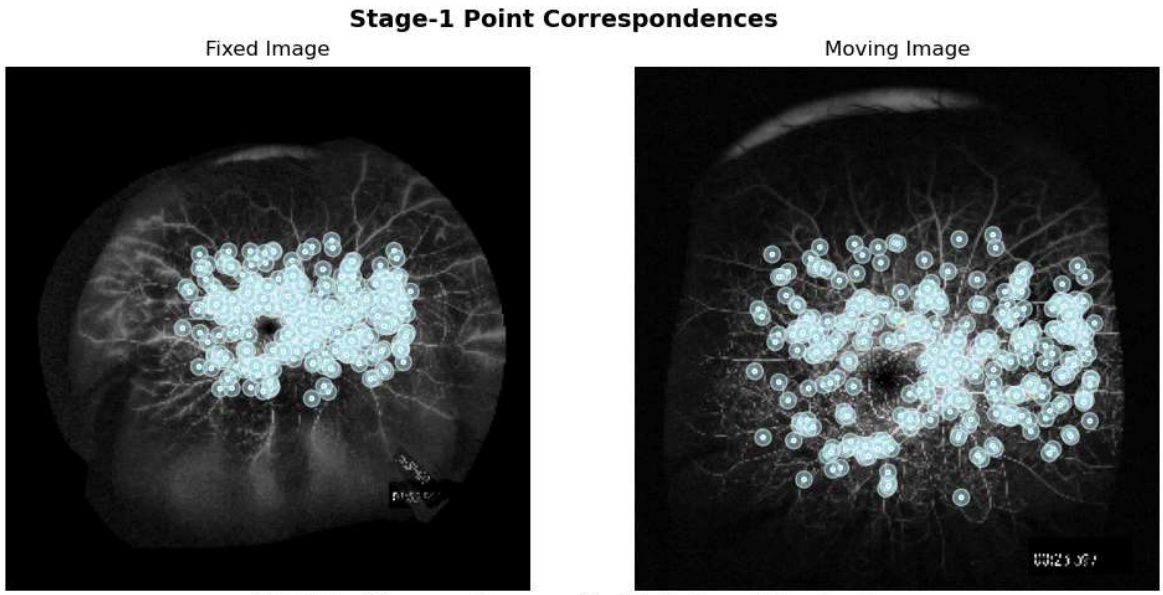


Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

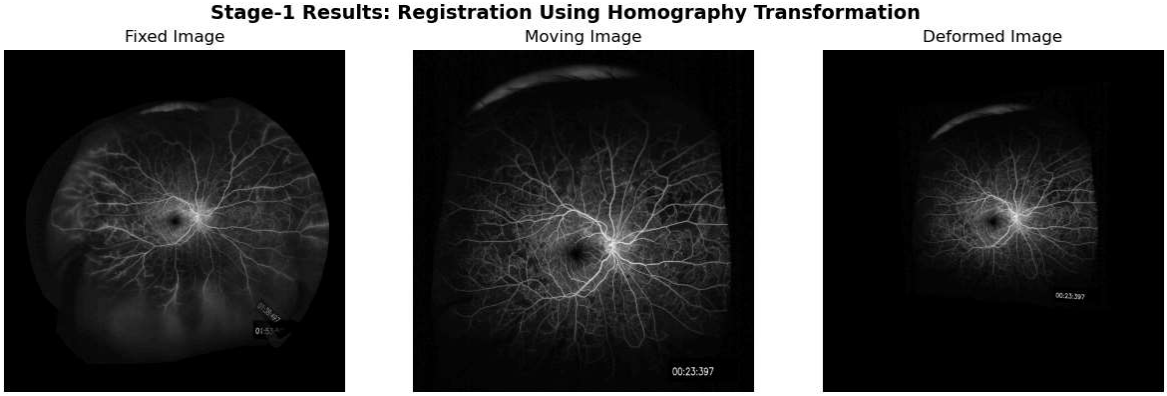


```
Mean Landmark Error for Case 11 Before Registration is 687.6665246743283 pixels
Mean Landmark Error for Case 11 After Registration is 14.879160687317492 pixels
Case 12
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_5/Montag
e/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_5/FA/Raw_FA_1_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

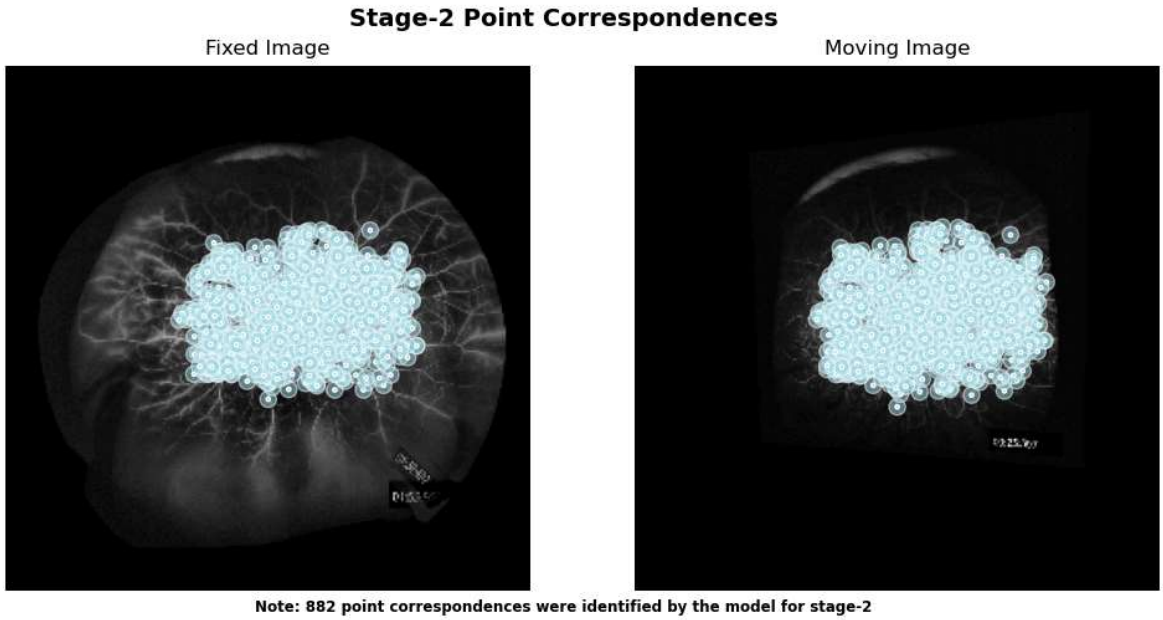


Note: 401 point correspondences were identified by the model for stage-1

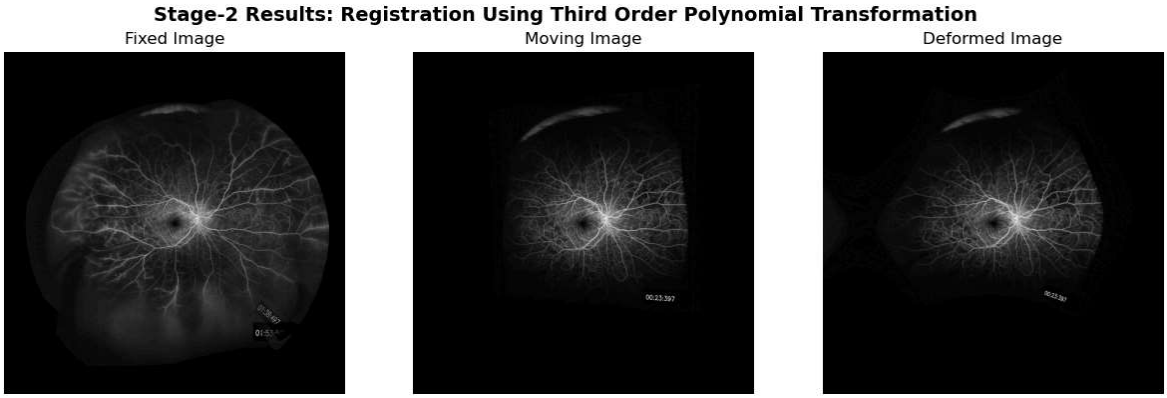
```
Homography Matrix:
[[ 4.78276624e-01  3.53078908e-02  2.23271845e+02]
 [-9.74075119e-02  5.86588553e-01  1.72097548e+02]
 [-1.91552990e-04  5.18675606e-05  1.00000000e+00]]
```

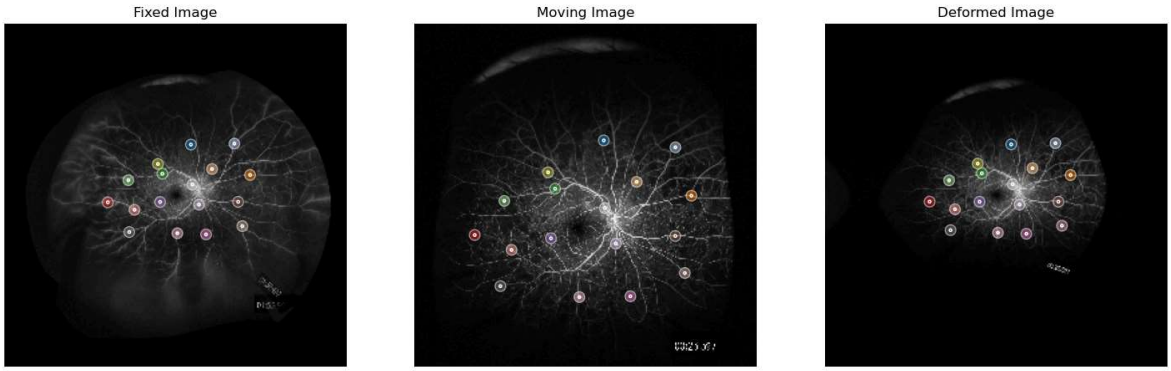



```
Maximum attempts reached, unable to find sufficient points with the specified cri
teria.
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 882 point correspondences were identified by the model for stage-2

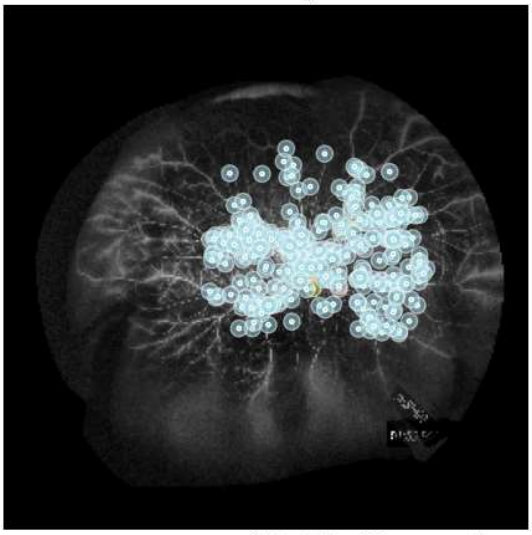




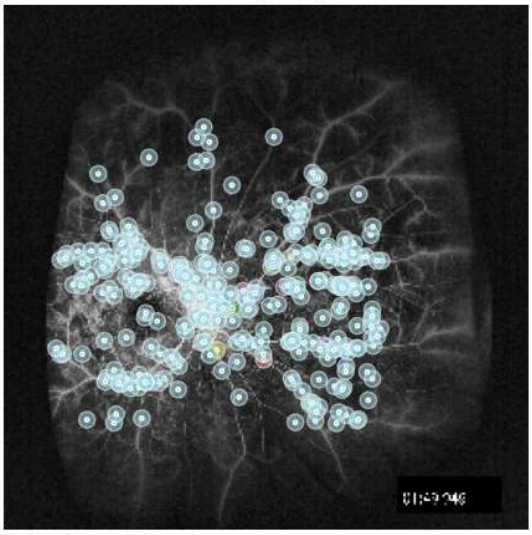

```
Mean Landmark Error for Case 12 Before Registration is 664.0966369010016 pixels
Mean Landmark Error for Case 12 After Registration is 9.373752666050184 pixels
Case 13
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_5/Montag
e/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_5/FA/Raw_FA_2_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image



Note: 306 point correspondences were identified by the model for stage-1

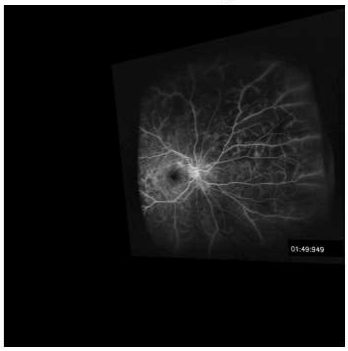
```
Homography Matrix:
[[ 4.31132923e-01  7.72512588e-02  3.22532505e+02]
 [-1.71506945e-01  5.97234648e-01  1.79163171e+02]
 [-2.55892120e-04  5.10126838e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

Fixed Image

Moving Image

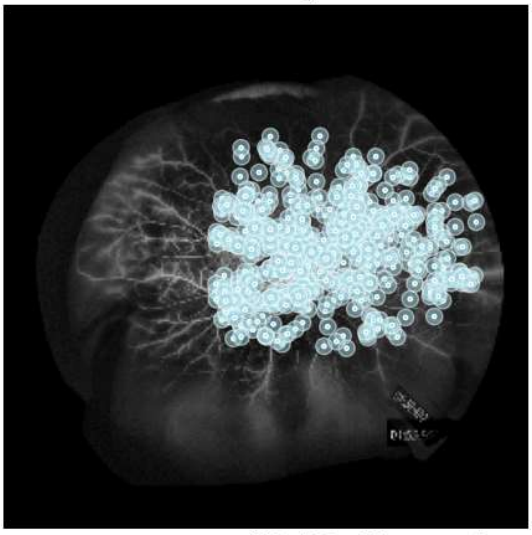
Deformed Image



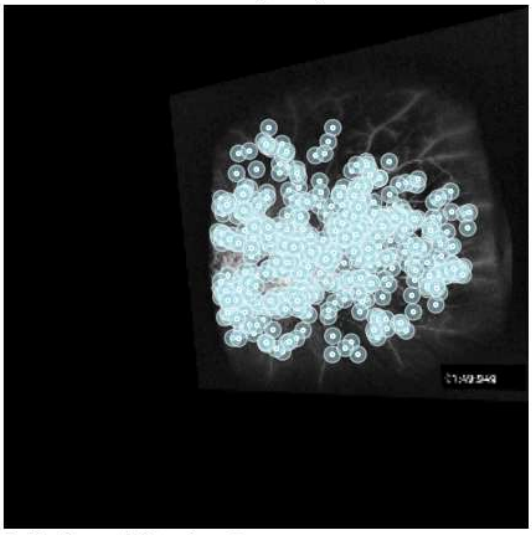
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

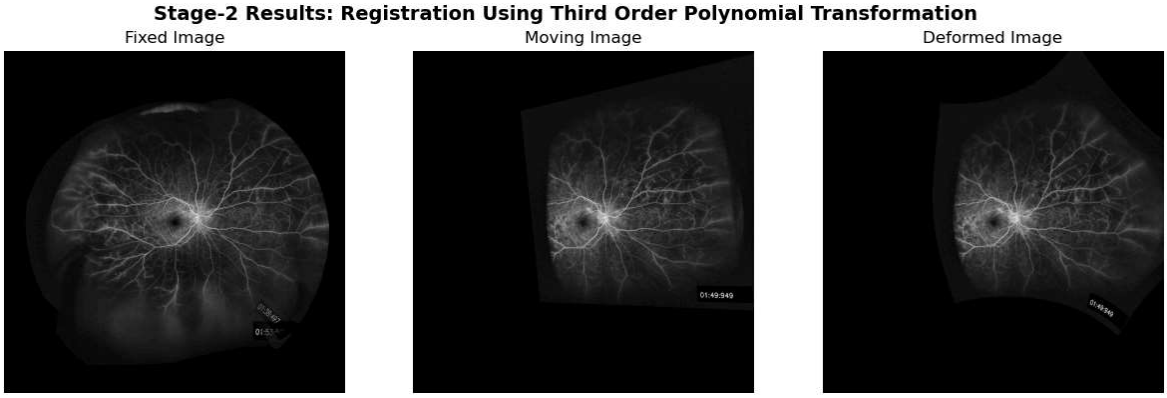
Fixed Image

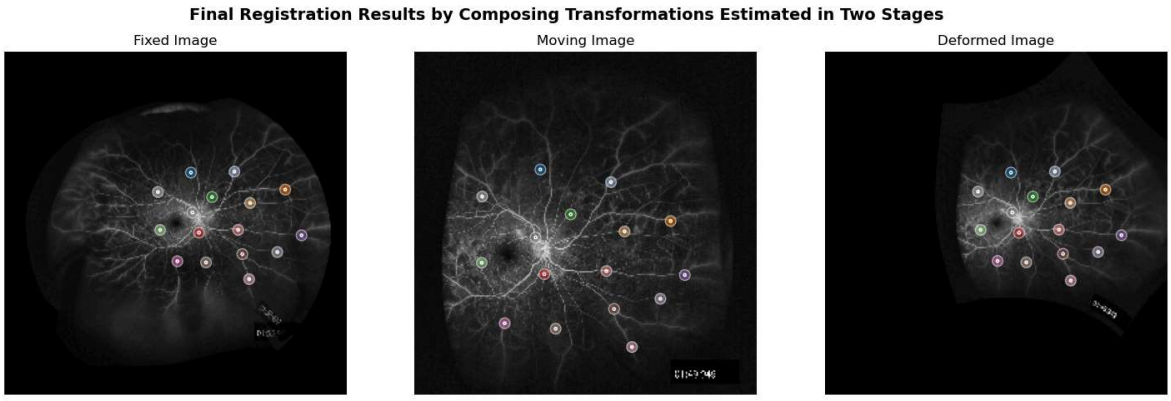


Moving Image

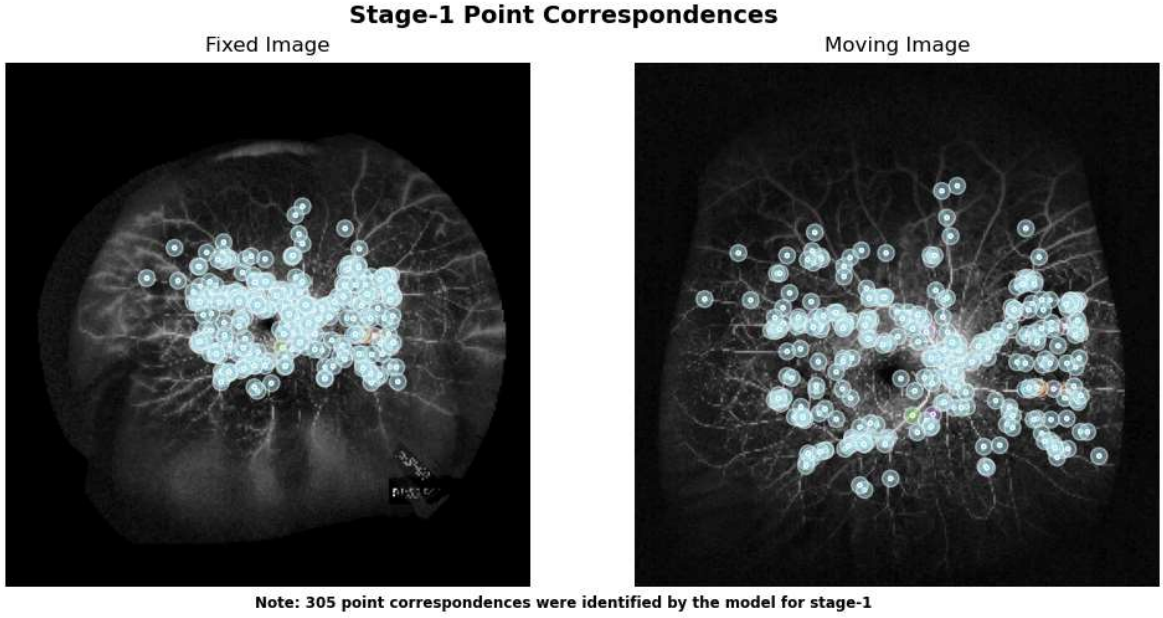


Note: 592 point correspondences were identified by the model for stage-2



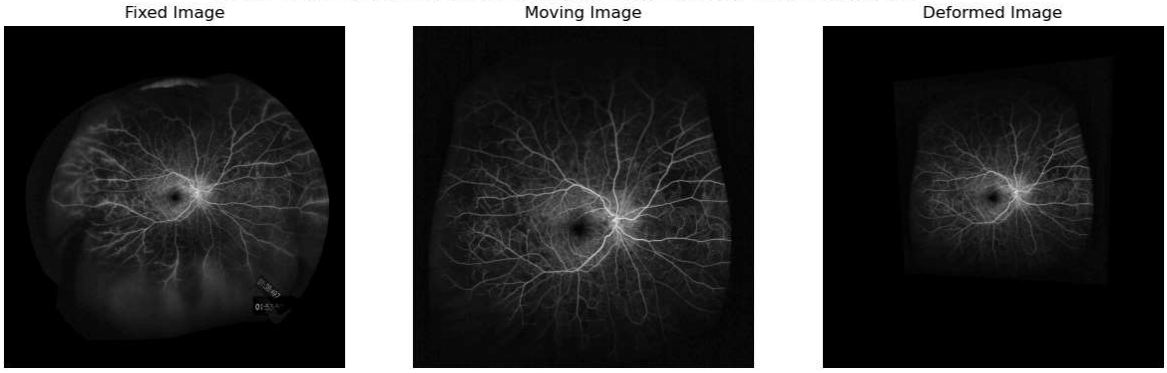



```
Mean Landmark Error for Case 13 Before Registration is 911.4162426368459 pixels
Mean Landmark Error for Case 13 After Registration is 8.865354830933484 pixels
Case 14
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Regis
tration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_5/Montag
e/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Tasks/Image_Regist
ration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Su
bject_5/FA/Raw_FA_3_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



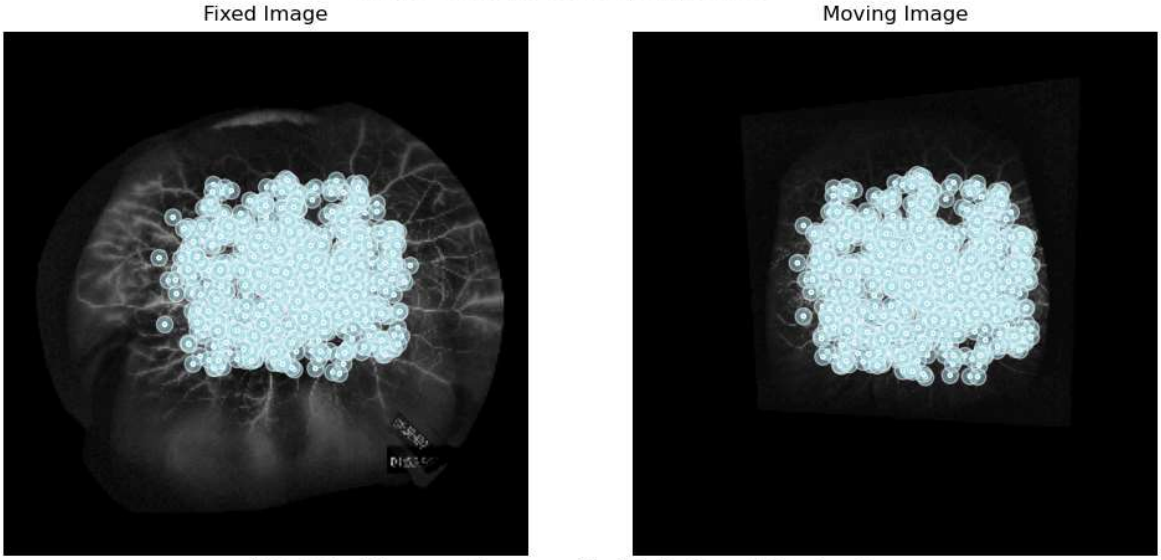

```
Homography Matrix:
[[ 5.05654758e-01  5.41996021e-02  2.09750402e+02]
 [-8.93800271e-02  6.19210604e-01  1.66476760e+02]
 [-1.62865576e-04  8.14973315e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation



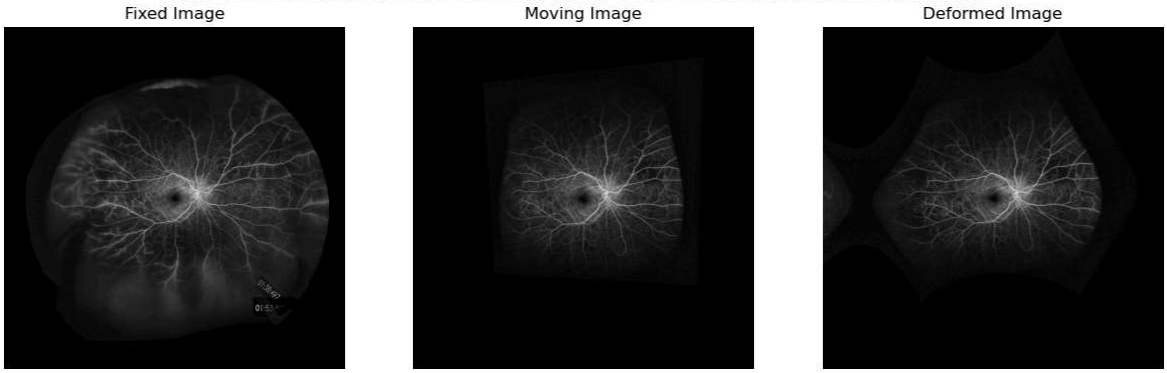

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

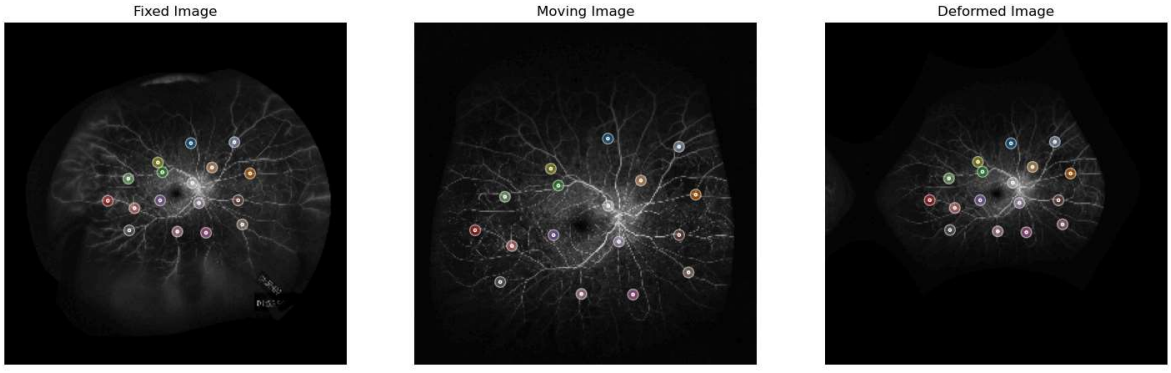


Note: 834 point correspondences were identified by the model for stage-2

Stage-2 Results: Registration Using Third Order Polynomial Transformation

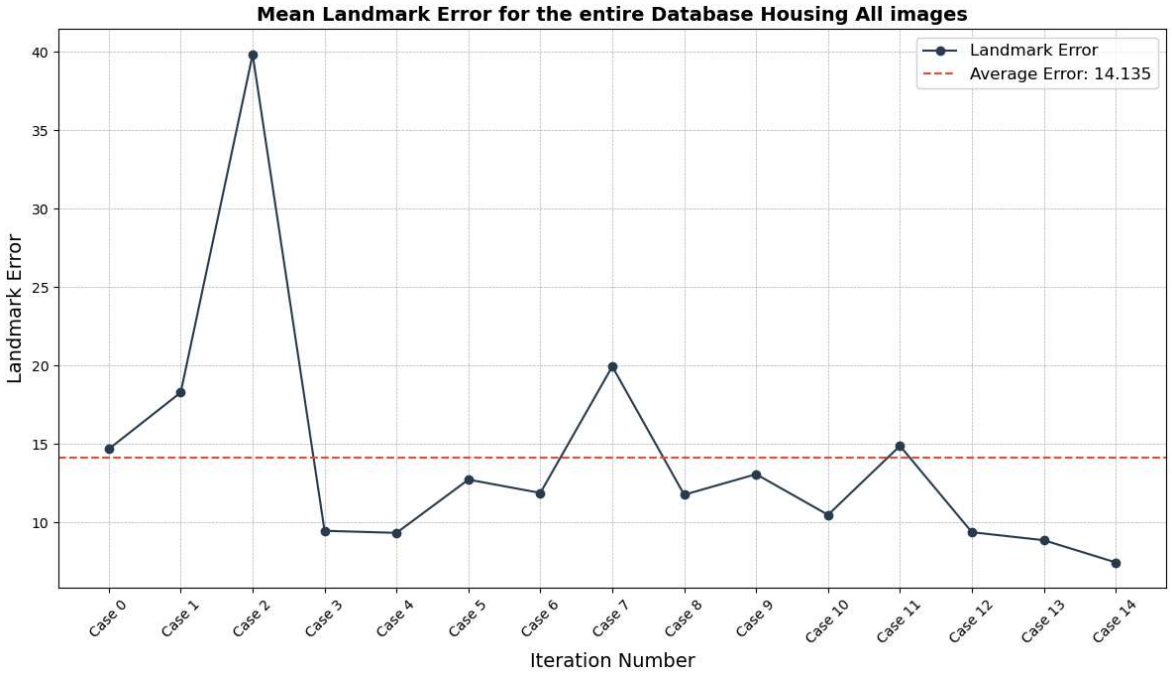


Final Registration Results by Composing Transformations Estimated in Two Stages

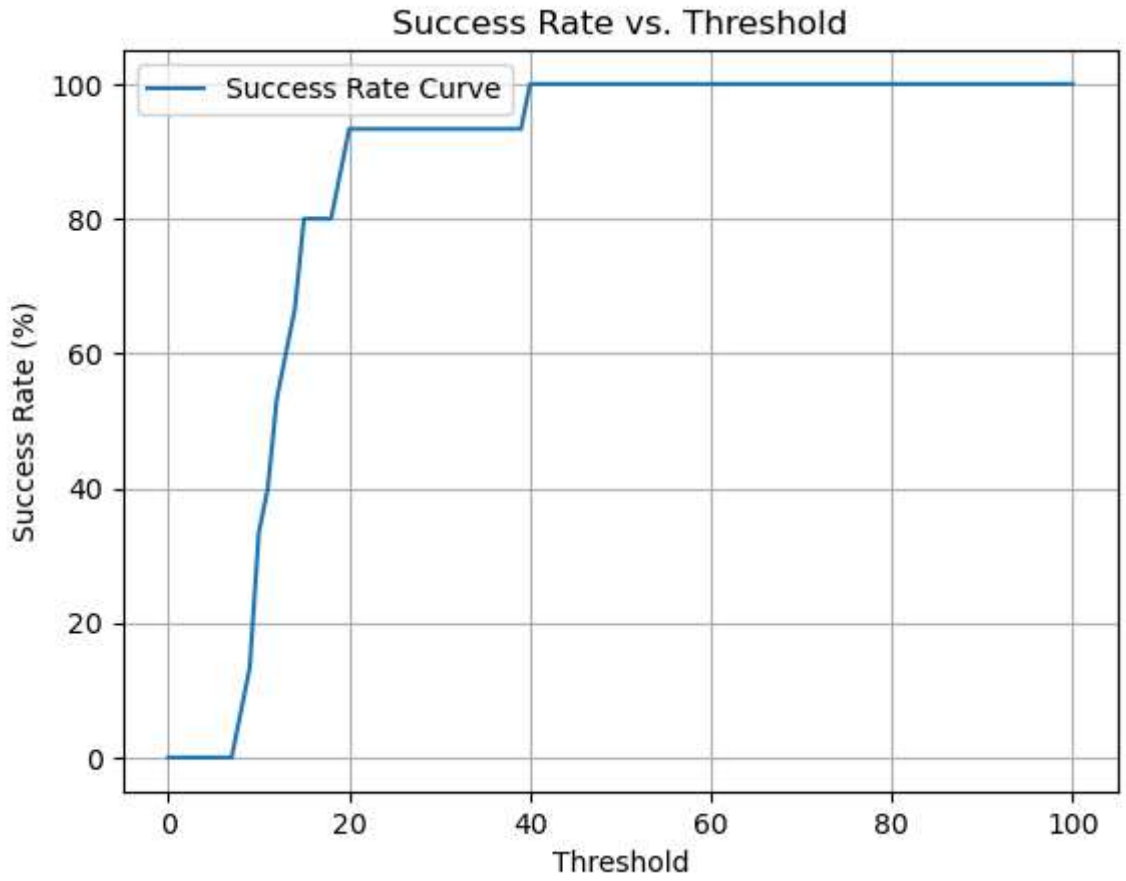



```
Mean Landmark Error for Case 14 Before Registration is 669.1965551074313 pixels
Mean Landmark Error for Case 14 After Registration is 7.447894656076012 pixels
```

In [20]:
```
plot_landmark_errors(landmark_errors,os.path.join(os.getcwd(),'FLoRI21_DataPort_
```



In [21]:
```
compute_plot_Flori21_AUC(landmark_errors)
```




```
AUC: 0.8646666666666666
```