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 14:00:53.122760: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimiz
ed to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, 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_Results'))
```

In [5]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Stage1')
os.makedirs(path,exist_ok=True)
```

In [6]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Stage2')
os.makedirs(path,exist_ok=True)
```

In [7]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Final_Registration_Results')
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 additional conditioning mechanisms
    such as encoder hidden states, attention mask, and cross-attention keyword arguments.
    """
    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, channel, height, width).
            timestep (torch.FloatTensor or float or int): Timesteps for each batch.
            up_ft_indices (list): List of upsampling indices.
            encoder_hidden_states (torch.FloatTensor): Encoder hidden states with shape (batch, sequence_length,
            class_labels (Optional[torch.Tensor], default=None): Class labels tensor.
            timestep_cond (Optional[torch.Tensor], default=None): Timestep condition tensor.
            attention_mask (Optional[torch.Tensor], default=None): Mask to avoid attention to certain positions.
            cross_attention_kwargs (Optional[dict], default=None): Keyword arguments passed along to the `AttnPro

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

        # By default samples have to be AT least a multiple of the overall upsampling factor.
        # The overall upsampling factor is equal to 2 ** (# num of upsampling layears).
        # However, the upsampling interpolation output size can be forced to fit any upsampling size
        # 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 `default_overall_up_factor`
        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 size.")
            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 timesteps as tensors if you can
            # 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.device)
        elif len(timesteps.shape) == 0:
            timesteps = timesteps[None].to(sample.device)

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

        t_emb = self.time_proj(timesteps)

        # timesteps does not contain any weights and will always return f32 tensors
        # but time_embedding might actually be running in fp16. so we need to cast here.
        # 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_embeds > 0")

            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_block.has_cross_attention:
```

```
                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, temb=emb)

            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_block.resnets)]

            # 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.has_cross_attention:
                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_samples, upsample_size=upsample_s
                )

            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 and U-Net based sampling.
    """
    @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]]] = None,
        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 prompts.
            generator (Optional[torch.Generator or list], default=None): Torch generator for random sampling.
            prompt_embeds (Optional[torch.FloatTensor], default=None): Precomputed prompt embeddings.
            callback (Optional[Callable], default=None): Callback function invoked during diffusion.
            callback_steps (int, default=1): Frequency of invoking the callback.
            cross_attention_kwargs (Optional[dict], default=None): Keyword arguments for cross-attention.

        Returns:
            dict: Dictionary containing output from U-Net.
        """
        device = self._execution_device
        latents = self.vae.encode(img_tensor).latent_dist.sample() * self.vae.config.scaling_factor
        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 input image, conditioned on a given p
    """
    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 diffusion model ID to be used for fea
        """
        unet = MyUNet2DConditionModel.from_pretrained(sd_id, subfolder="unet")
        onestep_pipe = OneStepSDPipeline.from_pretrained(sd_id, unet=unet, safety_checker=None)
        onestep_pipe.vae.decoder = None
        onestep_pipe.scheduler = DDIMScheduler.from_pretrained(sd_id, subfolder="scheduler")
        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, c, h, w].
            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 averaging.

        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, h, w
        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 identifying vital key feature correlati
    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 the specified shape.

        This function mimics numpy's `unravel_index` functionality, which is used to convert a flat index
        into a tuple of coordinate indices for an array of given shape. This is useful for finding the original
        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 derived.

        Returns:
        - tuple of ints: A tuple representing the coordinates of the index in an array of the specified shape.

        Note:
            This function operates under the assumption that indexing starts from 0, which is standard in Python.
        """
        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_range=1, stride=1):
        """
        Compute pooled and stacked feature maps.

        Parameters:
        - feature_map (torch.Tensor): Input feature map.
        - hierarchy_range (int, optional): Depth of hierarchical pooling. Defaults to 3.
        - 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 // 2, count_include_pad=False)
            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, feature_map2):
        """
```

```
        Computes 2D correlation maps between selected points in one feature map and another feature map.

        This method takes two feature maps and a list of points. It extracts features from the first feature map
        at specified points, normalizes them, and then computes a batched 2D correlation with the second feature
        The output is a set of correlation maps, each corresponding to a point in `pts_list`, showing how that po
        feature vector correlates across the spatial dimensions of the second feature map.

        Parameters:
        - pts_list (list of tuples): List of points (y, x) for which the correlation map is to be computed.
        - feature_map1 (torch.Tensor): The first feature map tensor of shape (1, C, H1, W1) where C is the number
        - feature_map2 (torch.Tensor): The second feature map tensor of shape (1, C, H2, W2) where C is the numbe
                                       and H2, W2 do not necessarily need to be equal to H1, W1.

        Returns:
        - torch.Tensor: A tensor of shape (NumPoints, H2, W2) where each slice corresponds to the correlation map
                        for each point in `pts_list`.
        Notes:
            The function assumes that the first dimension of feature_map1 and feature_map2 is 1 (batch size of 1)
            This method uses batch matrix multiplication and vector normalization for efficient computation.
            Running this method on a GPU is recommended due to its computational and memory intensity.
        """
        # 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[:, 1]].transpose(0, 1)  # Shape

        # 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_feature_map2)

        # Reshape the correlation maps to the desired output shape (NumPoints, H, W)
        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, feature_map2): # heirachy range - hpa
        """
        Compute the maximum locations in the batched correlation maps between two feature maps.

        Parameters:
        - pts_list (list of tuples): List of points for which the correlation maps were computed.
        - 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(feature_map1, hierarchy_range=1)
        enhanced_feature_map2 = self.compute_pooled_and_combining_feature_maps(feature_map2, hierarchy_range=1)
        # Compute the batched correlation maps
        batched_correlation_maps = self.compute_batched_2d_correlation_maps(pts_list, enhanced_feature_map1, enha

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

        # Find the maximum values and their locations along the last two dimensions for each map
        max_values, max_indices_flat = torch.max(batched_correlation_maps.view(len(pts_list), -1), dim=-1)

        x, y = zip(*[self.unravel_index(idx.item(), (H2, W2)) for idx in max_indices_flat.view(-1)])
        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='bilinear')(src_ft)  # (1, C, H, W)
            gc.collect()
            torch.cuda.empty_cache()
            trg_ft = nn.Upsample(size=(self.img_size, self.img_size), mode='bilinear')(ft[1:])  # (1, C, H, W)
        return src_ft,trg_ft


    def feature_maps(self,feature_map1,feature_map2,iccl):
        """
        Processes feature maps to extract points that meet the inverse consistency criteria between two images.

        This method computes the maximum locations of correlation between feature maps of two images and
        checks for inverse consistency between the mapped points. It filters these points based on the
        specified inverse consistency criteria limit (iccl), keeping only those pairs where the
        distance between the original point and its double-mapped location is within the threshold.

        Parameters:
        - feature_map1 (torch.Tensor): The first feature map, used as the base for initial correlations.
        - feature_map2 (torch.Tensor): The second feature map, used for reverse correlations to check consistency
        - iccl (float): The maximum allowed distance (inverse consistency criteria limit) for a point and
                        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 meet the inverse consistency crite
        - rmaxs (list of floats): The maximum correlation values at these points.
        - rspts (list of tuples): The corresponding points in the second feature map that have the highest correl
                                  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_locations(pts,feature_map1,feature_map2)
        x_prime_y_prime = max_indices_ST
        max_indices_TS, max_values_TS = self.compute_correlation_map_max_locations(max_indices_ST,feature_map2,fe
        x_prime_prime_y_prime_prime = max_indices_TS
        for i, (pt, max_idx) in enumerate(zip(self.pts, x_prime_prime_y_prime_prime)):
            # Calculate the distance between the point and the max correlation index
            if np.sqrt((pt[1] - max_idx.cpu()[0]) ** 2 + (pt[0] - max_idx.cpu()[1]) ** 2) <=iccl: ### inverse co
                pnts.append((int(pt[0]), int(pt[1])))
                rmaxs.append(max_values_ST[i].cpu().item())  # Assuming max_values_ST is a tensor with correspond
                rspts.append((x_prime_y_prime[i][1].cpu().item(), x_prime_y_prime[i][0].cpu().item()))  # Assumi
        return pnts, rmaxs, rspts
```

In [10]:

```
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 boundaries.

    Returns:
    - tuple: upper, lower, left, and right boundaries of the image region with intensities above mean_intensity.
    """
    # Compute the upper, lower, left, and right boundary
    upper_boundary = next((i for i, row in enumerate(image) if np.mean(row) > mean_intensity), 0)
    lower_boundary = next((i for i, row in enumerate(image[::-1]) if np.mean(row) > mean_intensity), 0)

    left_boundary = next((i for i, col in enumerate(image.T) if np.mean(col) > mean_intensity), 0)
    right_boundary = next((i for i, col in enumerate(image.T[::-1]) if np.mean(col) > mean_intensity), 0)

    return upper_boundary, image.shape[0]-lower_boundary, left_boundary, image.shape[1]-right_boundary
```

```python
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 constraints on distance, boundary, and coll

    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 to 256.
    - max_dist (int): Minimum distance between selected keypoints. Defaults to 25.

    Returns:
    - list: List of selected keypoints (cv2.KeyPoint objects).
    - list: List of keypoints' positions in the form (x, y).
    """
    # Load image
    image1 = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image1 = cv2.resize(image1, (img_shape, img_shape))
    clahe = cv2.createCLAHE(clipLimit=1.0, tileGridSize=(8, 8))
    image = clahe.apply(image1)

    # 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 threshold (not black)
            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_dist for kp in selected_keypoints)
                    selected_keypoints.append(keypoint)

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

    # Draw keypoints on the color image
    image_with_keypoints = cv2.drawKeypoints(image1, selected_keypoints, None)
    return selected_keypoints, [kp.pt for kp in selected_keypoints]

def select_random_points(img, num_points=100, img_size=1200,offset=0.01,window_size = 51,max_attempts_per_point=5
    """
    Selects a specified number of random points from an image, ensuring that each point is centered in a region
    meeting a defined intensity threshold within the image. The image is resized to a specified size, and points
    are chosen randomly, with each potential point undergoing validation against criteria before being accepted.

    Parameters:
    - img (str): Path to the image file.
    - num_points (int, optional): The number of random points to select. Defaults to 100.
    - img_size (int, optional): The size to which the image is resized (assumed square). Defaults to 1200.
    - offset (float, optional): Proportional offset to exclude points near the edges, represented as a fraction o
                                the image dimensions. Defaults to 0.01.
    - window_size (int, optional): Size of the square window used to check pixel intensity around each point.
```

```
                                    Defaults to 51.
    - max_attempts_per_point (int, optional): The maximum number of attempts allowed to find a suitable point
                                            that meets the criteria. Defaults to 50.

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

    Notes:
        The function converts the image to grayscale and resizes it to img_size x img_size. It avoids selecting
        points near the image boundary by applying a boundary offset calculated from the 'offset' parameter.
        Each point must be centered in a window (defined by 'window_size') where all pixels have an intensity
        greater than or equal to 5. If the function fails to find a suitable point after 'max_attempts_per_point
        for any location, it stops and returns the points found up to that moment.
    """

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

    while len(pts) < num_points:
        attempts = 0
        while attempts < max_attempts_per_point:
            x = random.randint(boundary_offset + window_offset, h - boundary_offset - window_offset - 1)
            y = random.randint(boundary_offset + window_offset, w - boundary_offset - window_offset - 1)

            # 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 with the specified criteria.")
            break  # Break outer loop if max attempts is reached without finding a point

    return pts
```

In [11]:

```
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 image contrast,
    and then converts it back to RGB. It uses OpenCV for the CLAHE operation and PIL for image
    conversions.

    Parameters:
    - imag (np.array): The input image array. Expected to be in format suitable for OpenCV.
    - clip (float): The clipping limit for the CLAHE algorithm, which controls the contrast limit.
                  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 parameter may affect
          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 for the dataset titled Flori21.

    Parameters:
```

```
    - landmark_errors: List of landmark errors including outliers.
    """
    landmark_errors_sorted = sorted(landmark_errors) # includes all outliers as well
    # 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 <= threshold])
        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 outliers_plot_condition(landmark_errors,cond):
    """
    Filters out specific outlier values from a list of landmark errors based on a condition.

    This function examines each error in the list of landmark errors and removes specific outlier values,
    in this case, the value 10000, if the condition specified by the 'cond' parameter is True. If 'cond'
    is False, the list is returned unchanged. This functionality can be useful for cleaning or preparing
    data before further analysis or visualization.

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

    Returns:
    - list of int or float: A list of landmark errors with specified outliers removed if the condition is met.
    """
    if cond:
        landmark_errors =[x for x in landmark_errors if x!=10000]
    return landmark_errors


def plot_landmark_errors(landmark_errors,rpth,chr='All',disable_outliers=False):
    """
    Plots a graph of landmark errors over successive iterations to provide a visual analysis of registration accu
    across samples. This function is designed to help in the assessment of registration processes in image proces
    or computer vision tasks by plotting each landmark error against its iteration number. It also calculates and
    displays the average landmark error across all iterations.

    Parameters:
    - landmark_errors (list of float): A list containing numerical errors for each landmark across multiple itera
                                       Outliers (e.g., errors set to 10000) are automatically excluded from the plo
    - rpth (str): Path where the resulting plot image will be saved.
    - chr (str, optional): Characteristic or description to include in the plot title, indicating the dataset or
                        used. Defaults to 'All'.
    - disable_outliers (bool, optional): If set to True, disables the automatic exclusion of outlier values in th
                                         data. Defaults to False.

    Returns:
    - None: This function does not return any value but saves the plot to the specified path and displays it.

    Notes:
        This plot is useful for tracking improvements or deteriorations in landmark detection algorithms over tim
        It automatically filters out error values set to 10000, considering them as outliers, unless disable_outl
        is set to True.
        The function saves the plot in the directory specified by `rpth` and names it 'Landmark_Error_Plot.png'.
    """
    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', label="Landmark Error")
    plt.axhline(y=avg_error, color='#E74C3C', linestyle='--', label=f"Average Error: {avg_error:.3f}")
    plt.title("Mean Landmark Error for the entire Database Housing {} images".format(chr), fontsize=14, fontweigh
    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,snum,disp_size=256):
    """
    Displays and compares point correspondences between two images using given landmarks.

    This function visualizes two images side-by-side with their respective landmarks. Each pair of corresponding
    landmarks across the images is marked with the same color for easy identification of correspondences. The fur
    is designed to handle visualization for studies involving image registration or similar tasks where landmark
    matching is crucial.

    Parameters:
    - images (list of str): File paths to the two images (source and target images).
    - img_size (int): The size to which images should be resized, specified as width and height (assumed square).
    - landmarks1 (list of tuples): Landmark points on the first image (source image).
    - landmarks2 (list of tuples): Corresponding landmark points on the second image (target image).
    - 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 resized for visualization. Defaults

    Returns:
    - None: This function directly displays the image using matplotlib and saves the output visualization to disk

    Notes:
        The function uses OpenCV for reading and resizing images. It employs a CLAHE function to enhance image co
        Matplotlib is used for visualizing the images and landmarks. The color map switches based on the number
        of landmarks; if there are more than 15 landmarks, a cyclic colormap is used to differentiate them.
        This function is particularly useful for visualizing transformations and registrations in medical imaging
        similar fields where point correspondence is critical.
    """
    image1 = cv2.resize(cv2.imread(images[0]),(disp_size,disp_size))
    image2 = cv2.resize(cv2.imread(images[1]),(disp_size,disp_size))
    image1 = clahe(image1, 3)
    image2 = clahe(image2, 3)
    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: {len(landmarks1)} != {len(landm
    num_points = len(landmarks1)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
    fig.suptitle("Stage-{} Point Correspondences".format(snum), fontsize=14, fontweight='bold', y=0.925)
    ax1.set_title('Fixed Image')
    ax2.set_title('Moving Image')
    ax1.axis('off')
    ax2.axis('off')
    ax1.imshow(image1,cmap='gray')
    ax2.imshow(image2,cmap='gray')
    if num_points > 15:
        cmap = plt.get_cmap('tab20')
    else:
        cmap = ListedColormap(["red", "yellow", "blue", "lime", "magenta", "indigo", "orange", "cyan", "darkgreen
                               "maroon", "black", "white", "chocolate", "gray", "blueviolet"])
    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='white', alpha=0.5)
        circ1_2 = plt.Circle((x1, y1), radius2, facecolor=color, edgecolor='white')
        ax1.add_patch(circ1_1)
        ax1.add_patch(circ1_2)
        x2, y2 = point2
        circ2_1 = plt.Circle((x2, y2), radius1, facecolor=color, edgecolor='white', alpha=0.5)
        circ2_2 = plt.Circle((x2, y2), radius2, facecolor=color, edgecolor='white')
        ax2.add_patch(circ2_1)
        ax2.add_patch(circ2_2)
    plt.figtext(0.5, 0.115, "Note: {0} point correspondences were identified by the model for stage-{1}".format(l
    plt.savefig(os.path.join(rpth,'Stage'+str(snum)+'Point_Correspondences'+str(num)+'.png'))
    plt.show();

def original_image_point_correspondences(images,img_size, landmarks1, landmarks2, landmarks3, rpth, num, disp_siz
    """
    Visualizes point correspondences across three images typically representing fixed, moving,
    and transformed states in image processing tasks. This function plots and saves the visualizations
    showing landmark points overlaid on each image. The images are enhanced using CLAHE for better visibility,
    and the landmarks are scaled according to the specified display size.
```

```
    Parameters:
    - images (list of np.array): A list containing three image arrays for fixed, moving,
                                 and transformed states.
    - img_size (tuple): The original size (width, height) of the images before resizing.
    - landmarks1 (list of tuples): Landmark coordinates for the fixed image.
    - landmarks2 (list of tuples): Landmark coordinates for the moving image.
    - landmarks3 (list of tuples): Landmark coordinates for the transformed image.
    - rpth (str): Path to the directory where the result image will be saved.
    - num (int): A numeric label to differentiate the output file name.
    - disp_size (int, optional): The display size to which the images will be resized for visualization. Defaults

    Raises:
    - AssertionError: If the lengths of landmarks1, landmarks2, and landmarks3 do not match.

    Notes:
        The function uses CLAHE for contrast enhancement of the images.
        The landmarks are resized to fit the specified display size for visualization.
        A colormap is used to differentiate points; if the number of points exceeds 15, a 20-color map is used,
        otherwise a specific 15-color map is applied.
    """
    assert len(landmarks1) == len(landmarks2) == len(landmarks3), "All landmarks lists must have the same length
    num_points = len(landmarks1)
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle("Final Registration Results by Composing Transformations Estimated in Two Stages", fontsize=14,

    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(cv2.resize(images[0],(disp_size,disp_size)),1.2),cmap='gray')
    ax2.imshow(clahe(cv2.resize(images[1],(disp_size,disp_size)),1.2),cmap='gray')
    ax3.imshow(clahe(cv2.resize(images[2],(disp_size,disp_size)),1.2),cmap='gray')

    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", "indigo", "orange", "cyan", "darkgreen
                               "maroon", "black", "white", "chocolate", "gray", "blueviolet"])

    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, colors):
        # Landmarks for Image 1
        x1, y1 = point1
        circ1_1 = plt.Circle((x1, y1), radius1, facecolor=color, edgecolor='white', alpha=0.5)
        circ1_2 = plt.Circle((x1, y1), radius2, facecolor=color, edgecolor='white')
        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='white', alpha=0.5)
        circ2_2 = plt.Circle((x2, y2), radius2, facecolor=color, edgecolor='white')
        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='white', alpha=0.5)
        circ3_2 = plt.Circle((x3, y3), radius2, facecolor=color, edgecolor='white')
        ax3.add_patch(circ3_1)
        ax3.add_patch(circ3_2)

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

In [12]:

```
def estimate_affine_transformation(points):
    """
    Estimate affine transformation matrix using point correspondences.
```

```
    Parameters:
    - points (np.array): Array of (x, y) 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 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 by z-coordinate
    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 coefficients.

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

    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 provided coefficients. This function is
    typically used in image processing and computer vision tasks to deform points according to a quadratic model.

    Parameters:
    - points (list of tuples): A list of points, where each point is represented as a tuple (x, y).
    - coefficients (list): A list of 12 coefficients for the quadratic transformation model.

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

    Raises:
    - ValueError: If the number of coefficients is not equal to 12, as the quadratic model requires exactly 12 co

    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 transformed coordinates.
        The coefficients must be specified in the order [a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12].
    """
    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_points,new_image_size,image_size):
    """
    Compute the landmark error between fixed points and transformed moving points.

    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-order polynomial.
    - 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, polynomial_matrix)
    transformed_points = coordinates_rescaling_high_scale(transformed_points,new_image_size,new_image_size, image
    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_image_size,new_image_size):
    """
    Calculates the mean landmark error between fixed points and transformed moving points
    after rescaling to a new image size. This function is primarily used in image processing
    to measure the accuracy of image registration by quantifying the displacement of landmark points.

    Parameters:
    - fixed_points (list of tuples): Coordinates of landmark points in the fixed image as (x, y) tuples.
    - fixed_image_size (tuple): The original size (width, height) of the fixed image.
    - moving_points (list of tuples): Coordinates of landmark points in the moving image as (x, y) tuples.
    - moving_image_size (tuple): The original size (width, height) of the moving image.
    - new_image_size (int): The size to which both sets of points will be resized.

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

    Notes:
        The function first rescales the coordinates of both fixed and moving points to a new size.
        It then calculates the Euclidean distance between the corresponding rescaled points.
        This metric is useful for evaluating the precision of image registration methods, particularly in medical
    """
    rescaled_fixed_points = coordinates_rescaling_high_scale(fixed_points,new_image_size,new_image_size,fixed_ima
    rescaled_moving_points = coordinates_rescaling_high_scale(moving_points,new_image_size,new_image_size,moving_
    errors = np.linalg.norm(np.array(rescaled_fixed_points) - rescaled_moving_points, axis=1)
    mle = np.mean(errors)
    return mle
```

```python
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, and at least 10 points are requir

    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, 0, 0, 0, 0, 0, 0, 0, 0, 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, x**2, x * y, y**2, x, y, 1])

        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, and at least 6 points are require

    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(landmarks2))
    return homography_matrix
```

```python
def transform_points_third_order_polynomial_matrix(landmarks1, landmarks2,img_size,new_img_size):
    """
    Computes a third-order polynomial transformation matrix based on rescaled landmark points from one image spac
    to another. This transformation is typically used for tasks like geometric transformation of images where pre
    alignment or registration of image features is necessary.

    Parameters:
    - landmarks1 (list of tuples): List of original landmark points in the source image given as (x, y) tuples.
    - landmarks2 (list of tuples): List of corresponding landmark points in the target image given as (x, y) tupl
                                   The points in landmarks2 should correspond one-to-one with those in landmarks1.
    - img_size (int): Original size of the images from which the landmarks were extracted. This is used to help
                    rescale points for accurate computation of the transformation matrix.
    - new_img_size (int): New size to which the points will be rescaled before computing the transformation matri
                        This should reflect the size of the image space into which the points will be transformed

    Returns:
    - numpy.ndarray: A transformation matrix which can be used to map the points from the space defined by landma
                     to the space defined by landmarks2. The matrix is represented as a 10x1 array of coefficie
                     corresponding to the terms of a third-order polynomial.

    Notes:
        Ensure that the number of points in landmarks1 and landmarks2 are equal and that they correspond to each
        This function involves rescaling coordinates, calculating a transformation matrix, and is typically used
        tasks where geometric transformations are necessary for alignment and registration.
    """
    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(landmarks1, landmarks2)
    return third_order_polynomial_matrix

def transform_points_quadratic_matrix(landmarks1, landmarks2,img_size,new_img_size):
    """
    Computes a quadratic transformation matrix based on rescaled landmarks from one set of image coordinates to a

    This function rescales the input landmarks from their original dimensions (img_size) to new dimensions (new_i
    It then calculates a quadratic transformation matrix that describes how points from the first set of landmark
    can be transformed to align with the second set (landmarks2). This matrix could be used to apply geometric tr
    to images or coordinates.

    Parameters:
    - landmarks1 (list of tuples): List of (x, y) tuples representing original landmarks in the source image.
    - landmarks2 (list of tuples): List of (x, y) tuples representing target landmarks in the target image,
                                   corresponding to landmarks1.
    - img_size (int): The original size (width and height, assumed square) of the images from which the landmarks
    - new_img_size (int): The new size (width and height, assumed square) to which the images and landmarks are r
                        before computing the transformation matrix.

    Returns:
    - numpy.ndarray: A matrix that contains the coefficients of the quadratic transformation. This matrix is used
                     to transform points from the source image to the target image based on the calculated poly

    Notes:
        Ensure that the number of points in landmarks1 and landmarks2 are equal and that they correspond to each
        This function is essential in image processing tasks where precise transformations are necessary for imag
    """
    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 coefficients, effectively deformir

    Parameters:
    - image (numpy.ndarray): The image to deform, provided as a numpy array. The array can be either
                            two-dimensional (grayscale image) or three-dimensional (color image).
    - coefficients (list or array): An array of 20 coefficients for the third-order polynomial transformation.

    Raises:
    - ValueError: If the number of coefficients provided is not 20, an error is raised due to the requirement
                of exactly 20 coefficients to perform the transformation.

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

    Notes:
        The deformation is defined by a polynomial transformation that adjusts the coordinates of each pixel
```

```
        based on the polynomial defined by the coefficients.
        This function supports both grayscale and color images. For color images, the transformation is applied
        to each color channel independently.
        The transformation involves calculating new pixel positions and mapping the original pixel values
        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 consistency
    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**2 + a4*y_coords**3 +
               a5*x_coords**2 + a6*x_coords*y_coords + a7*y_coords**2 + a8*x_coords + a9*y_coords + a10)
    y_prime = (a11*x_coords**3 + a12*x_coords**2*y_coords + a13*x_coords*y_coords**2 + a14*y_coords**3 +
               a15*x_coords**2 + a16*x_coords*y_coords + a17*y_coords**2 + a18*x_coords + a19*y_coords + a20)

    # 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], order=1, mode='constant', cval=0.0

    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 coefficients.

    Parameters:
    - image (numpy.ndarray): The image to deform, represented as a numpy array. This array can be
                             either two-dimensional (grayscale image) or three-dimensional (color image).
    - coefficients (list or array): A list or array of 12 coefficients defining the quadratic transformation.

    Raises:
    - ValueError: If the number of coefficients provided is not equal to 12, raises an error indicating
                that exactly 12 coefficients are required for the transformation.

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

    Notes:
        The deformation involves calculating new pixel coordinates using the quadratic equation defined
        by the coefficients and then mapping the original pixel values to these new coordinates.
        The function checks if the image is in grayscale or color and processes each channel independently.
        The mapping of pixels uses spline interpolation of order 1 for accuracy and fills any areas outside
        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 consistency
    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 + a5*y_coords**2 + a6
    y_prime = a7*x_coords + a8*y_coords + a9*x_coords*y_coords + a10*x_coords**2 + a11*y_coords**2 + a12

    # 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], order=1, mode='constant', cval=0.0

    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, landmarks2, rpth, num,snum,cll=1
    """
    Computes a third-order polynomial transformation matrix based on landmark correspondences
    between two images and applies this transformation to align one image with another. This function
    also displays and saves the original and transformed images, enhancing their contrast for better
    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 should be resized.
    - landmarks1 (list of tuples): Coordinates of landmarks in the source image.
    - landmarks2 (list of tuples): Corresponding coordinates of landmarks in the target image.
    - 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.
    - cll (float, optional): Clipping limit for the CLAHE algorithm, used for contrast enhancement. Default is 1.

    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 with the transformed image.
        - imgs (list of str): Paths to the saved output images.
        - coefficients (np.array): Coefficients of the third-order polynomial used for the transformation.

    Notes:
        This function is suited for complex registration tasks where finer control over the transformation is req
        The transformation matrix is applied to the target image to align it with the source image, effectively w
        The images are displayed and saved with enhanced contrast to aid in visual assessment of the registration
    """
    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")

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

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

    # Display and save the images
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    fig.suptitle("Stage-{} Results: Registration Using Third Order Polynomial Transformation".format(snum), fonts

    axs[0].imshow(clahe(img1, cll))
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')
```

```
    axs[1].imshow(clahe(img2, cll))
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(clahe(transformed_image.astype(np.uint8), cll))
    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'),transformed_image);
    return imags,imgs,coefficients




def compute_affine_matrix_and_plot(images,img_size,landmarks1, landmarks2,rpth,num,snum,cll=1.5):
    """
    Computes an affine transformation matrix based on provided landmarks from two images and applies
    this transformation to visually compare the source, target, and transformed images.

    This function computes the affine transformation matrix that best maps the source image to align
    with the target image using landmark correspondences. It then applies this transformation to
    the source image and displays the original (source and target) and transformed images side-by-side.
    The images are enhanced using CLAHE for better visibility and are saved to the specified path.

    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 resized.
    - landmarks1 (list of tuples): Landmark points (x, y) on the source image.
    - landmarks2 (list of tuples): Corresponding landmark points (x, y) on the target image.
    - 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.
    - cll (float, optional): Clipping limit for the CLAHE algorithm used in contrast enhancement. Default is 1.5.

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

    Raises:
    - ValueError: If the landmarks list is empty, indicating insufficient data to compute the matrix.

    Notes:
        The affine transformation matrix is computed using a least squares method based on provided landmarks.
        This function is useful for tasks in image registration where visual comparison of alignment is required.
        Enhanced contrast is used to aid in the visual assessment of image registration quality.
    """
    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")

    # 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], img2.shape[0]))
    imags.append(transformed_image)
    # Display the original and transformed images
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    fig.suptitle("Stage-{} Results: Registration Using Affine Transformation".format(snum), fontsize=14, fontweig

    axs[0].imshow(clahe(img1, cll))
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(clahe(img2, cll))
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(clahe(transformed_image.astype(np.uint8), cll))
    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'),transformed_image);
    return imgs,imags,affine_matrix

def compute_quadratic_matrix_and_plot(images,img_size,landmarks1, landmarks2,rpth,num,snum,cll=1.5):
    """
    Computes a quadratic transformation matrix from source to target landmarks and applies this transformation
    to the source image. The transformed source image is displayed alongside the original source and target image
    and all images are saved to disk.

    This function takes pairs of corresponding landmarks from the source and target images to compute a quadratic
    transformation matrix. This matrix is then used to warp the source image to match the target image. The
    result, along with the original images, is displayed and saved for comparison.

    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 resized.
    - landmarks1 (list of tuples): Landmark points (x, y) on the source image.
    - landmarks2 (list of tuples): Corresponding landmark points (x, y) on the target image.
    - 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 contrast enhancement. Default is 1.5.
    - snum (int): Stage number used for displaying in the title of the plot.

    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 original and transformed images.
        - quadratic_matrix (numpy.ndarray): The computed quadratic transformation matrix.

    Raises:
    - AssertionError: If the number of points in `landmarks1` and `landmarks2` are not equal, since a matching
                    number of points is required for matrix computation.

    Notes:
        The function uses OpenCV for image processing tasks such as reading, resizing, transforming, and saving i
        The quadratic transformation matrix is computed using a least squares method based on provided landmarks.
        Matplotlib is used for visualizing the before and after effects of the transformation.
        This function is particularly useful in applications such as image registration and geometric transformat
    """
    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)
```

```
    assert len(landmarks1) == len(landmarks2), "landmarks lists must have the same length."

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

    quadratic_matrix_for_image_deformed = compute_quadratic_matrix(landmarks2, landmarks1)

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

    # Apply the quadratic transformation using cv2.warpquadratic
    transformed_image =  warp_image_quadratic_matrix(img2, quadratic_matrix_for_image_deformed)
    transformed_image = cv2.resize(transformed_image,  (img2.shape[1], img2.shape[0]))
    imags.append(transformed_image)

    # Display the original and transformed images
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    fig.suptitle("Stage-{} Results: Registration Using Quadratic Transformation".format(snum), fontsize=14, fontw

    axs[0].imshow(clahe(img1, cll))
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(clahe(img2, cll))
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(clahe(transformed_image.astype(np.uint8), cll))
    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'),transformed_image);
    return imgs,imags,quadratic_matrix

def compute_homography_matrix_and_plot(images, img_size, landmarks1, landmarks2, rpth, num,snum,cll=1.5):
    """
    Computes the homography transformation matrix based on landmark correspondences between two images
    and applies this transformation to the source image. The function displays the original source and
    target images along with the transformed source image. It also saves these images to disk.

    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 target image.
    - rpth (str): The directory path where the resultant images will be saved.
    - num (int): An identifier number used to differentiate the output file names.
    - snum (int): Stage number used for displaying in the title of the plot.
    - cll (float): The clipping limit for the CLAHE algorithm used in contrast enhancement. Default is 1.5.

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

    Raises:
    - ValueError: If the list of landmarks is empty, indicating that there are not enough data points to compute

    Notes:
        The function uses OpenCV for image reading, resizing, and applying the homography transformation.
        Matplotlib is used for displaying the images.
        Ensure the landmarks are accurately defined as their correspondence directly affects the quality of the t
        Homography transformations are particularly useful for applications in image registration, computer visio
    """
    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")

    # 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], img2.shape[0]))
    imags.append(transformed_image)

    # Display the original and transformed images
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    fig.suptitle("Stage-{} Results: Registration Using Homography Transformation".format(snum), fontsize=14, font

    axs[0].imshow(clahe(img1, cll))
    axs[0].set_title('Fixed Image')
    axs[0].axis('off')

    axs[1].imshow(clahe(img2, cll))
    axs[1].set_title('Moving Image')
    axs[1].axis('off')

    axs[2].imshow(clahe(transformed_image.astype(np.uint8), cll))
    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'),transformed_image);
    return imgs,imags,homography_matrix
```

In [13]:

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

    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 (set2).

    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 affine transformation and
    removing pairs that have an error greater than a specified threshold. The function first estimates
    an affine transformation matrix based on all given point pairs. Each point in the first set is then
    transformed using this matrix, and the error is calculated as the Euclidean distance between the
    transformed point and the corresponding point in the second set. Points with an error exceeding the
    threshold are considered outliers and are excluded from the results.

    Parameters:
    - set1 (list of tuples): A list of (x, y) tuples representing coordinates of points in the first image.
    - set2 (list of tuples): A list of (x, y) tuples representing corresponding coordinates of points in the
                             second image. The indices in `set1` and `set2` must correspond.
    - threshold (float, optional): The maximum allowed error distance between the original and transformed
                                   points for them to be considered inliers. Default value is 20.

    Returns:
    - tuple of lists: Returns two lists (updated_set1, updated_set2) containing the inlier points from
                      `set1` and `set2` respectively.
    Notes:
        It is critical that `set1` and `set2` are of equal length and that the points correspond correctly,
        as any misalignment could result in incorrect calculations and poor results.
        This function is typically used in image processing and computer vision tasks where alignment and
        transformation of point sets between images is required, particularly in stereo vision and motion trackir
    """
    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 homography transformation
    and removing pairs that have an error greater than a specified threshold. The function first estimates
    a homography transformation matrix based on all given point pairs. Each point in the first set is then
    transformed using this matrix, and the error is calculated as the Euclidean distance between the
    transformed point and the corresponding point in the second set. Points with an error exceeding the
    threshold are considered outliers and are excluded from the results.

    Parameters:
    - set1 (list of tuples): A list of (x, y) tuples representing coordinates of points in the first image.
    - set2 (list of tuples): A list of (x, y) tuples representing corresponding coordinates of points in the
                             second image. The indices in `set1` and `set2` must correspond.
    - threshold (float, optional): The maximum allowed error distance between the original and transformed
                                   points for them to be considered inliers. Default value is 20.

    Returns:
    - tuple of lists: Returns two lists (updated_set1, updated_set2) containing the inlier points from
                      `set1` and `set2` respectively.
    Notes:
        Ensure that `set1` and `set2` are of equal length and that the points correspond correctly,
        as any misalignment could result in incorrect calculations and poor results.
```

```
        This function is typically used in image processing and computer vision tasks where precise
        alignment and transformation of point sets between images are required, especially in applications
        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_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 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 filters out outliers based on a specif

    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 to compare against.
    - criteria (str, optional): The criteria to use for filtering outliers. Options are 'affine' or 'homography'.
    - thresh (int, optional): Threshold value used in the outlier removal process. Defaults to 20.

    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 homography matrix between the compute
        If 'affine' is chosen, it removes outliers based on affine transformation error exceeding the threshold.
    """
    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,original,thresh)
    return computed,original
```

In [14]:

```
def coordinates_rescaling_high_scale(pnts,H,W,img_shape):
    """
    Rescale a list of coordinates based on given distinct 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 CLAHE_Images(imags, clip):
    """
    Applies Contrast Limited Adaptive Histogram Equalization (CLAHE) to a list of image files to enhance
    their contrast. This method is particularly useful for improving the visibility of features in images
    that suffer from poor contrast.

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

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

    Notes:
        This function uses OpenCV's `createCLAHE` method to apply the CLAHE algorithm. Each image is
        first converted to grayscale as CLAHE is typically applied to single-channel images for better
        visualization of detail.
        The images are processed in-place and saved in the same directory as the original, with 'CLAHE_'
        prefixed to their original filenames.
        It is recommended to adjust the `clip` parameter based on the specific requirements of the image
        content and the desired level of contrast enhancement.
    """
    imgs=[]
    clahe = cv2.createCLAHE(clipLimit=clip, tileGridSize=(8, 8))
    for img in imags:
      fn,_ = os.path.splitext(os.path.basename(img))
      ifn = '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 specified uniform size using biline

    Parameters:
    - feature_maps (list of tensors): A list of feature map tensors to be resized.
    - 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='bilinear', align_corners=False))
    return uniform_feature_maps


def multi_resolution_features(images,img_size,N,clip,offset,window_size,max_dist,timestep,up_ft_indices,multi_ch
    """
    Generate multi-resolution features from images using SIFT, and Random Points.

    This function processes images to generate feature maps at multiple resolutions. It combines techniques like

    Parameters:
    - 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 Model.
    - multi_ch (bool): Flag to indicate multi-channel mode.
    - multi_img_size (int): The size of the images for multi-resolution processing.
    - 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):
            sks,pts = SIFT_top_n_keypoints(images[0],N,multi_img_size*(i+1),max_dist)
            pts = pts+select_random_points(images[0],N,multi_img_size*(i+1),offset,window_size)
            if clip > 0:
                images = CLAHE_Images(images, clip = clip)
            dft = DFT(images,multi_img_size*(i+1),pts)
            src_ft1,trg_ft1 = dft.feature_upsampling(RetinaRegNet_Intialization(images,multi_img_size*(i+1),time
            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:
        sks,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)
        src_ft,trg_ft = dft.feature_upsampling(RetinaRegNet_Intialization(images,img_size,timestep,up_ft_indices
    return src_ft,trg_ft

def landmarks_condition_check(orig_images, img_size, t, uft, landmarks1, landmarks2, max_tries=2, num=100, clip =
    """
    Iteratively attempts to improve image registration quality by enhancing image contrast and adjusting landmark
    until certain quality conditions are met or a maximum number of attempts is reached. This function applies CL
    for image contrast enhancement and uses various feature transformation and scaling techniques to improve the
    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.
    - t (float): Threshold parameter for initializing the Diffusion Model.
    - uft (float): Parameter for extracting diffusion features from the diffusion model.
    - landmarks1 (list of tuples): Initial landmarks as (x, y) coordinates in the first image.
    - landmarks2 (list of tuples): Target landmarks as (x, y) coordinates in the second image.
    - max_tries (int, optional): Maximum number of attempts to improve image registration. Defaults to 2.
    - num (int, optional): Minimum required number of landmarks. Defaults to 100.
    - clip (float, optional): Clip limit for CLAHE. Defaults to 1.0.
    - N (int, optional): Number of points to be chosen at random for processing. Defaults to 100.
    - offset (float, optional): Offset used in point selection to avoid edge effects. Defaults to 0.01.
    - window_size (int, optional): Size of the window used in point selection. Defaults to 51.
    - iccl (float, optional): Inverse consistency criteria limit used in landmark filtering. Defaults to 3.
    - outlier_cond (str, optional): Condition used to determine outliers. Defaults to 'affine'.
    - thresh (float, optional): Threshold used for filtering outliers. Defaults to 20.

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

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

    Notes:
        This function is particularly useful in medical imaging or computer vision tasks where accurate image
        registration is crucial for further analysis.
        The effectiveness of the registration process depends heavily on the quality and accuracy of the input la
        CLAHE and other image processing techniques may not always produce the desired results if the input image
        are of poor quality or the initial landmarks are inaccurately defined.
    """
    imgs,lim,land_marks1, land_marks2,list_landmarks_2, list_sim_scores,list_landmarks_1,temp = [], [], [], [],
    tries, ch, = 0, 0
    assert len(landmarks1) == len(landmarks2), f"Points lengths are incompatible: {len(landmarks1)} != {len(landr
    landmarks2,landmarks1 = filter_outlier_cond(landmarks2,landmarks1,outlier_cond,thresh)
    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)
            images = orig_images
            pts = select_random_points(orig_images[0], N, img_size,offset,window_size)
            dft = DFT(images, img_size, pts)
            src_ft,trg_ft = dft.feature_upsampling(RetinaRegNet_Intialization(images,img_size,t + 75*tries,uft))
            land_marks1,sim_score, land_marks2 = dft.feature_maps(src_ft,trg_ft,iccl)
            del src_ft
            del trg_ft
            torch.cuda.empty_cache()
            gc.collect()
            land_marks2,land_marks1 = filter_outlier_cond(land_marks2,land_marks1,outlier_cond,thresh)
            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=True)

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 values.
    """
    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:])))] .append(i)
        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 replicate each element.

    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 corresponding files as values.

    Returns:
    - tuple: Lists containing paths to fixed images, moving images, and point files.
    """
    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(fixed)} != {len(moving)}."
    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_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 image.
    - mpnts (list of tuples): List of (x, y) coordinates related to the second image.
    - img_shape (int, optional): Desired image dimension for rescaling. Default is 256.

    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,W1),max(H2,W2)))
    return (H2,W2),(H1,W1),max(max(H1,W1),max(H2,W2)),scaled_fixed_points,scaled_moving_points,scaled_original_m



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 images.
    - 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_pointss,scaled_moving_points =[],[],[
    for i in range(len(images)):
        fhs,mhss,mhs,fpnts,mpnts,scmpnts = coordinates_processing(images[i][0],images[i][1],fixed_points[i],movi
        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,moving_pointss,scaled_moving_points

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

    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 = 2):
    """
    Initialize RetinaRegNet by processing a list of image files.

    Parameters:
    - filelist (list of str): List of paths to image files for feature extraction.
    - img_size (int, optional): Desired size for resizing images. Default is 256.
    - timestep (int, optional): Time step for the intializing the diffusion model. Default is 75.
    - up_ft_index (int, optional): Index for the extracting diffusion features from the diffusion model . Default

    Returns:
    - ft (torch.Tensor): A tensor containing the Diffusion features of the images in the list.

    Notes:
        The function uses the SDFeaturizer from the 'stabilityai/stable-diffusion-2-1' model to extract stable di
        from each image. After processing all images, the extracted features are concatenated into a single tensc
        To avoid memory issues, the function cleans up resources after processing.
    """
    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(images,rpth,ifn,stage_num,img_size=256,up_ft_indices = 1,timestep = 75,N=50,offset=0.01,window_size=51,r
    """
    Perform image registration and point correspondence using a series of processing steps.

    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 (default is 1).
    - 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 is 0.01).
    - window_size (int, optional): Size of the window for feature extraction (default is 51).
    - max_dist (int, optional): Maximum distance for feature matching (default is 5).
    - iccl (int, optional): ICC level for feature matching (default is 3).
    - outlier_cond (str, optional): Condition for outlier removal (default is 'affine').
    - thresh (int, optional): Threshold value for outlier removal (default is 20).
    - max_tries (int, optional): Maximum number of attempts for matching features (default is 3).
    - num (int, optional): Number of iterations for matching features (default is 50).
    - clip (float, optional): Clip parameter for image enhancement (default is 1.0).
    - multi_ch (bool, optional): Flag indicating whether to use multi-channel processing (default is True).
    - multi_iter (int, optional): Number of iterations for multi-channel processing (default is 3).
    - multi_img_size (int, optional): Size of images for multi-channel processing (default is 256).

    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 extraction, feature matching,
        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 both original and computed points
    """
    sks,pts = SIFT_top_n_keypoints(images[0],N,img_size,max_dist)
```

```
    pts = pts+select_random_points(images[0],N,img_size)
    if clip > 0:
            images = CLAHE_Images(images, clip = clip)
    dft = DFT(images,img_size,pts)
    src_ft,trg_ft = multi_resolution_features(images,img_size,N,clip,offset,window_size,max_dist,timestep,up_ft_
    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, timestep, up_ft_indices, pnts, rspts,
    if len(computed)!=0:
        image_point_correspondences(images[::-1],img_size,computed,original,rpth,ifn,stage_num)
        return original,computed
    else:
        print("Image Registration is Unsuccessful for the presented Images due to unsufficent Matching Features")
        return [],[]
    torch.cuda.empty_cache()
```

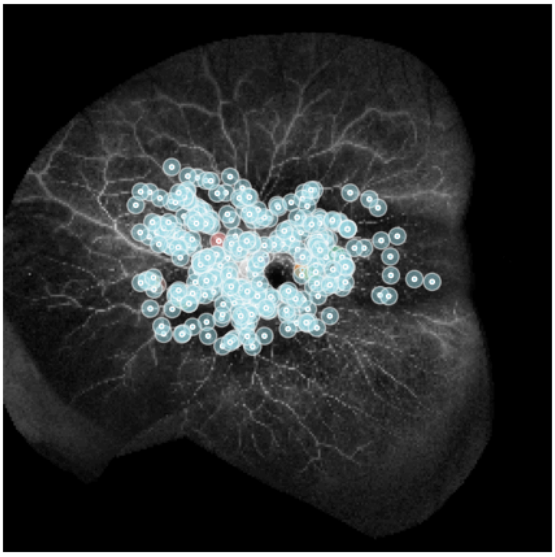
In [18]:
```
img_size= 1024
images,fixed_points,moving_points = data_preprocessing('FLoRI21_DataPort')
fixed_image_size,moving_image_size,max_image_size,scaled_fixed_points,scaled_moving_points,scaled_original_moving
```

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(images[i][1],images[i][0]))
    original_low_res,computed_low_res = main(images[i],os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registrat
    imags,imgs,homography_matrix_low_res = compute_homography_matrix_and_plot(images[i][::-1], img_size,original_
    if len(homography_matrix_low_res) !=0:
        transformed_points_hom = transform_points_homography(scaled_moving_points[i],homography_matrix_low_res)
        transformed_points_high_res_hom =  coordinates_rescaling(transformed_points_hom,img_size,img_size,max_ima
        original_low_res,computed_low_res = main(imags,os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registrat
        imag,imgs,polynomial_matrix_low_res = compute_third_order_polynomial_matrix_and_plot(imags[::-1], img_si
        if len(polynomial_matrix_low_res) !=0:
            ## rescaled version for dispaly purposes
            transformed_points_poly = transform_points_third_order_polynomial(transformed_points_hom, polynomial_
            imag[1]=cv2.imread(images[i][0])
            original_image_point_correspondences(imag,img_size, scaled_fixed_points[i], scaled_moving_points[i],
            ### Original Version for computation of errors
            polynomial_matrix = transform_points_third_order_polynomial_matrix(original_low_res,computed_low_res_
            bef_error = compute_landmark_error(fixed_points[i],fixed_image_size[i],moving_points[i],moving_image_
            aft_error = compute_landmark_error_fixed_space(polynomial_matrix,fixed_points[i],transformed_points_h
            print("Mean Landmark Error for Case {0} Before Registration is {1} pixels".format(i,bef_error))
            print("Mean Landmark Error for Case {0} After Registration is {1} pixels".format(i,aft_error))
            landmark_errors.append(aft_error)
        else:
            landmark_errors.append(10000)
    else:
        landmark_errors.append(10000)
```

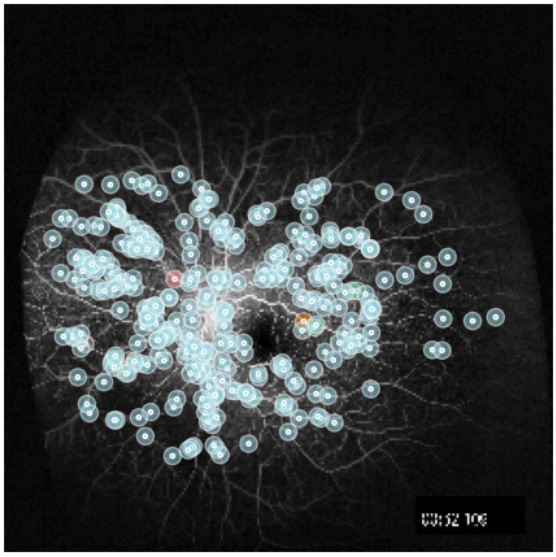
```
Case 0
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_1/FA/Raw_FA_1
_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: UserWarni
ng: Applied workaround for CuDNN issue, install nvrtc.so (Triggered internally at ../aten/src/ATen/native/cudnn/Co
nv_v8.cpp:80.)
  return F.conv2d(input, weight, bias, self.stride,
/scratch/local/29752099/ipykernel_4102899/3422066467.py:101: UserWarning: To copy construct from a tensor, it is r
ecommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  points_indices = torch.tensor(pts_list)
```

Stage-1 Point Correspondences

Fixed Image



Moving Image

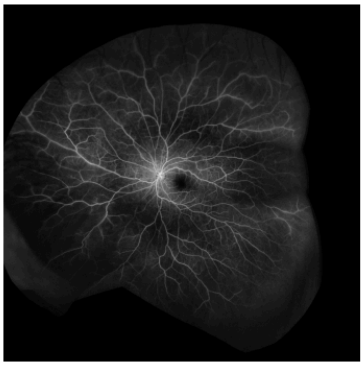


Note: 341 point correspondences were identified by the model for stage-1

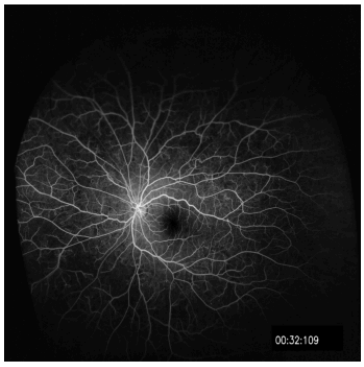
```
Homography Matrix:
[[8.45189846e-01 2.99915060e-02 1.77293829e+02]
 [1.27907586e-01 7.64249572e-01 6.85835688e+01]
 [2.10141751e-04 1.26162747e-04 1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

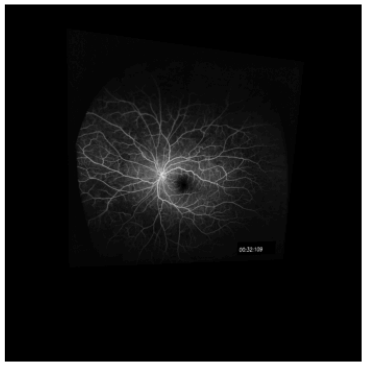
Fixed Image



Moving Image



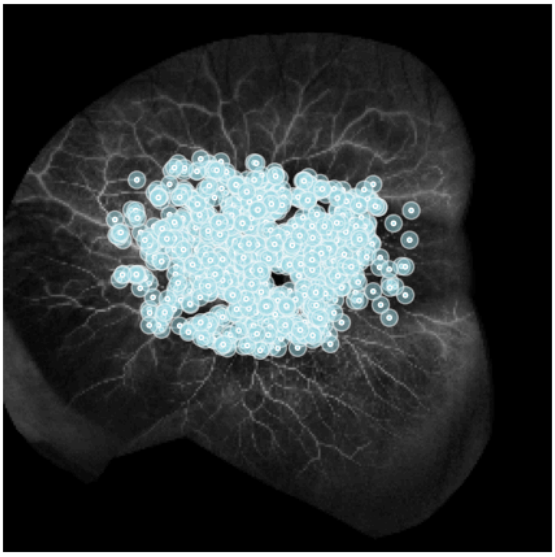
Deformed Image



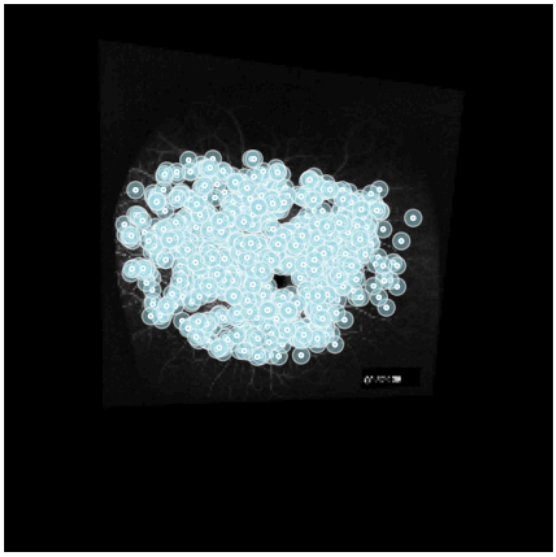
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



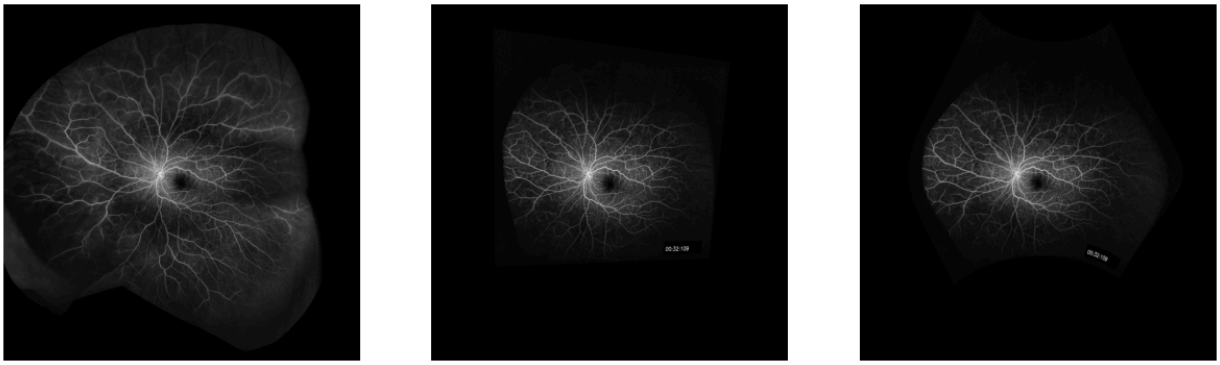
Moving Image



Note: 935 point correspondences were identified by the model for stage-2

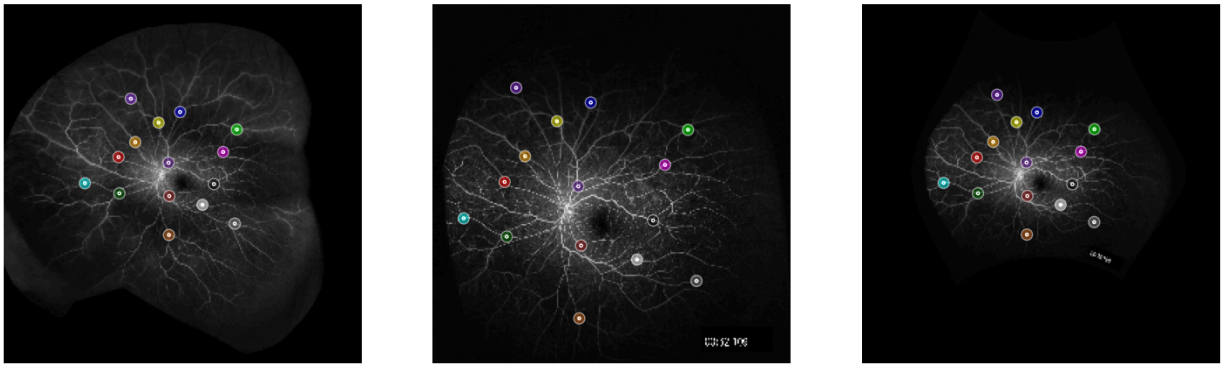
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



Final Registration Results by Composing Transformations Estimated in Two Stages

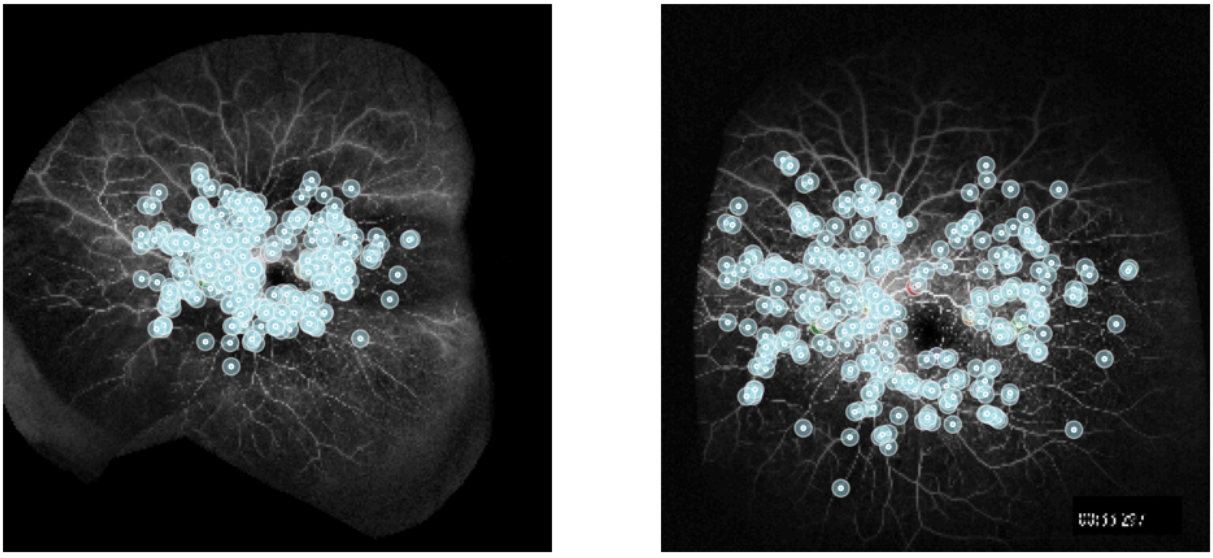
Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 0 Before Registration is 660.5727742589274 pixels
Mean Landmark Error for Case 0 After Registration is 16.298474218261774 pixels
Case 1
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_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

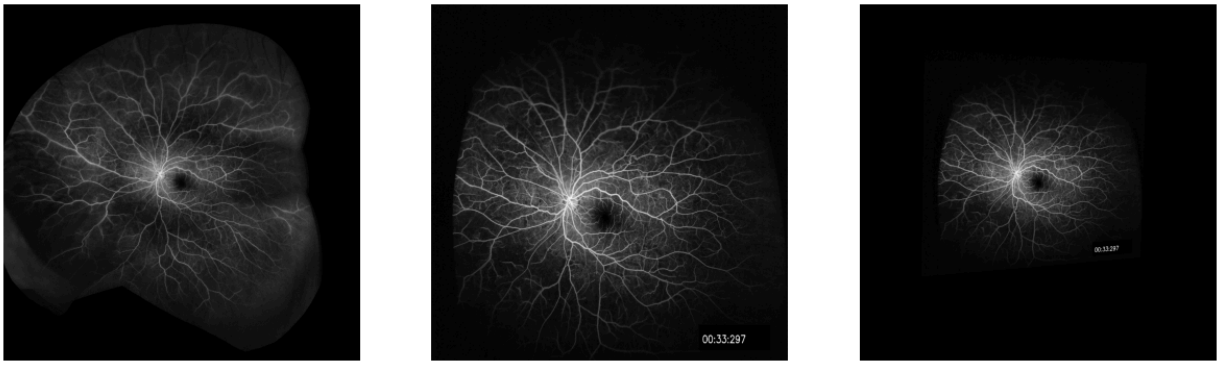


Note: 342 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 7.61731532e-01 -5.45989545e-03  1.84406763e+02]
 [ 6.65926947e-02  6.54084352e-01  1.27726677e+02]
 [ 1.66376261e-04  1.76032440e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

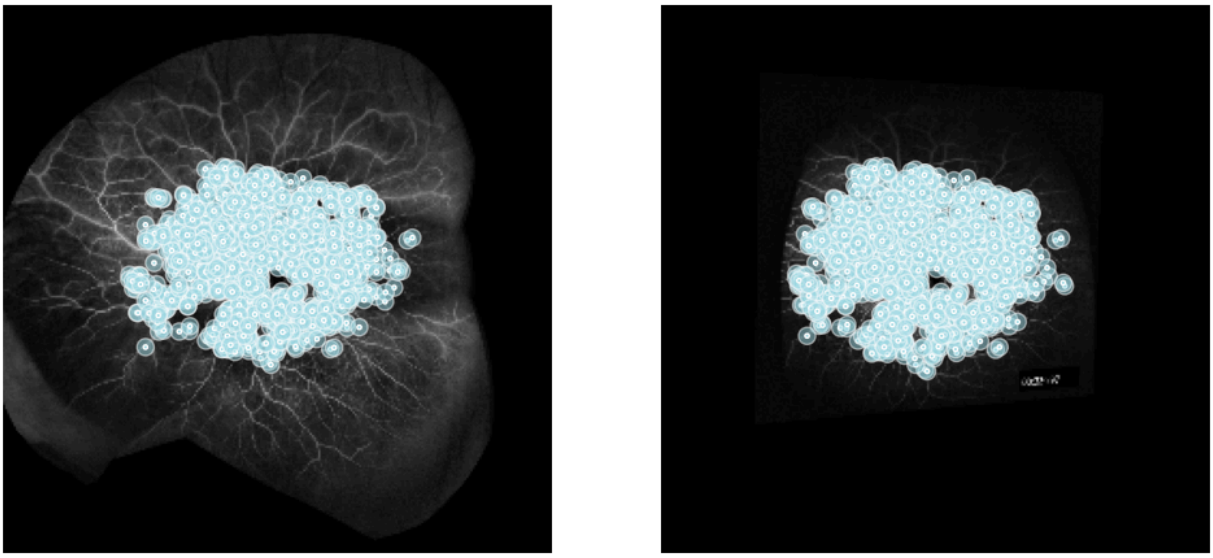
Fixed Image | Moving Image | Deformed Image



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

Stage-2 Point Correspondences

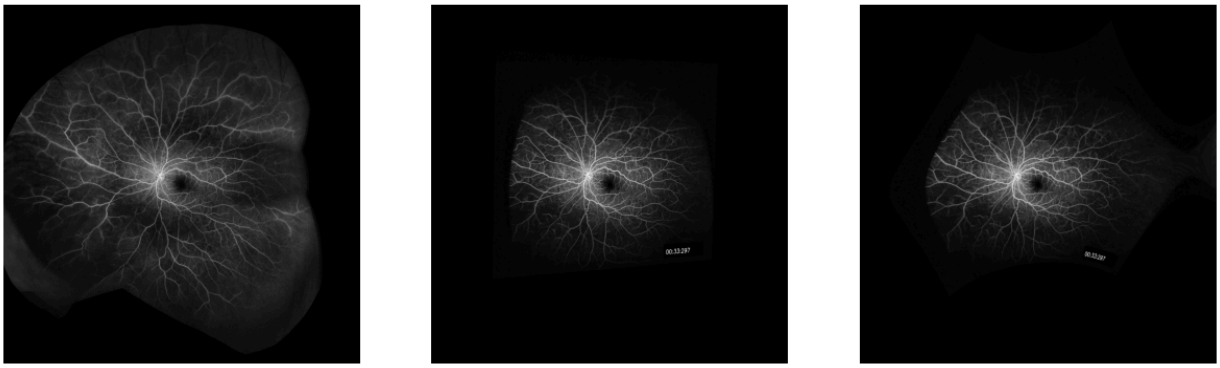
Fixed Image | Moving Image



Note: 1050 point correspondences were identified by the model for stage-2

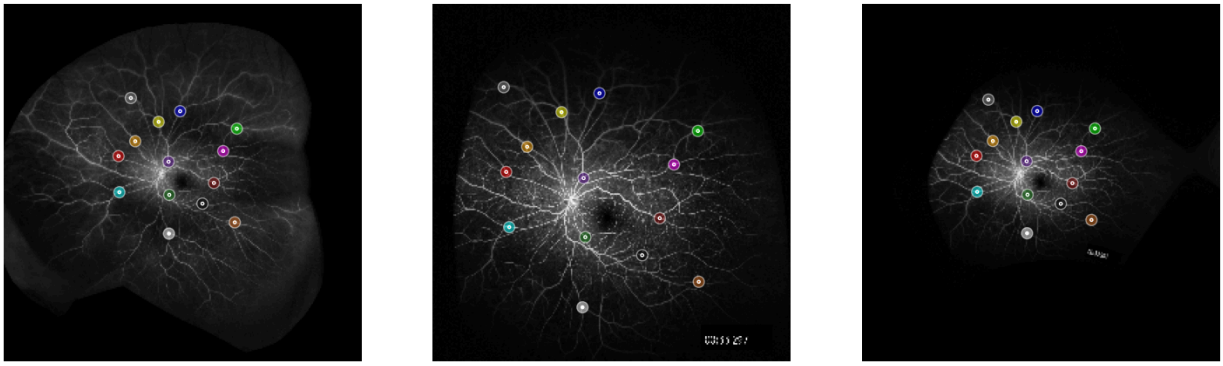
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image

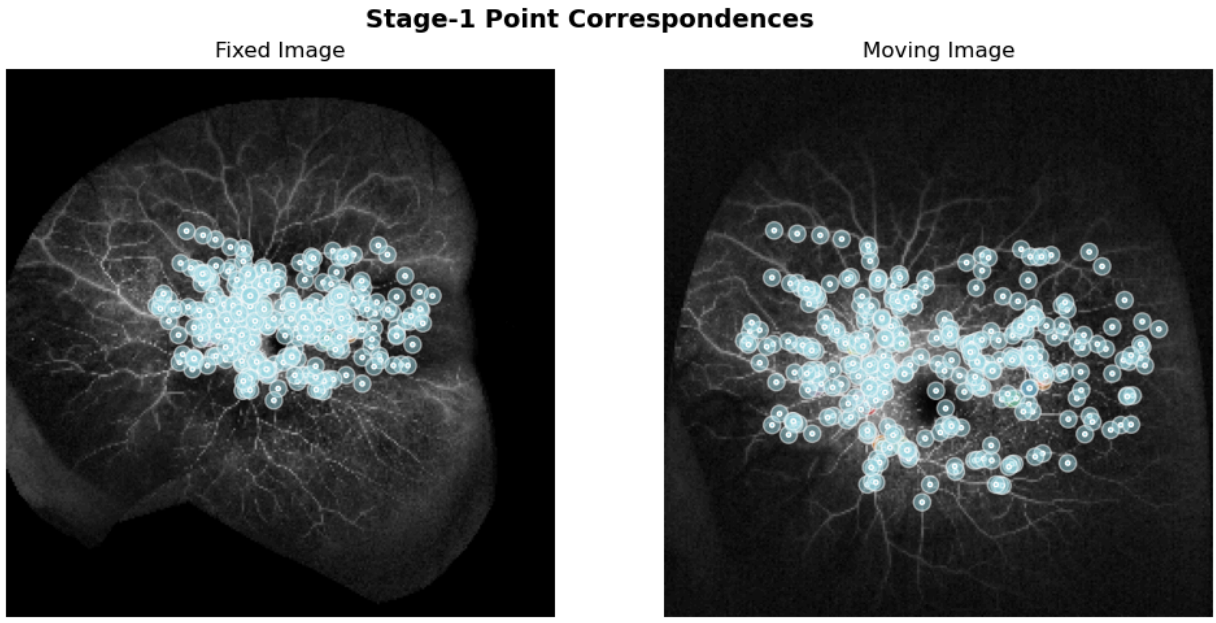


Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

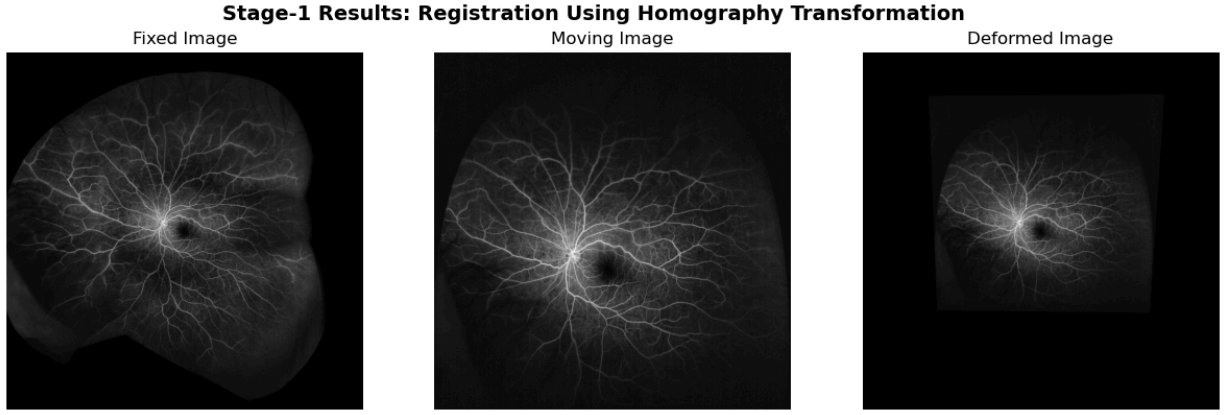


```
Mean Landmark Error for Case 1 Before Registration is 673.4817677455603 pixels
Mean Landmark Error for Case 1 After Registration is 18.64909699701632 pixels
Case 2
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_1/FA/Raw_FA_3
_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



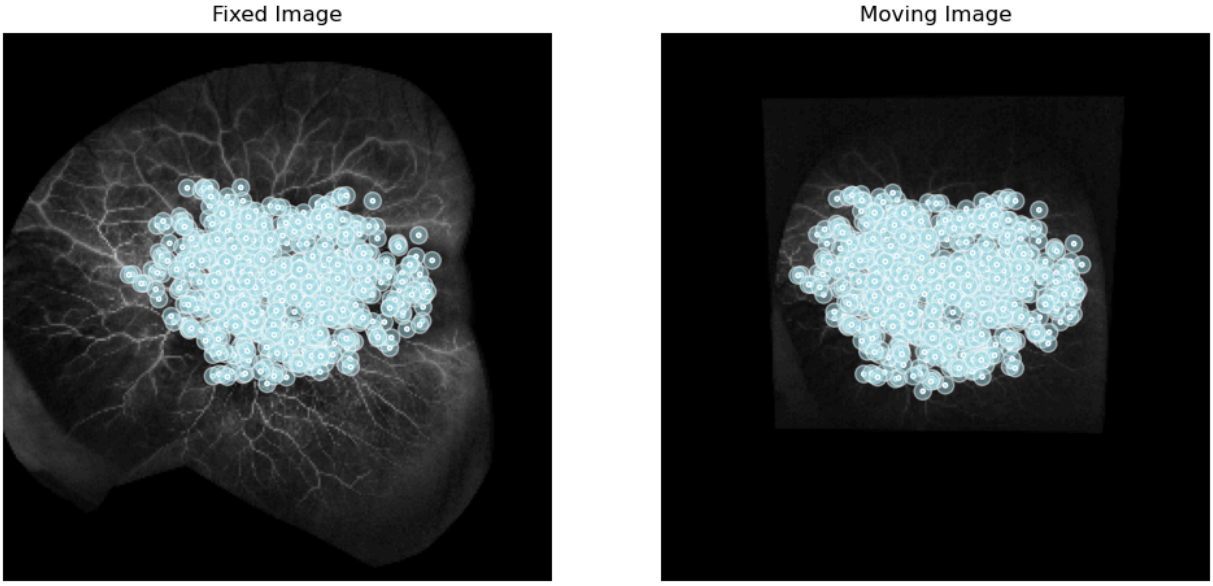
Note: 338 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 6.44268801e-01  4.57459472e-02  1.88411892e+02]
 [-6.36860001e-03  6.77485374e-01  1.23578934e+02]
 [-1.93768360e-05  1.03045277e-04  1.00000000e+00]]
```



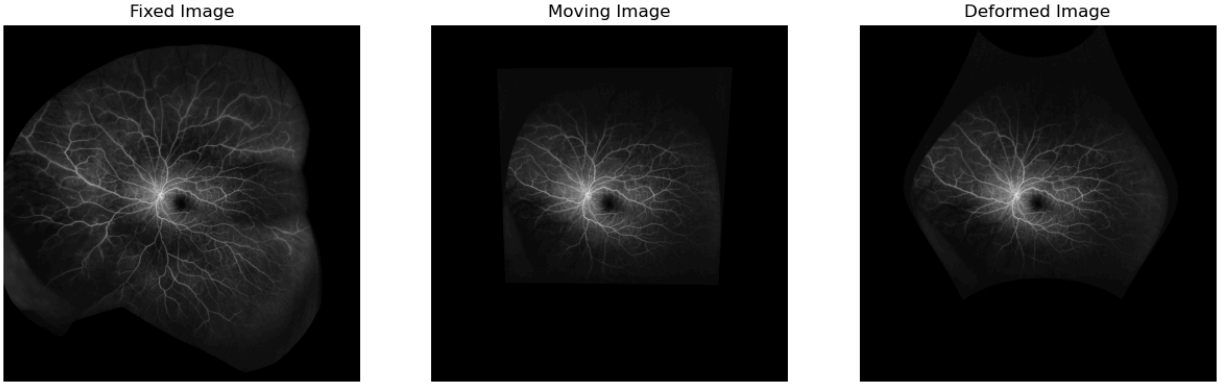

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

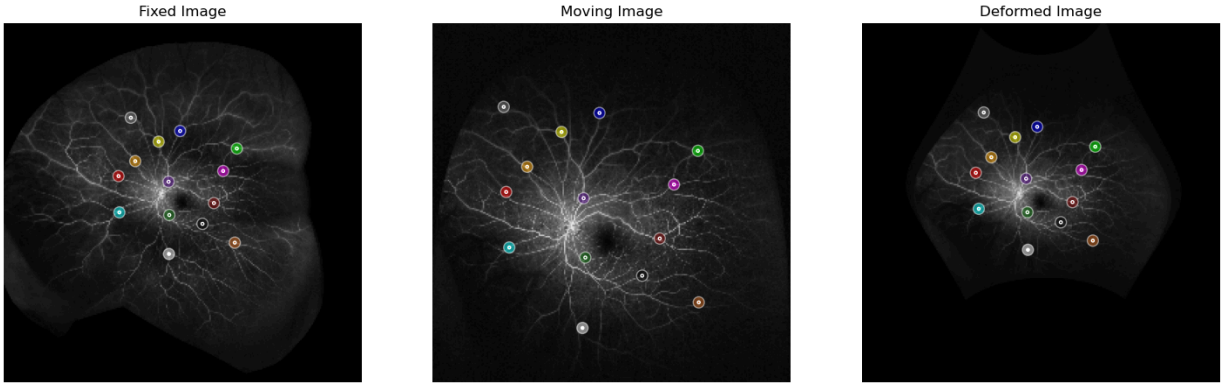


Note: 733 point correspondences were identified by the model for stage-2

Stage-2 Results: Registration Using Third Order Polynomial Transformation



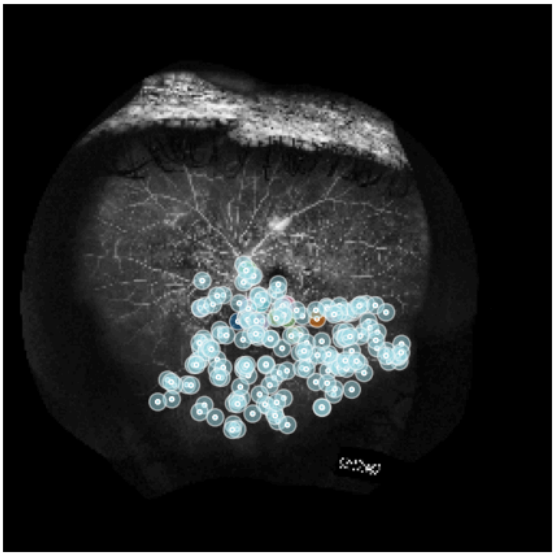
Final Registration Results by Composing Transformations Estimated in Two Stages



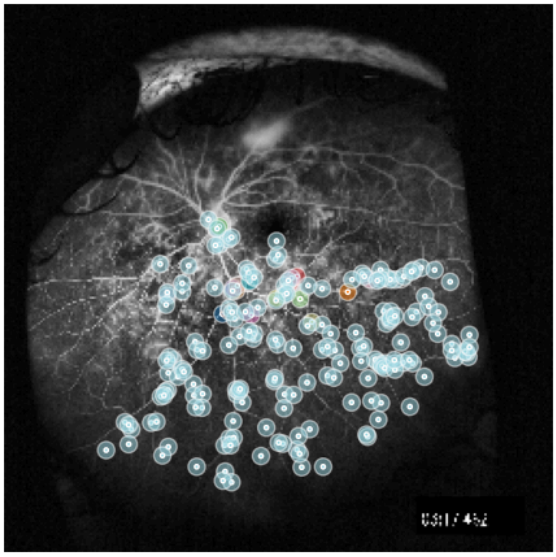

```
Mean Landmark Error for Case 2 Before Registration is 673.4817677455603 pixels
Mean Landmark Error for Case 2 After Registration is 39.69697930115462 pixels
Case 3
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/FA/Raw_FA_1
_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: 188 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 5.07549292e-01 -1.34478672e-01  2.59977125e+02]
 [ 1.50503126e-02  4.30593210e-01  2.64621741e+02]
 [-6.88996661e-05 -1.59254505e-04  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

Fixed Image

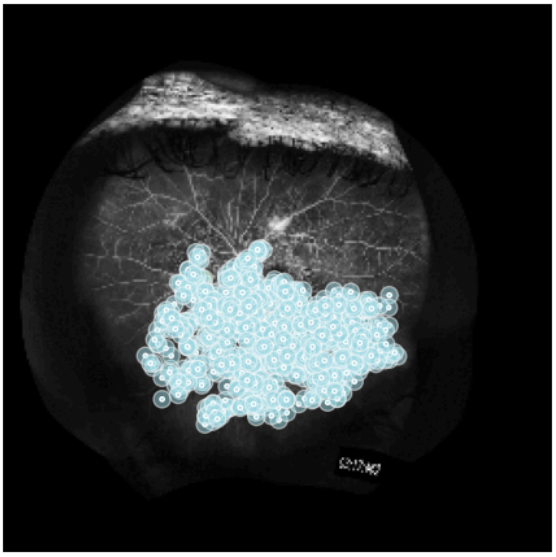
Moving Image

Deformed Image

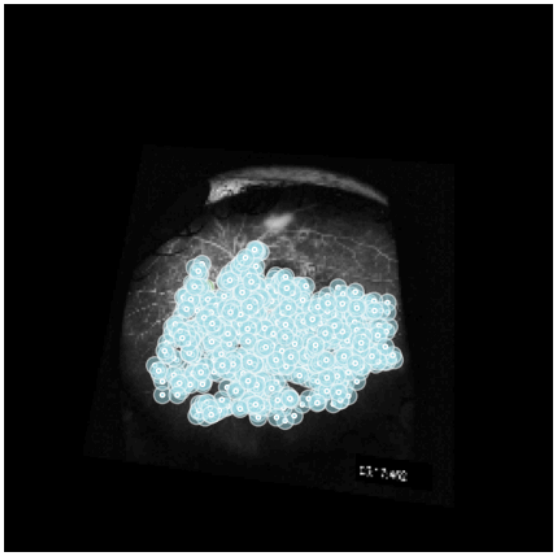
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



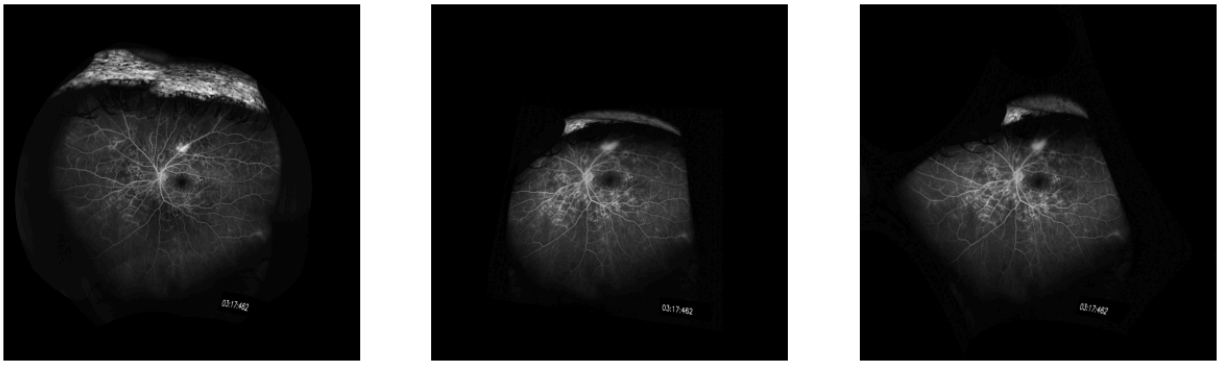
Moving Image



Note: 728 point correspondences were identified by the model for stage-2

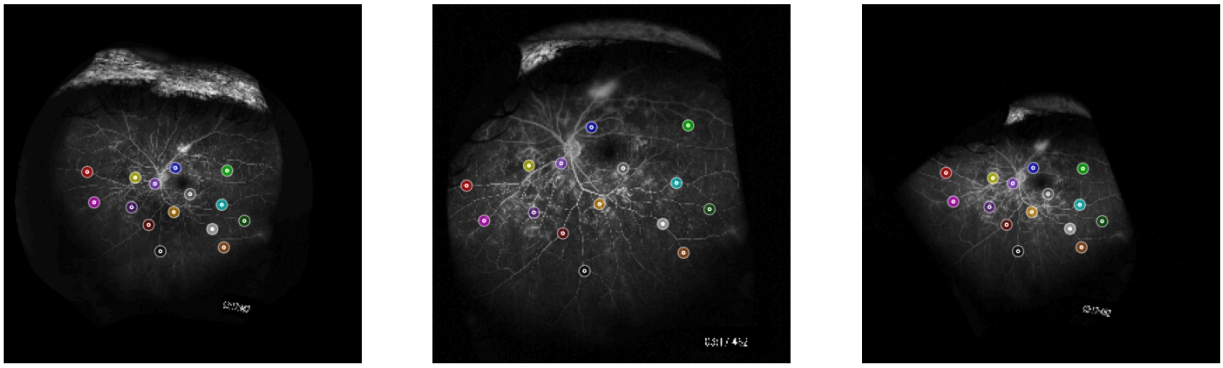
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



Final Registration Results by Composing Transformations Estimated in Two Stages

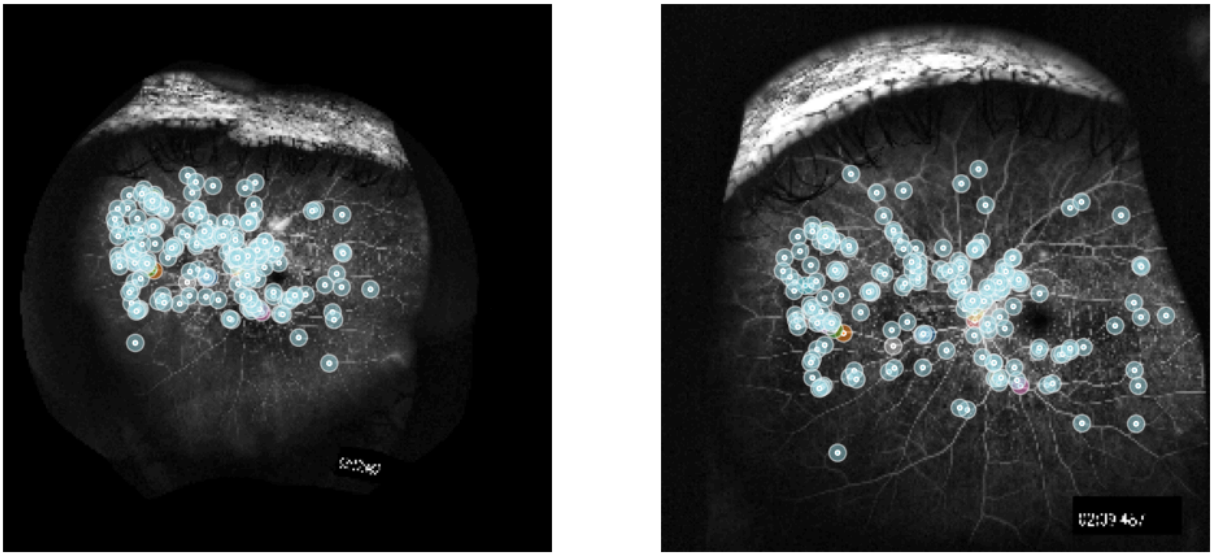
Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 3 Before Registration is 1013.6612836814534 pixels
Mean Landmark Error for Case 3 After Registration is 10.44375078207543 pixels
Case 4
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_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

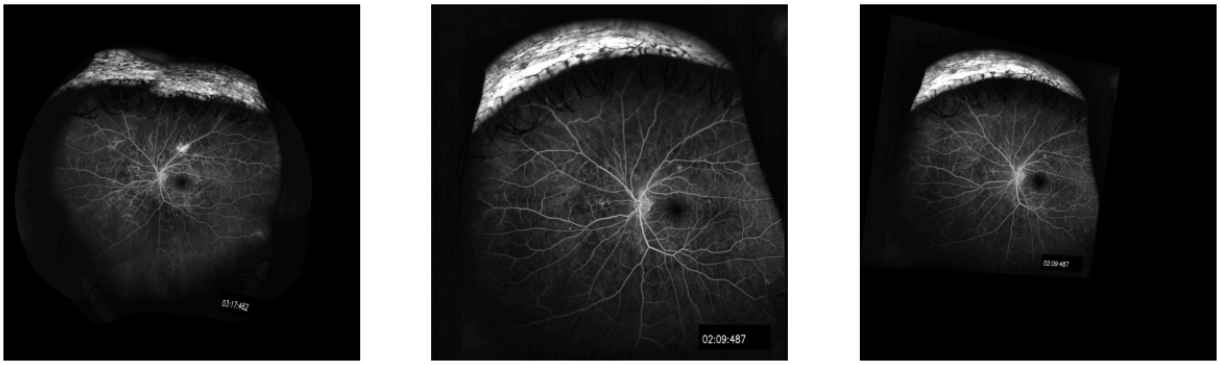


Note: 184 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 7.87179538e-01 -9.06890970e-02  8.96274068e+01]
 [ 1.86005884e-01  7.23249382e-01  2.65623141e+01]
 [ 1.91544098e-04  1.88493944e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

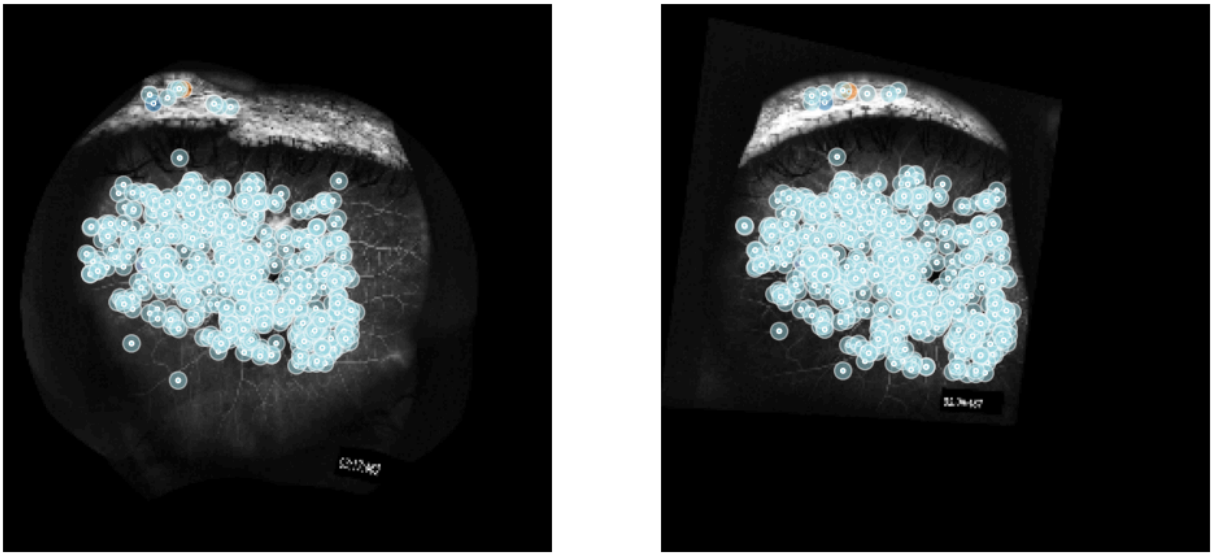
Fixed Image | Moving Image | Deformed Image



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

Stage-2 Point Correspondences

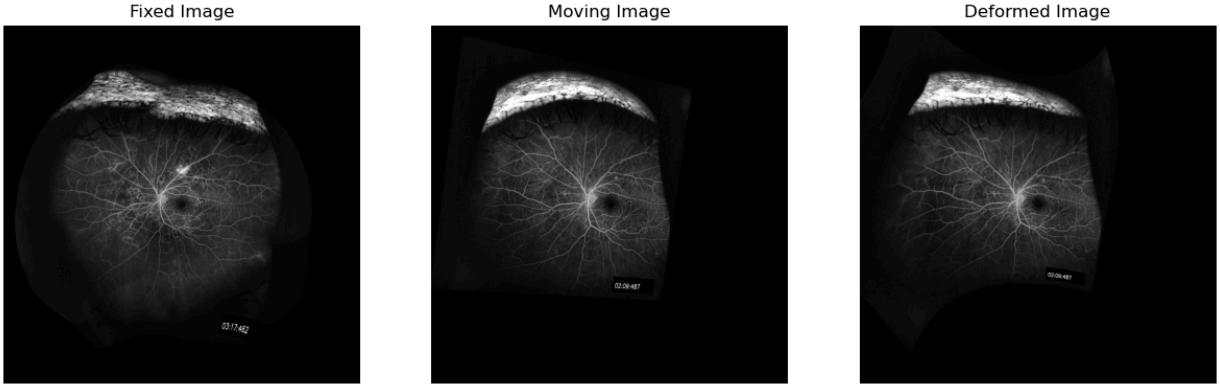
Fixed Image | Moving Image



Note: 617 point correspondences were identified by the model for stage-2

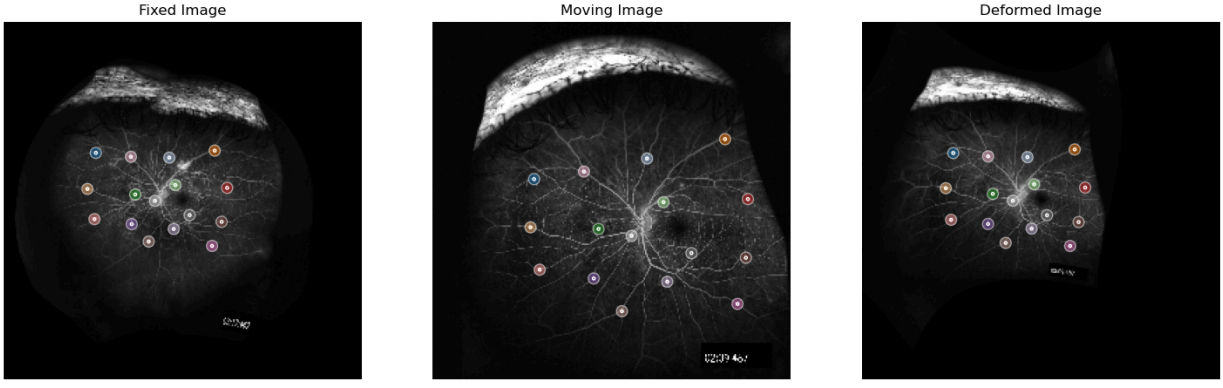
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



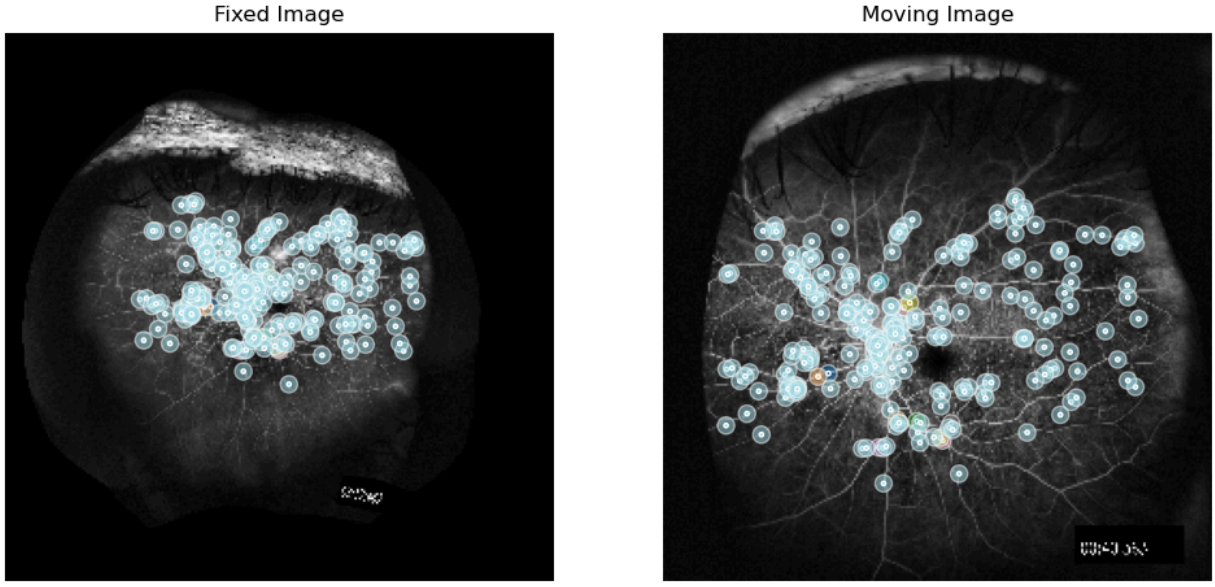
Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 4 Before Registration is 776.2780167748374 pixels
Mean Landmark Error for Case 4 After Registration is 10.370002734521773 pixels
Case 5
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/FA/Raw_FA_3
_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

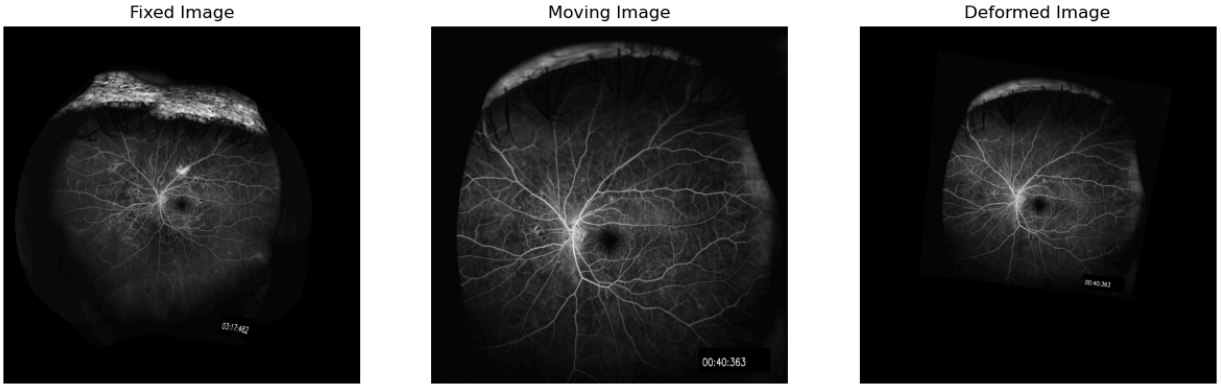
Stage-1 Point Correspondences



Note: 237 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 6.99581509e-01 -3.42843340e-02  2.23754294e+02]
 [ 1.12391807e-01  7.05041394e-01  6.82439680e+01]
 [ 4.52163727e-05  1.07559694e-04  1.00000000e+00]]
```

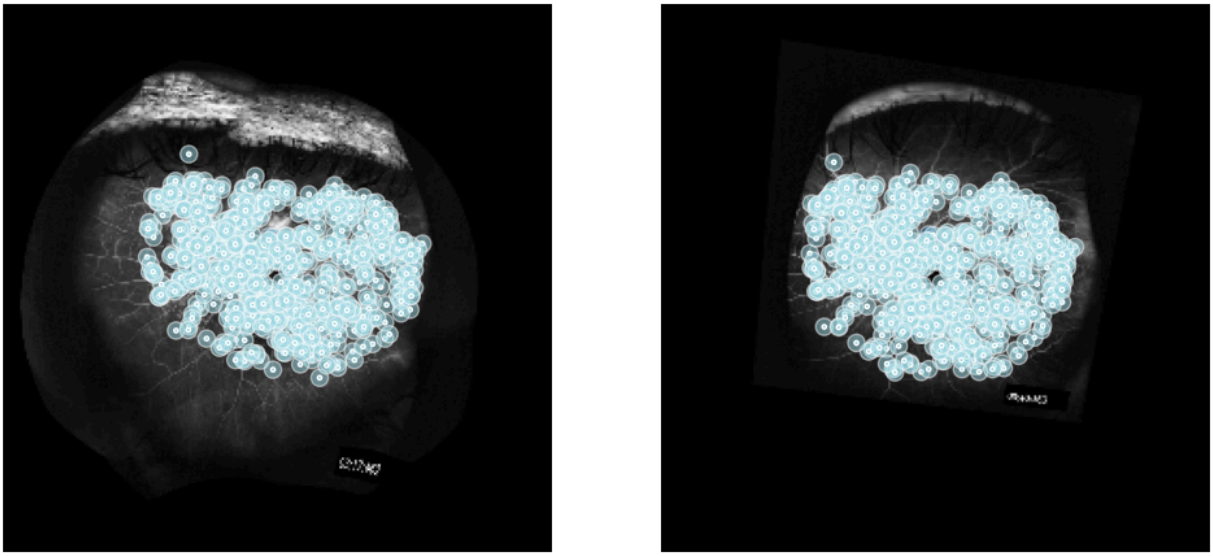
Stage-1 Results: Registration Using Homography Transformation




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

Stage-2 Point Correspondences

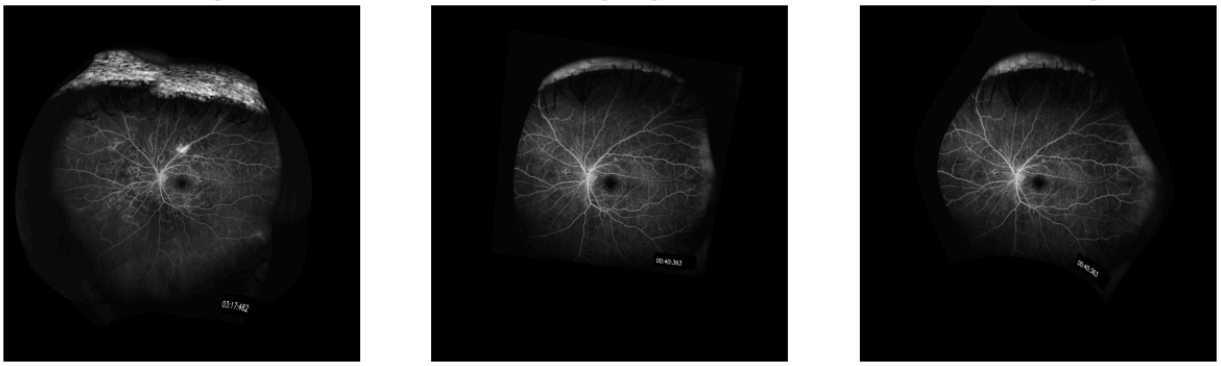
Fixed Image | Moving Image



Note: 749 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



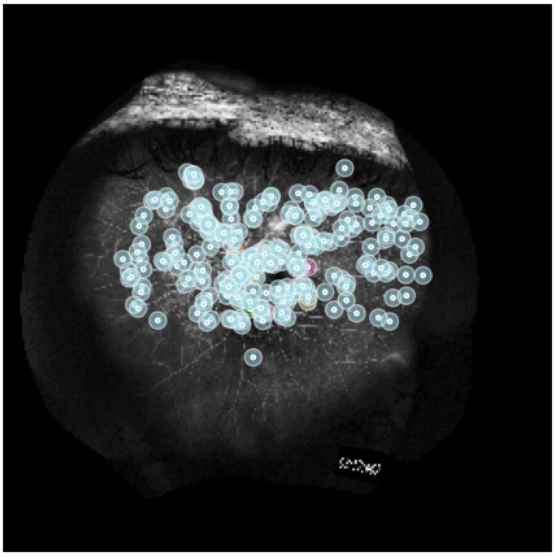
Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

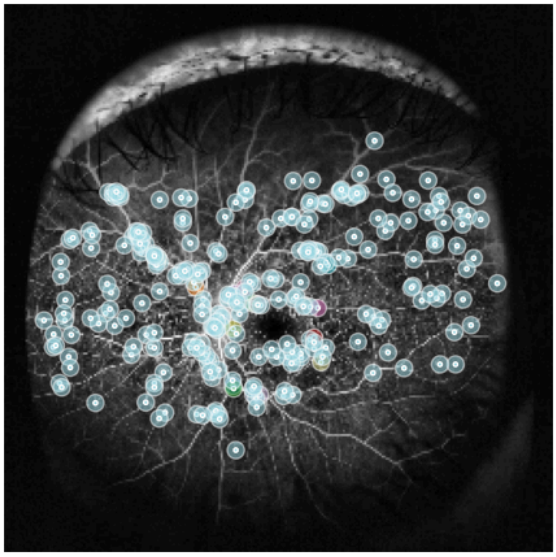
```
Mean Landmark Error for Case 5 Before Registration is 674.9261885180194 pixels
Mean Landmark Error for Case 5 After Registration is 12.354815049599154 pixels
Case 6
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/FA/Raw_FA_4
_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: 261 point correspondences were identified by the model for stage-1

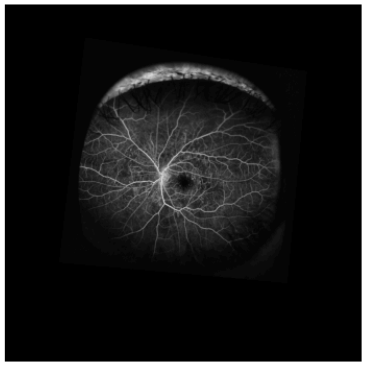
```
Homography Matrix:
[[ 6.77740095e-01 -7.61602923e-02  2.29886976e+02]
 [ 1.06835023e-01  6.15069521e-01  9.41585136e+01]
 [ 6.37039702e-05 -2.14880547e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

Fixed Image

Moving Image

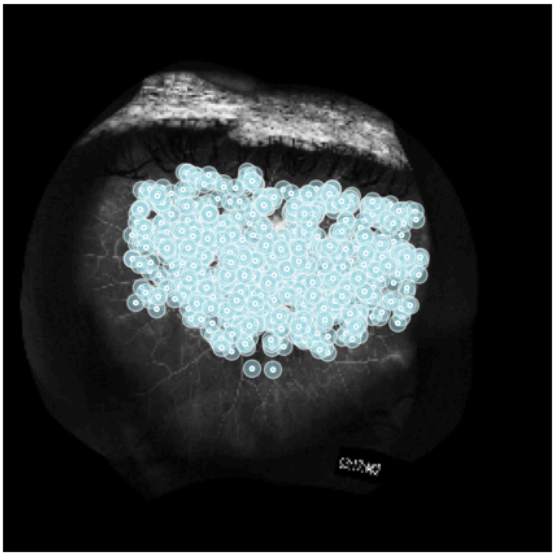
Deformed Image



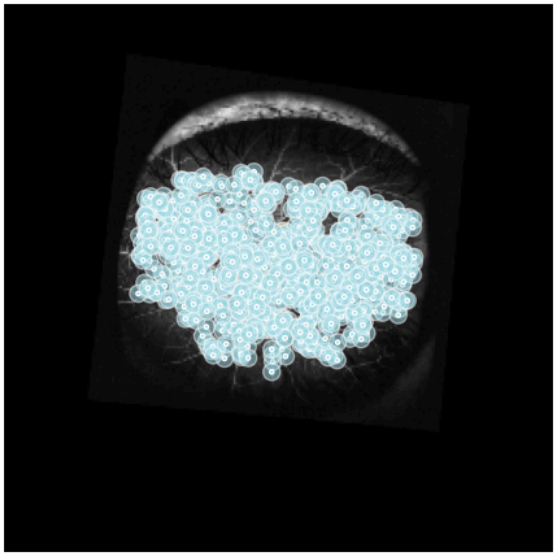
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



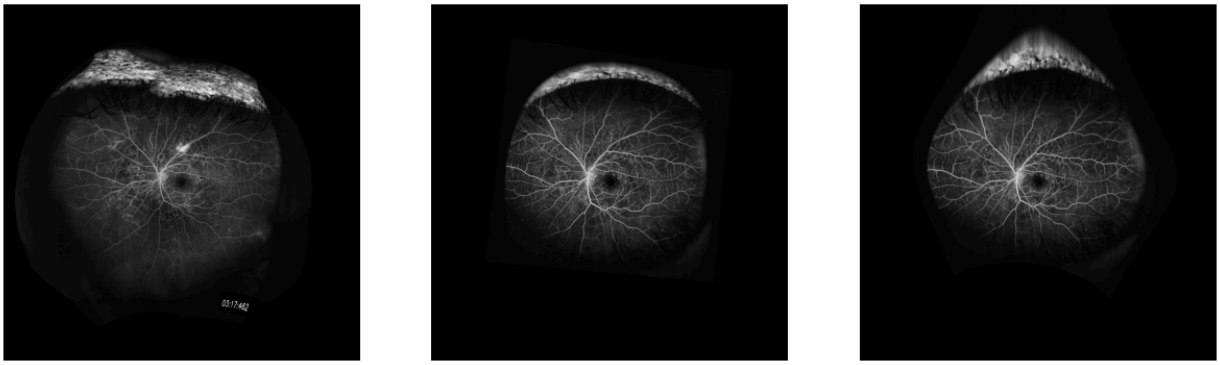
Moving Image



Note: 917 point correspondences were identified by the model for stage-2

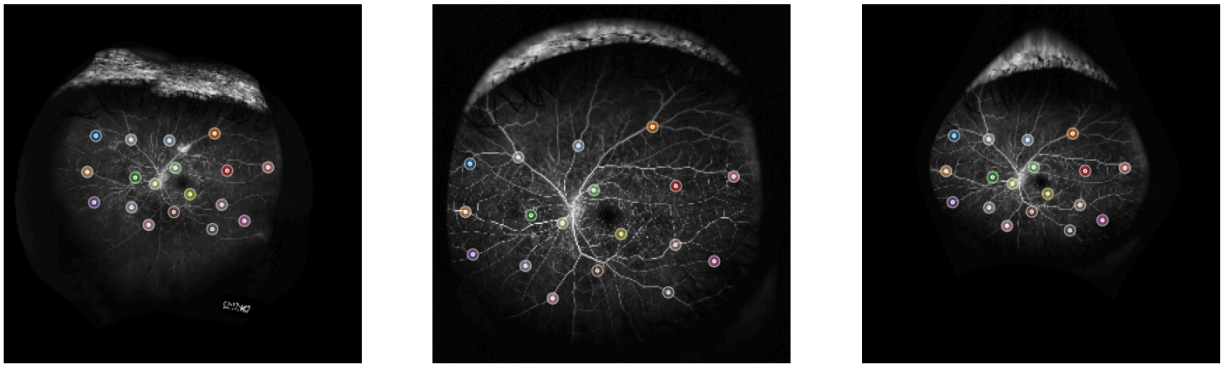
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



Final Registration Results by Composing Transformations Estimated in Two Stages

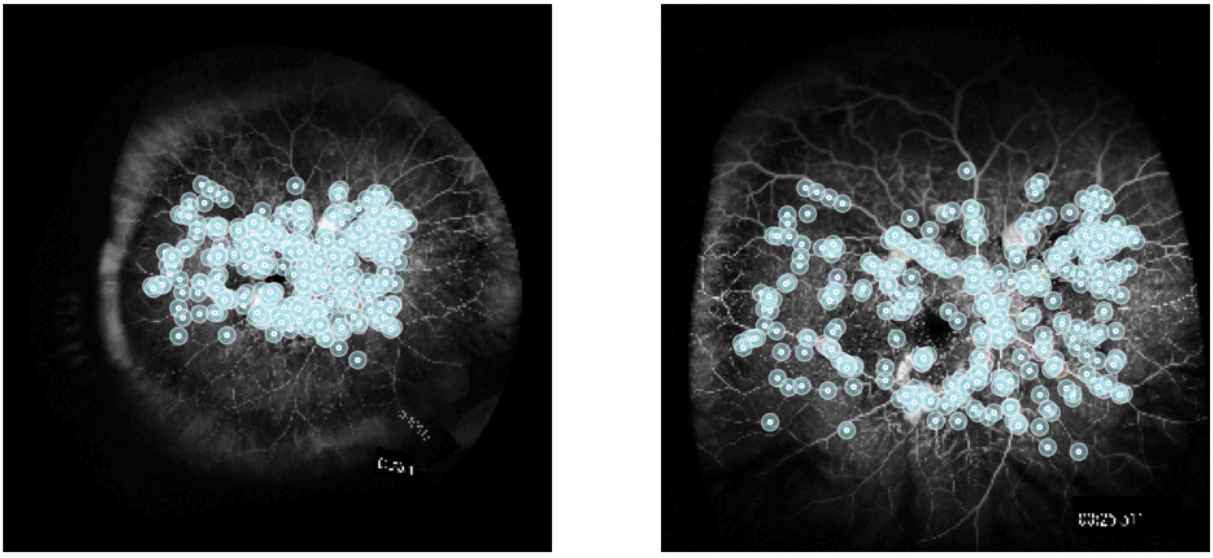
Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 6 Before Registration is 683.3470707329728 pixels
Mean Landmark Error for Case 6 After Registration is 11.595106309499334 pixels
Case 7
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_3/Montage/Montage_Subject_3.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_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

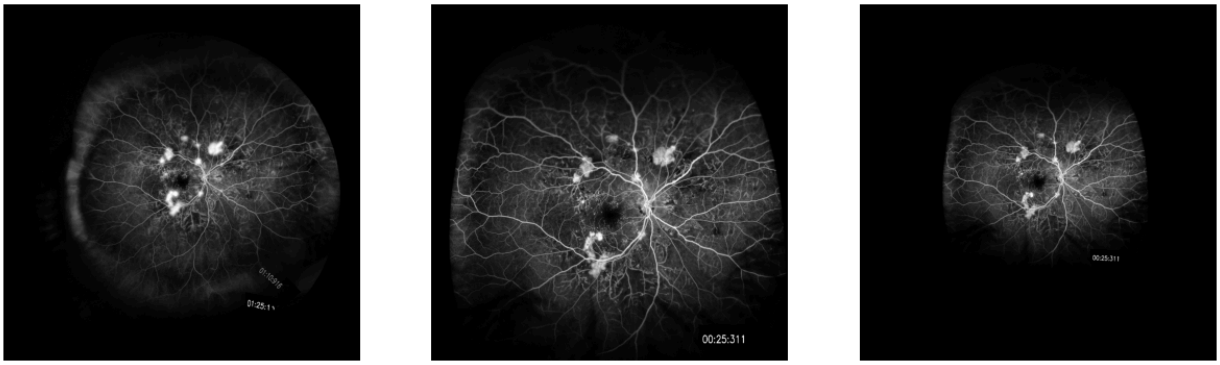


Note: 376 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 5.86266784e-01  2.32611252e-02  2.01888411e+02]
 [-1.74724913e-02  6.44785709e-01  1.34905038e+02]
 [-6.75154054e-05  7.47524937e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

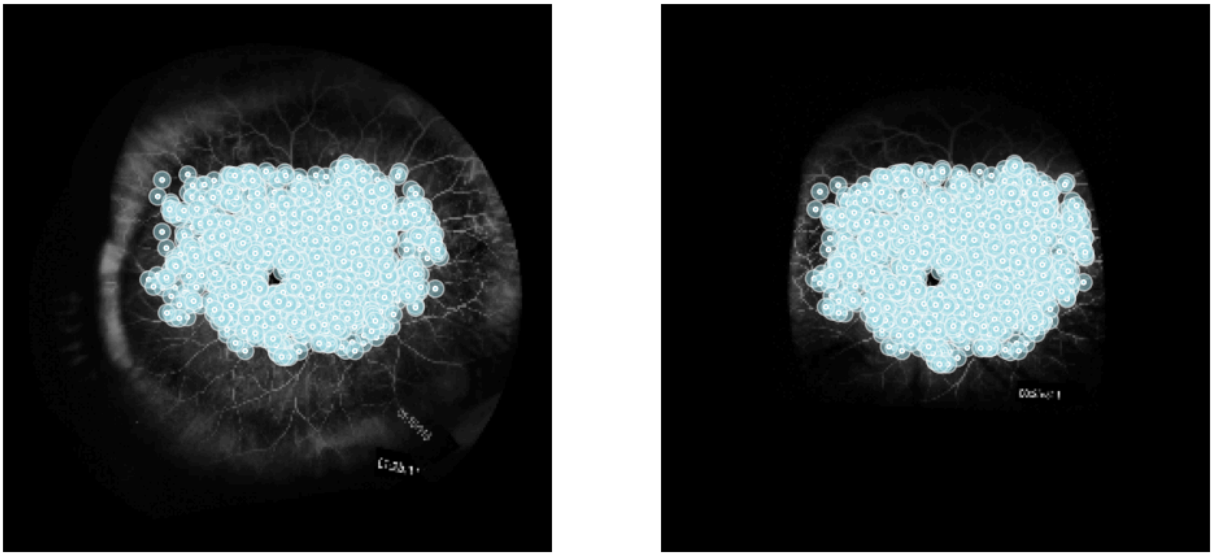
Fixed Image | Moving Image | Deformed Image



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

Stage-2 Point Correspondences

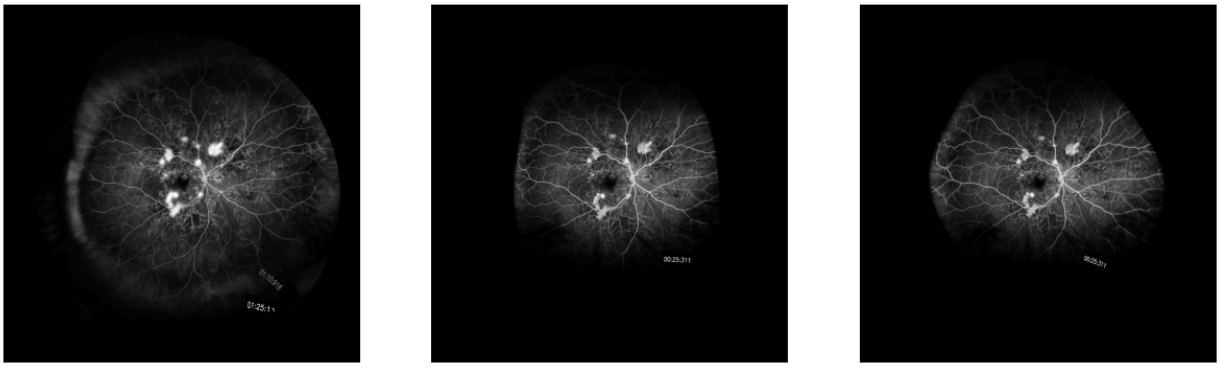
Fixed Image | Moving Image



Note: 1209 point correspondences were identified by the model for stage-2

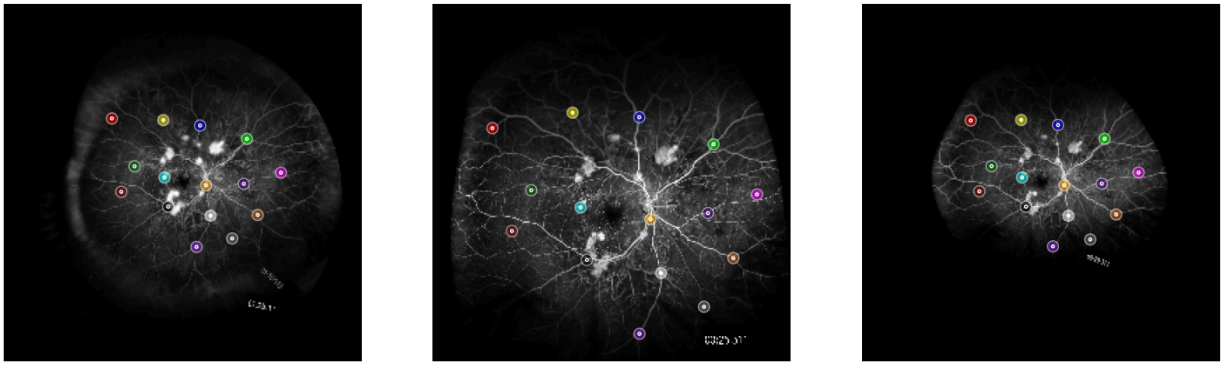
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



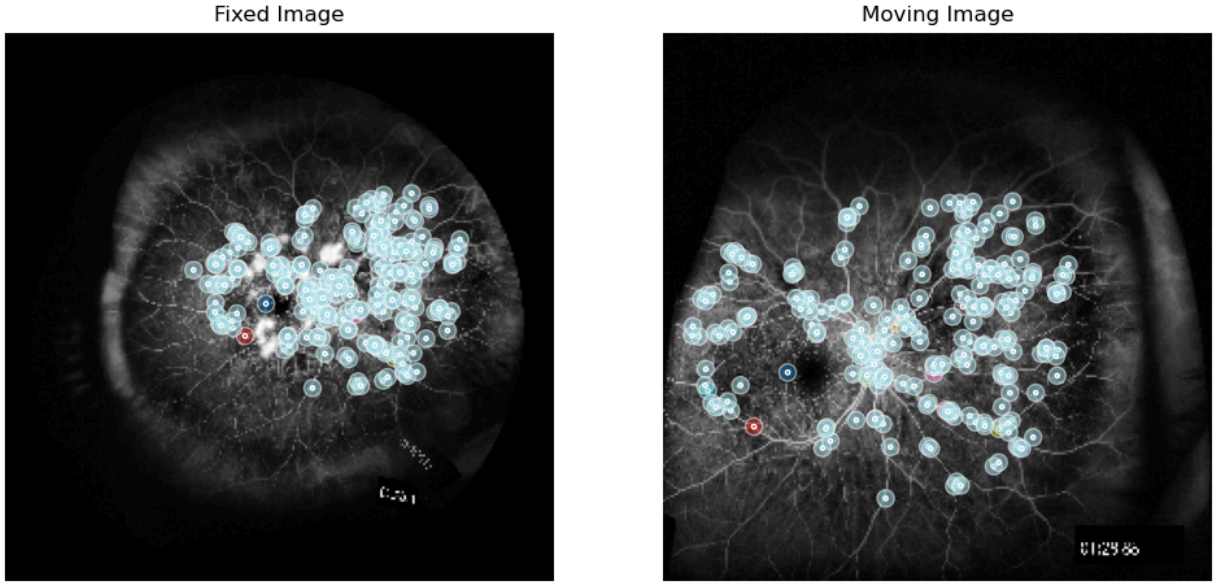
Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 7 Before Registration is 666.06809535934 pixels
Mean Landmark Error for Case 7 After Registration is 21.023304980266293 pixels
Case 8
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_3/Montage/Montage_Subject_3.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_3/FA/Raw_FA_2
_Subject_3.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

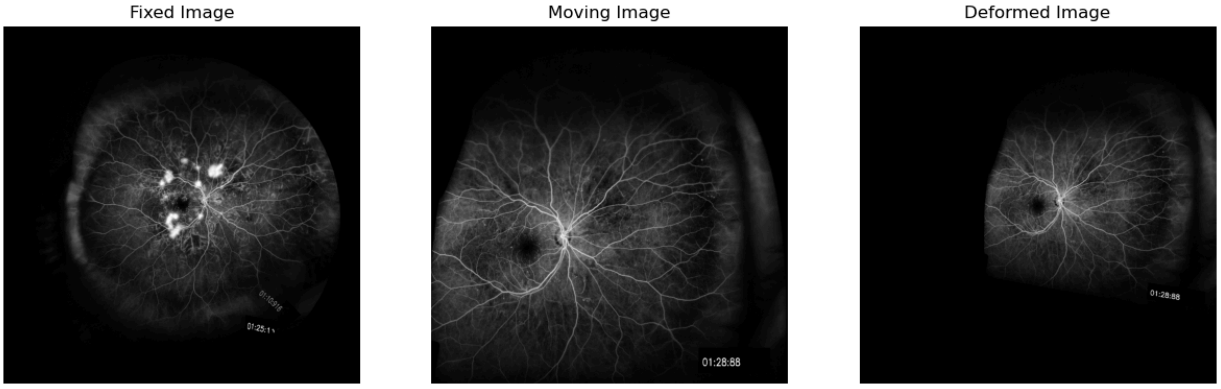
Stage-1 Point Correspondences



Note: 252 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 4.56767219e-01  3.76091021e-02  3.57921320e+02]
 [-9.05389034e-02  6.65548863e-01  1.16057924e+02]
 [-2.65972677e-04  1.10806301e-04  1.00000000e+00]]
```

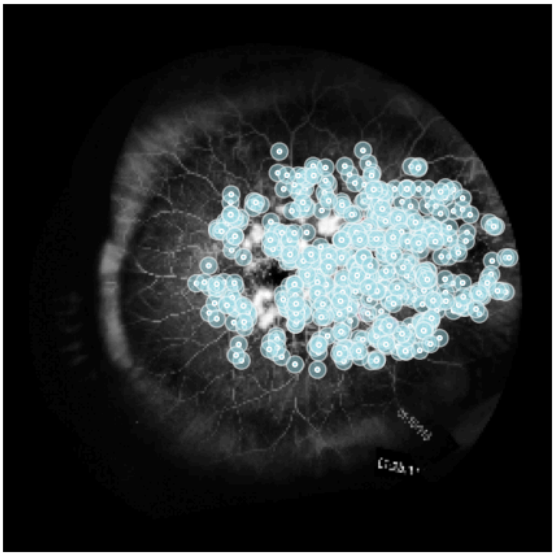
Stage-1 Results: Registration Using Homography Transformation



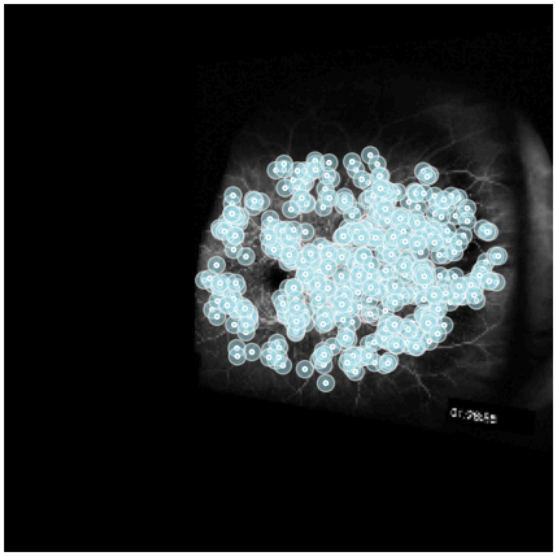

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



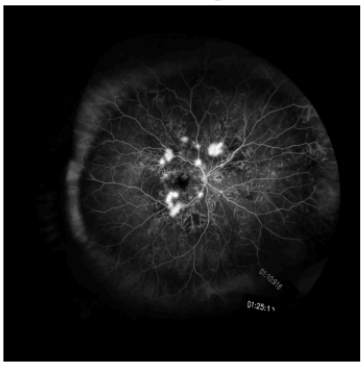
Moving Image



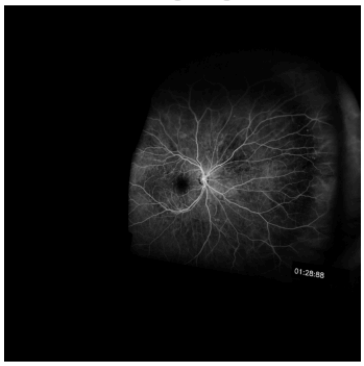
Note: 581 point correspondences were identified by the model for stage-2

Stage-2 Results: Registration Using Third Order Polynomial Transformation

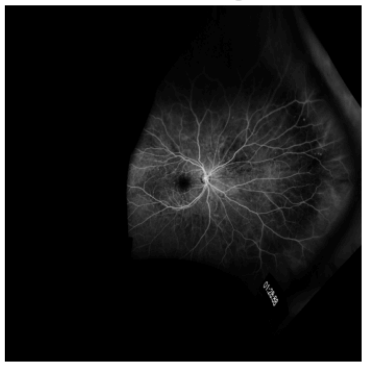
Fixed Image



Moving Image

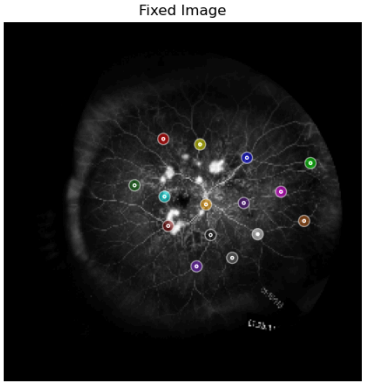


Deformed Image

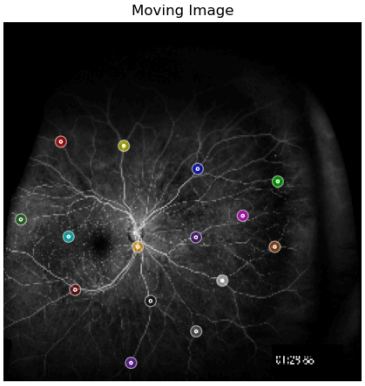


Final Registration Results by Composing Transformations Estimated in Two Stages

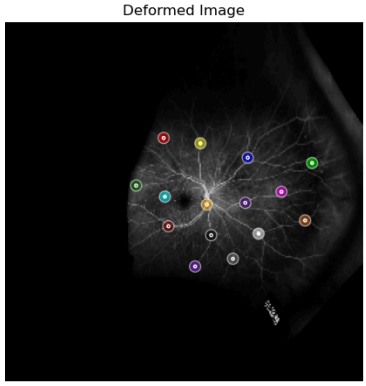
Fixed Image



Moving Image



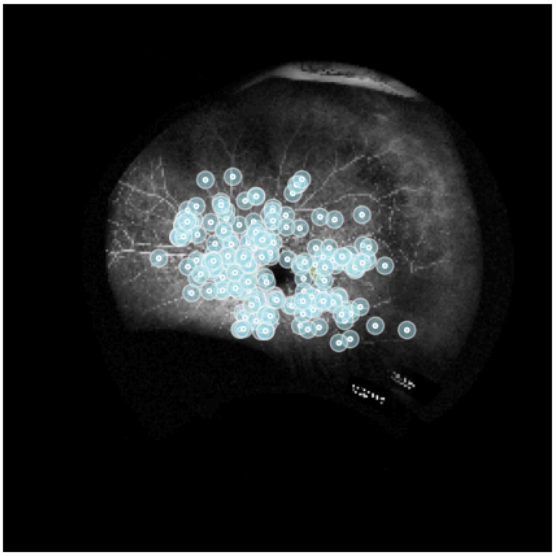
Deformed Image



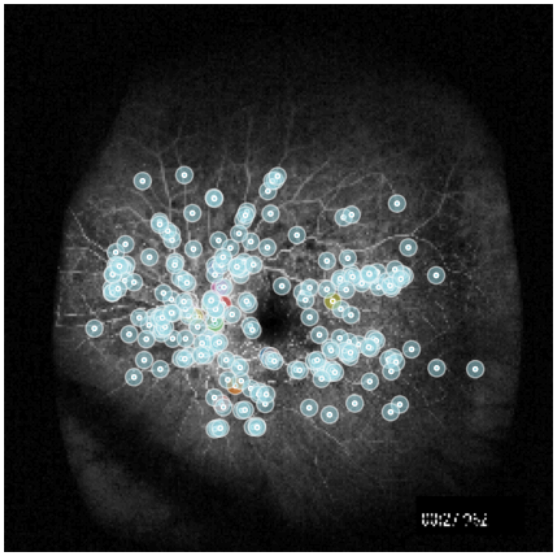
```
Mean Landmark Error for Case 8 Before Registration is 987.7230206941801 pixels
Mean Landmark Error for Case 8 After Registration is 10.932752420657065 pixels
Case 9
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_4/FA/Raw_FA_1
_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image



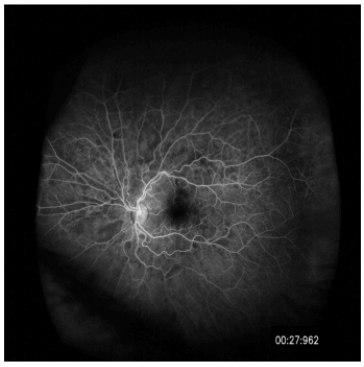
Note: 226 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 7.50158345e-01 -2.34947734e-03  2.22365316e+02]
 [ 1.73656340e-01  7.63524882e-01  4.46639115e+01]
 [ 9.35432531e-05  1.79916525e-04  1.00000000e+00]]
```

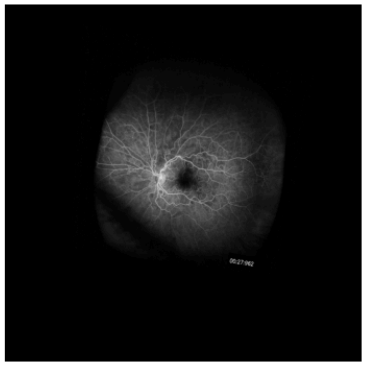
Stage-1 Results: Registration Using Homography Transformation

Fixed Image

Moving Image



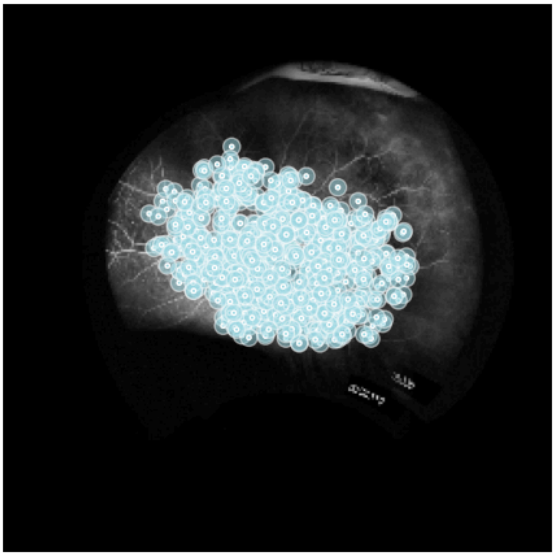
Deformed Image



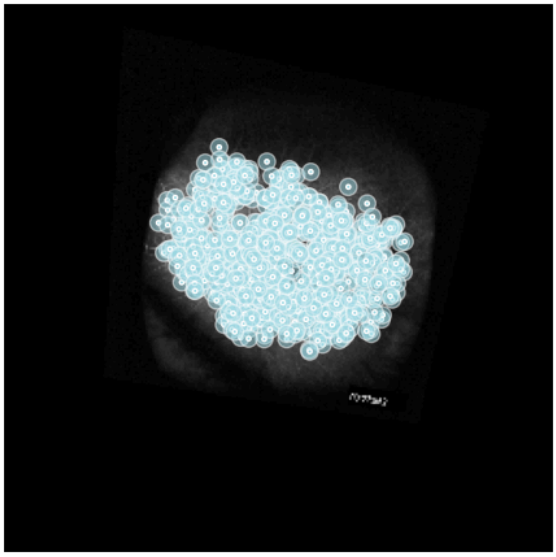
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



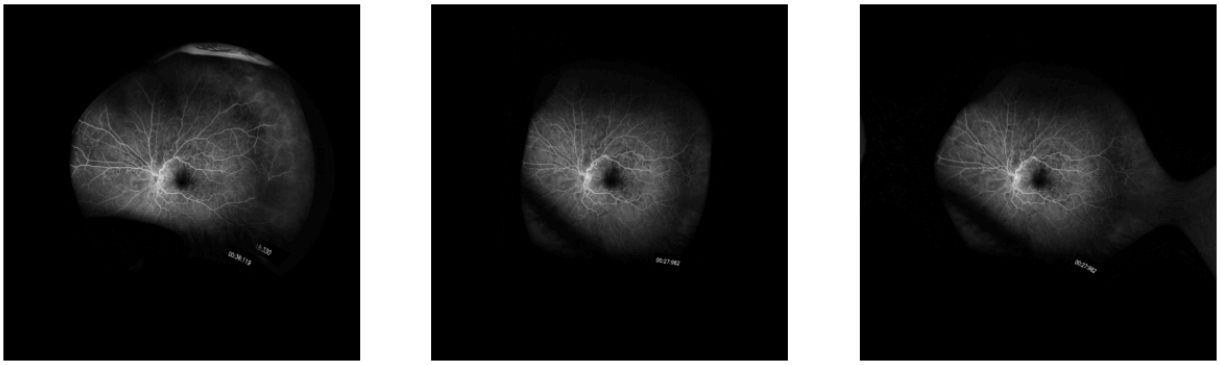
Moving Image



Note: 793 point correspondences were identified by the model for stage-2

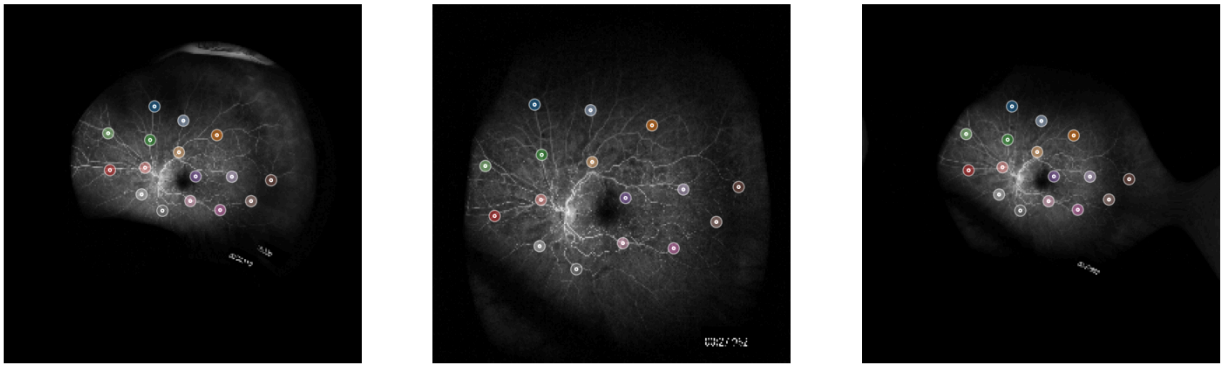
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



Final Registration Results by Composing Transformations Estimated in Two Stages

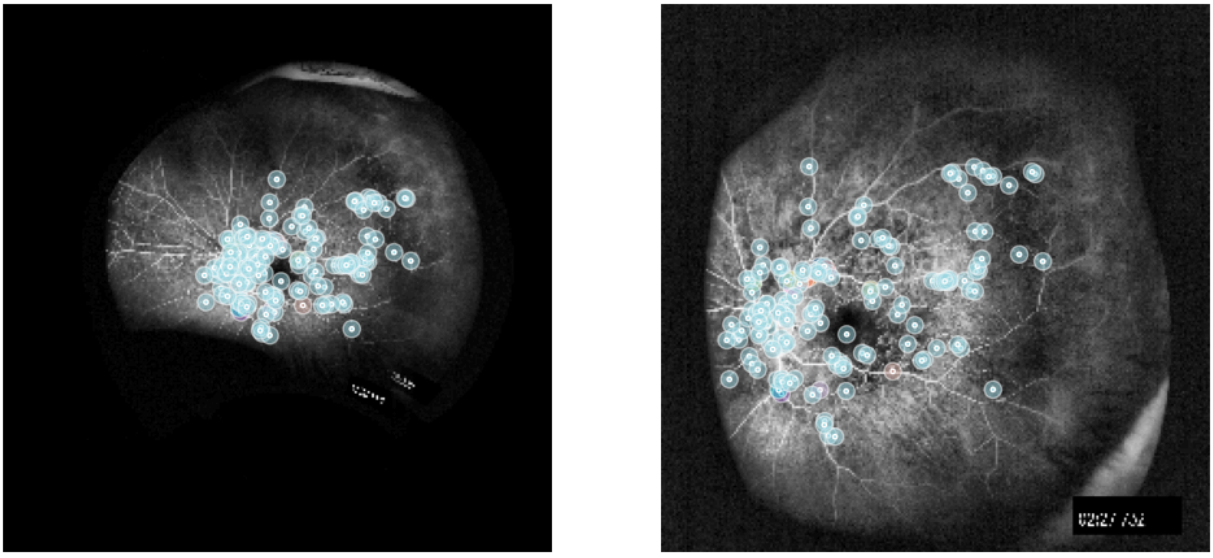
Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 9 Before Registration is 696.7319535892544 pixels
Mean Landmark Error for Case 9 After Registration is 12.374749030257451 pixels
Case 10
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_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

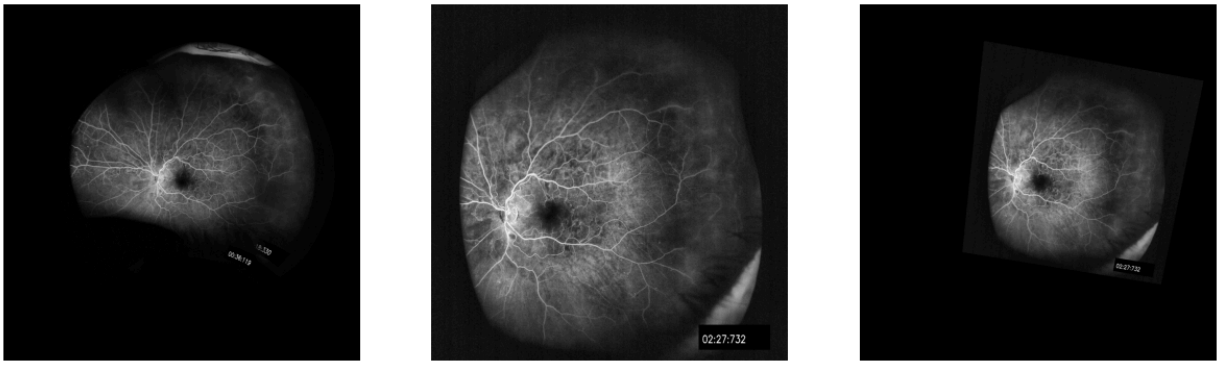


Note: 151 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 6.36596039e-01 -2.94459096e-02  3.55793733e+02]
 [ 1.17222539e-01  6.70205996e-01  1.04215847e+02]
 [ 2.09126694e-05  1.06886169e-04  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

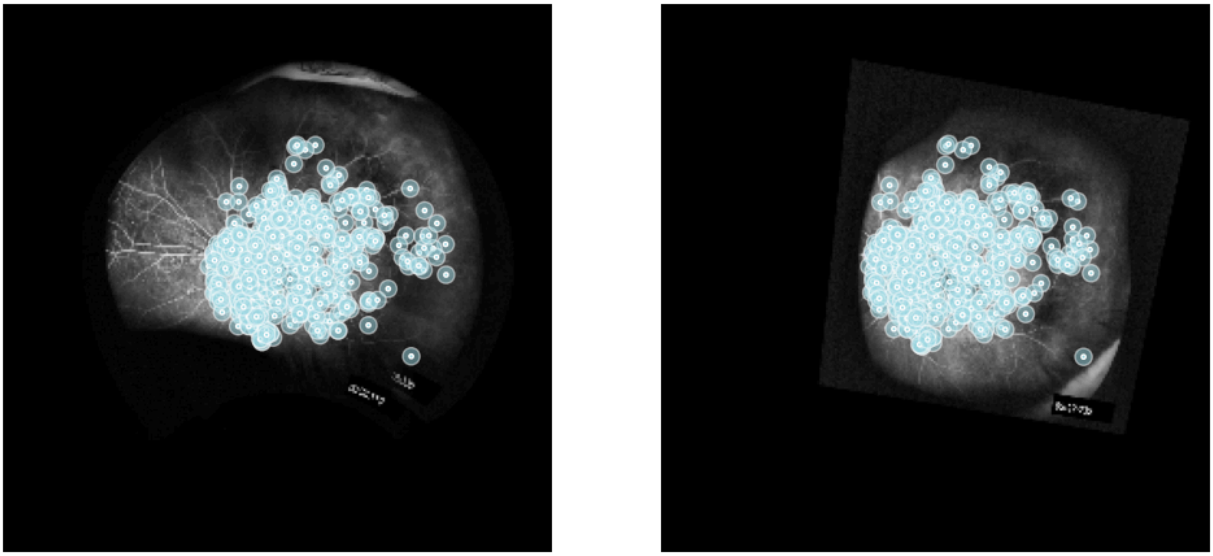
Fixed Image | Moving Image | Deformed Image



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

Stage-2 Point Correspondences

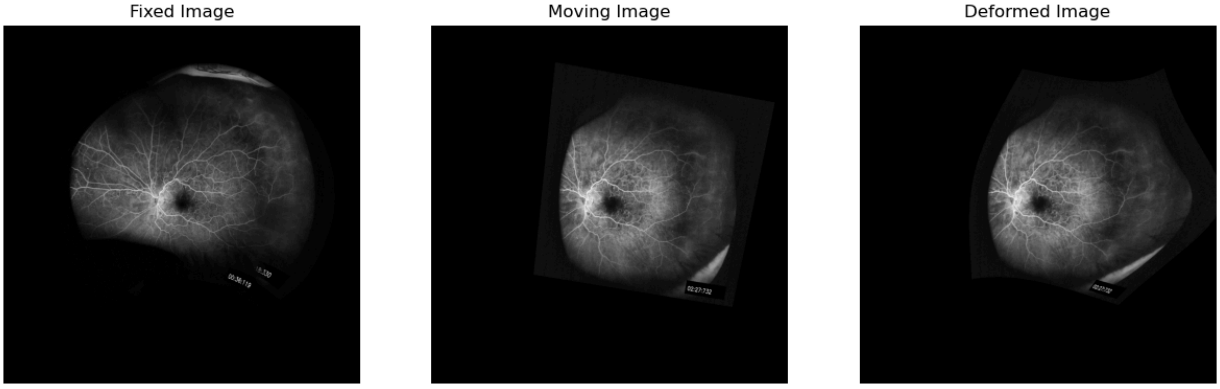
Fixed Image | Moving Image



Note: 484 point correspondences were identified by the model for stage-2

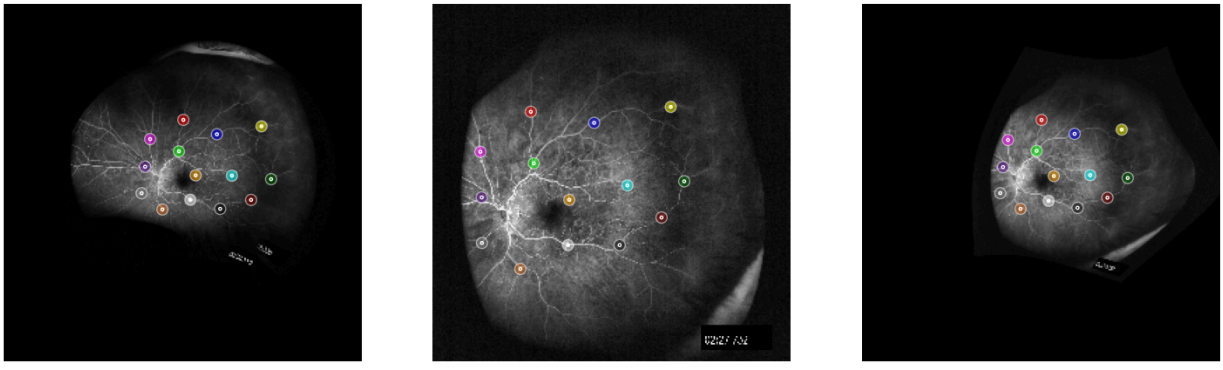
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image

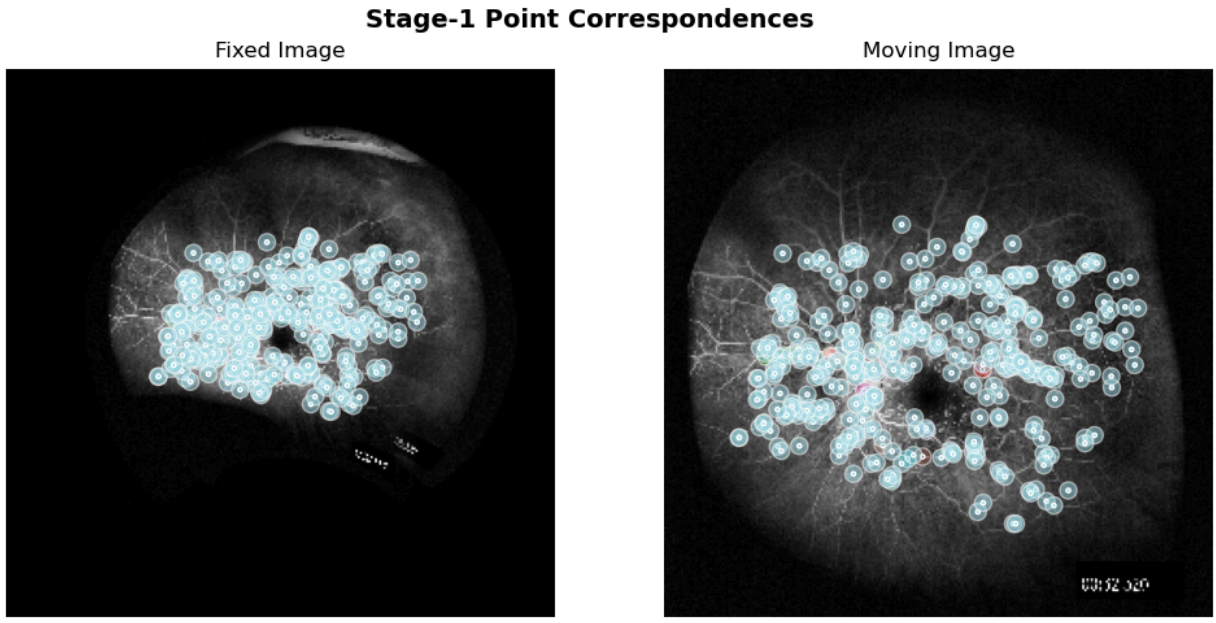


Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

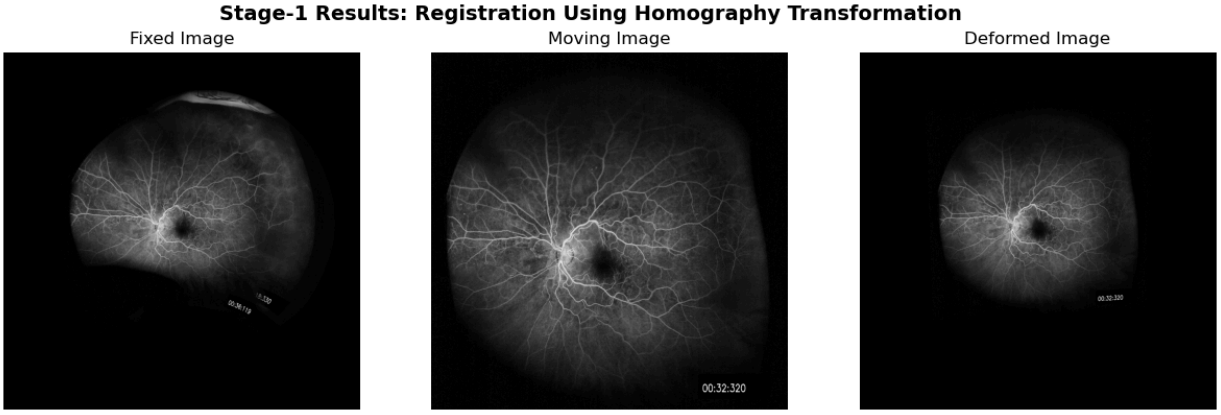


```
Mean Landmark Error for Case 10 Before Registration is 980.0529717789981 pixels
Mean Landmark Error for Case 10 After Registration is 15.429902773538695 pixels
Case 11
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_4/FA/Raw_FA_3
_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 374 point correspondences were identified by the model for stage-1

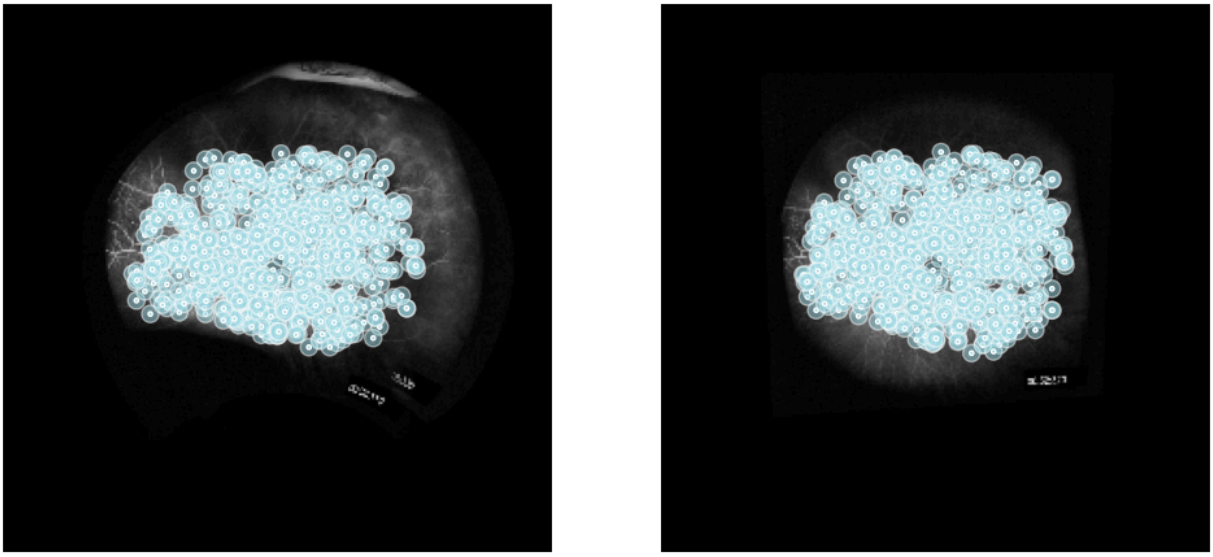
```
Homography Matrix:
[[7.00956909e-01 3.57341489e-02 1.85859771e+02]
 [7.91973610e-03 6.77720726e-01 1.33493265e+02]
 [6.86558125e-05 7.05954944e-05 1.00000000e+00]]
```




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

Stage-2 Point Correspondences

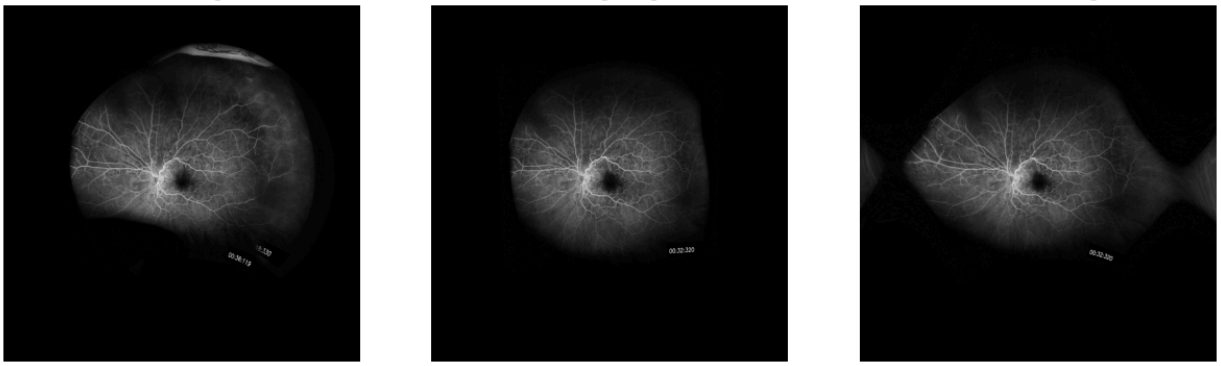
Fixed Image | Moving Image



Note: 822 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



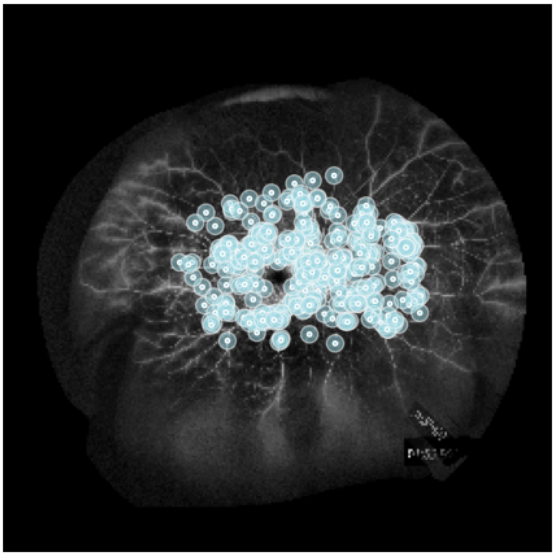
Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

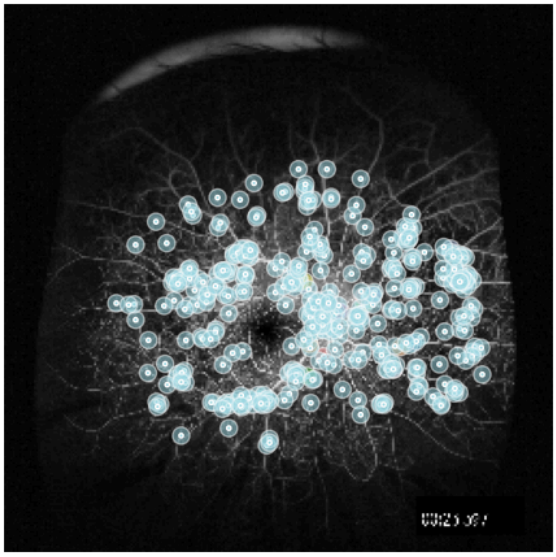
```
Mean Landmark Error for Case 11 Before Registration is 687.6665246743283 pixels
Mean Landmark Error for Case 11 After Registration is 14.088927304236796 pixels
Case 12
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_5/FA/Raw_FA_1
_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-1 Point Correspondences

Fixed Image



Moving Image

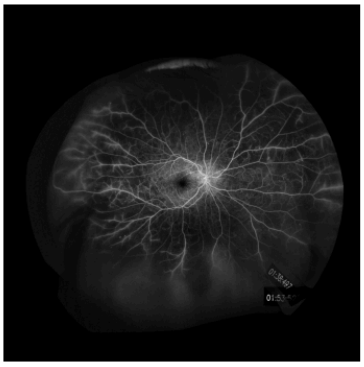


Note: 372 point correspondences were identified by the model for stage-1

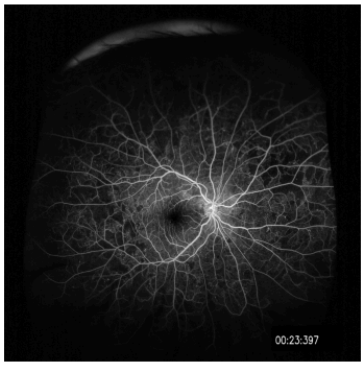
```
Homography Matrix:
[[ 5.02259180e-01  4.29085442e-02  2.17152739e+02]
 [-8.69475577e-02  6.00530301e-01  1.68693807e+02]
 [-1.74306250e-04  7.19869224e-05  1.00000000e+00]]
```

Stage-1 Results: Registration Using Homography Transformation

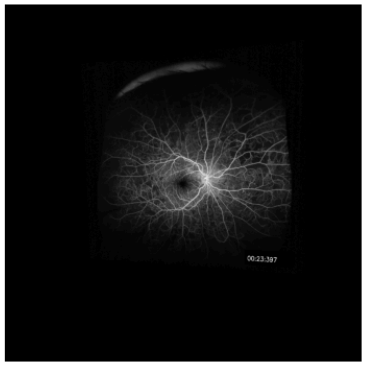
Fixed Image



Moving Image



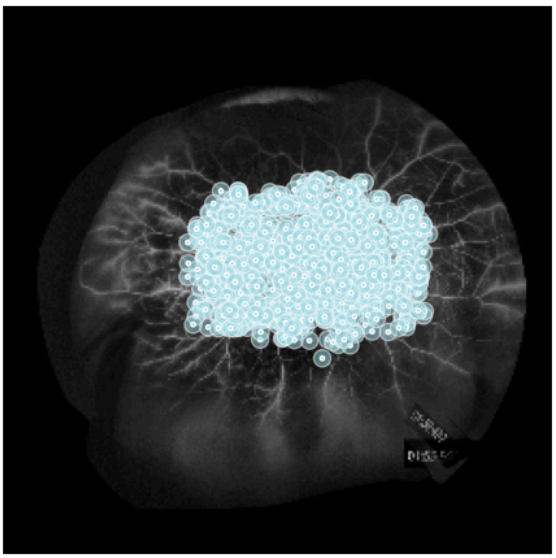
Deformed Image



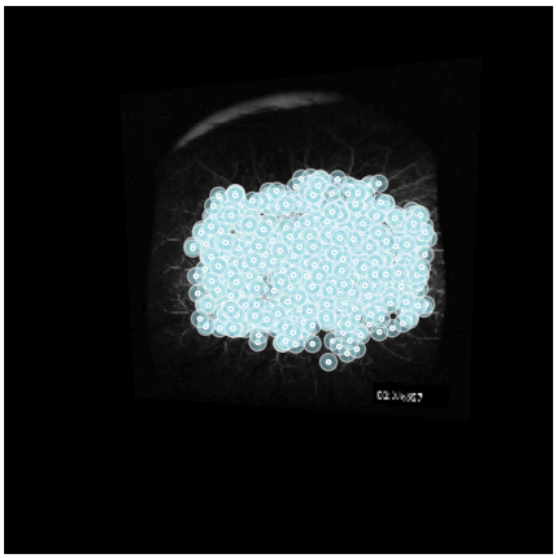
```
Maximum attempts reached, unable to find sufficient points with the specified criteria.
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

Stage-2 Point Correspondences

Fixed Image



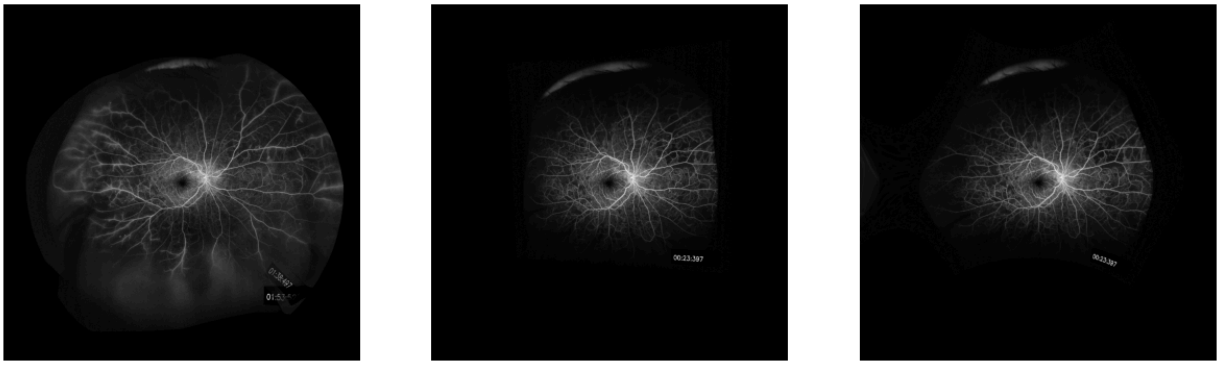
Moving Image



Note: 1088 point correspondences were identified by the model for stage-2

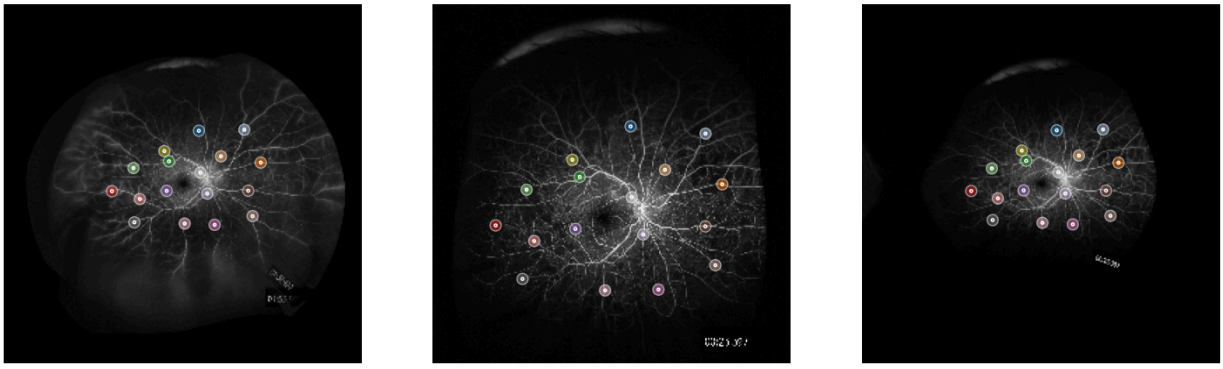
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image



Final Registration Results by Composing Transformations Estimated in Two Stages

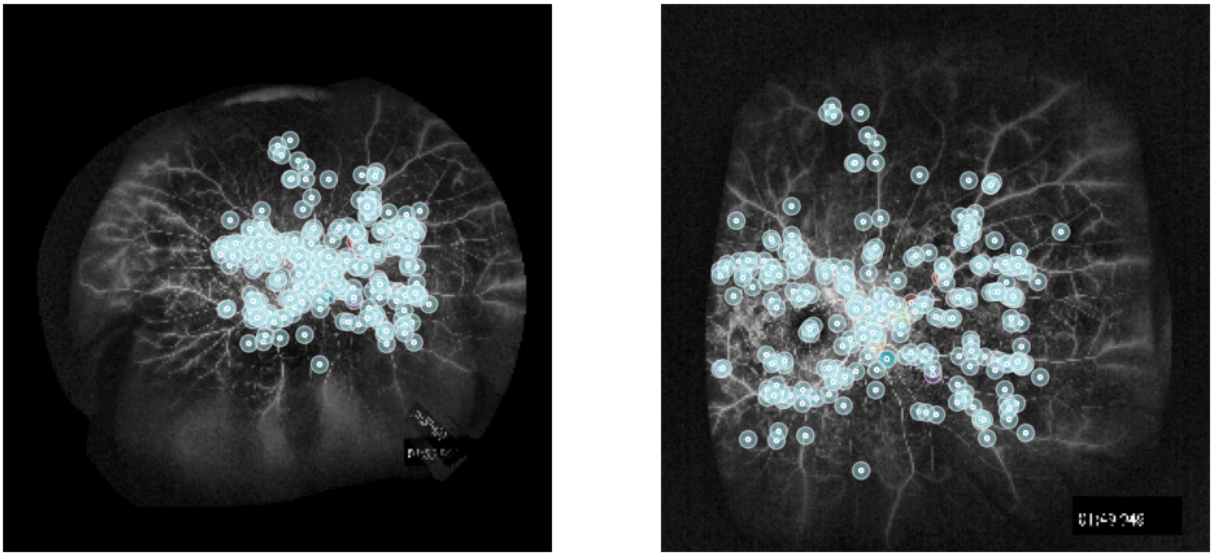
Fixed Image | Moving Image | Deformed Image



```
Mean Landmark Error for Case 12 Before Registration is 664.0966369010016 pixels
Mean Landmark Error for Case 12 After Registration is 10.14869896003407 pixels
Case 13
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_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: 274 point correspondences were identified by the model for stage-1

```
Homography Matrix:
[[ 4.30444461e-01  5.42791634e-02  3.28267130e+02]
 [-1.57721250e-01  5.61156169e-01  1.86781013e+02]
 [-2.32916956e-04  1.14585890e-05  1.00000000e+00]]
```

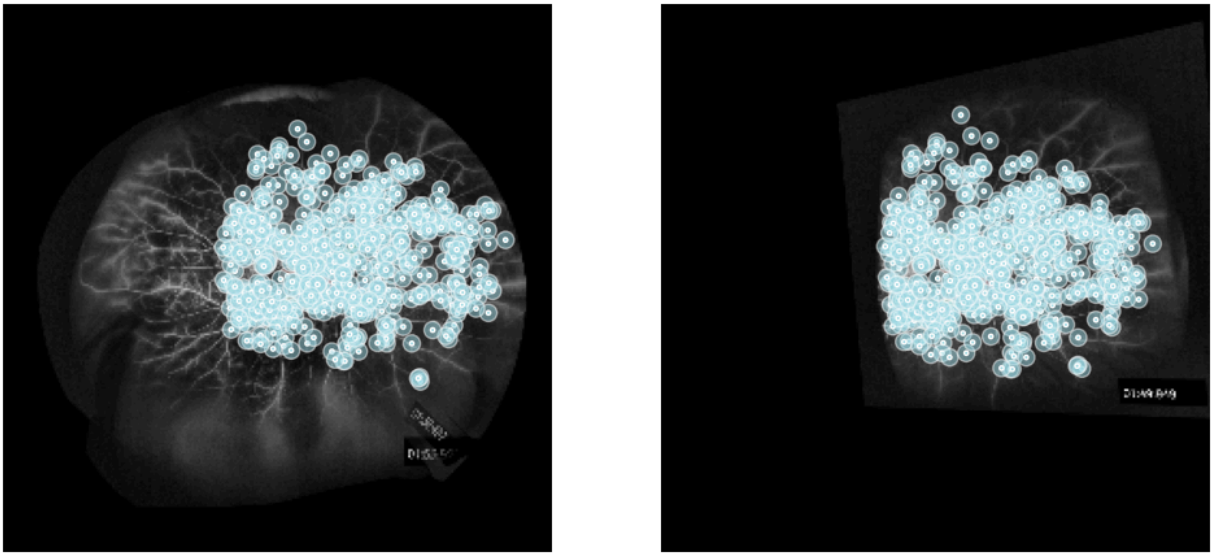
Stage-1 Results: Registration Using Homography Transformation

Fixed Image | Moving Image | Deformed Image

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

Stage-2 Point Correspondences

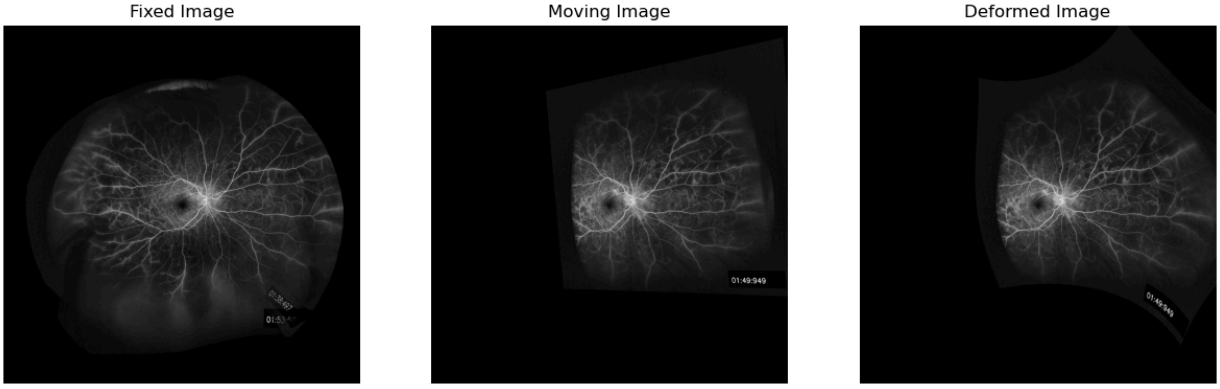
Fixed Image | Moving Image



Note: 593 point correspondences were identified by the model for stage-2

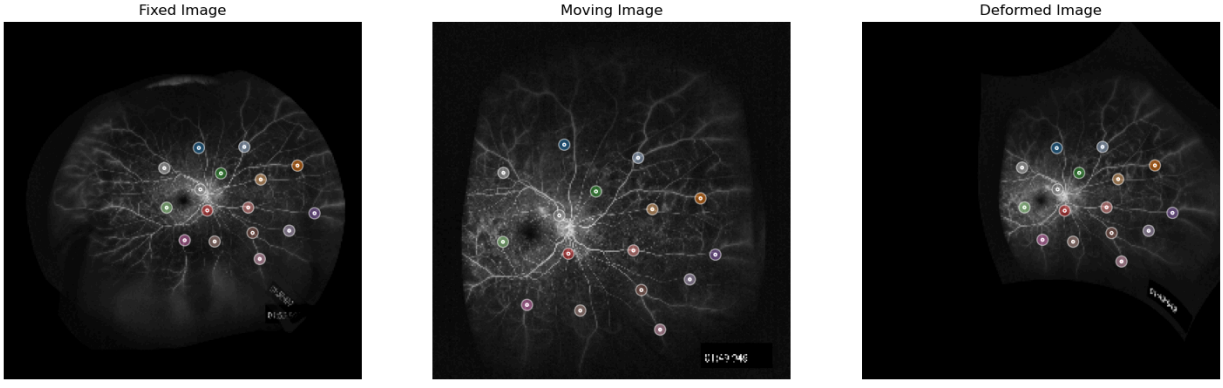
Stage-2 Results: Registration Using Third Order Polynomial Transformation

Fixed Image | Moving Image | Deformed Image

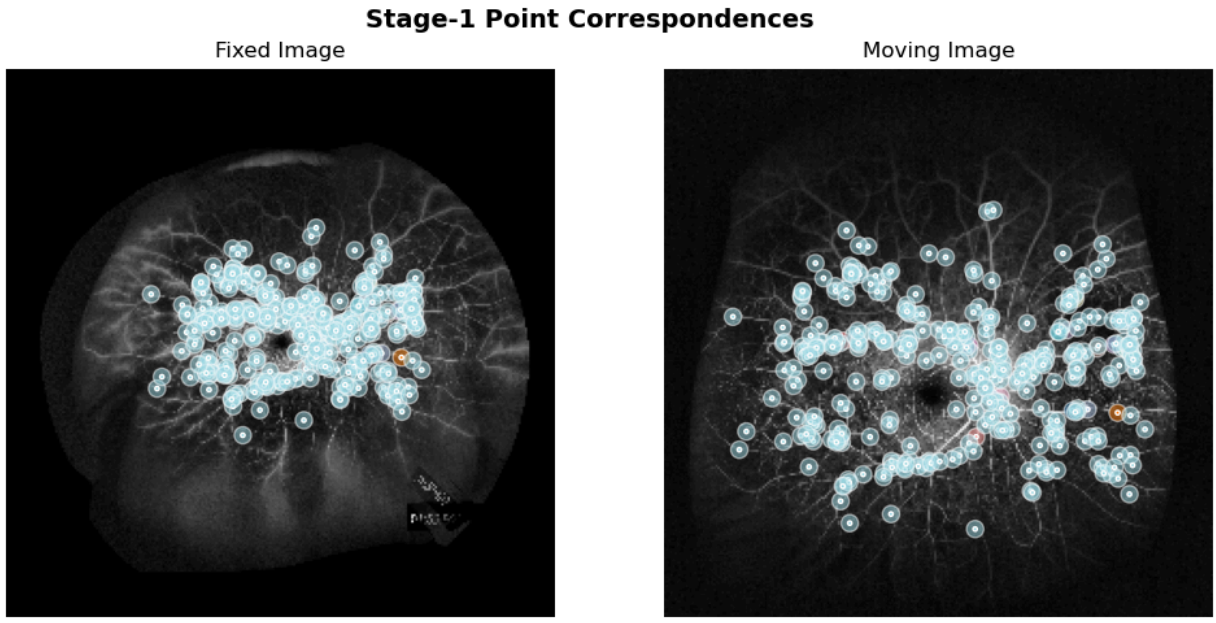


Final Registration Results by Composing Transformations Estimated in Two Stages

Fixed Image | Moving Image | Deformed Image

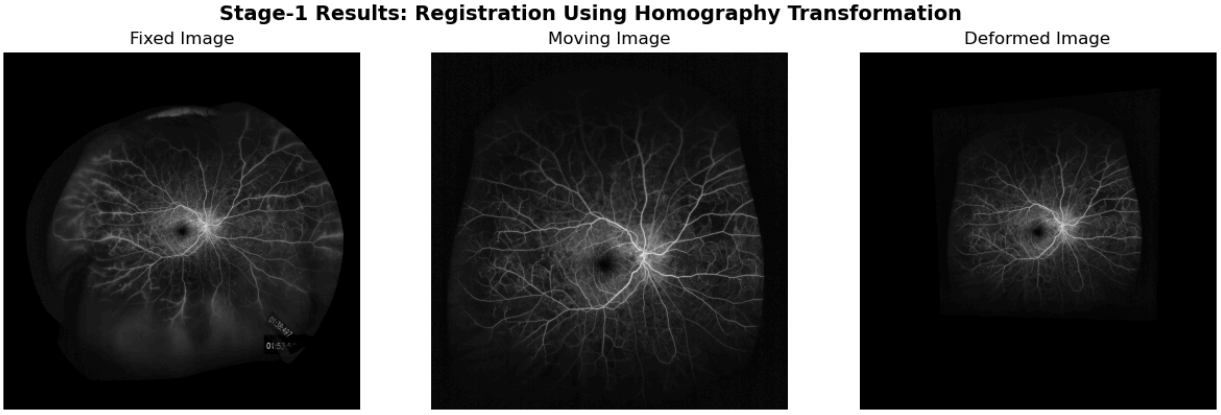


```
Mean Landmark Error for Case 13 Before Registration is 911.4162426368459 pixels
Mean Landmark Error for Case 13 After Registration is 11.942259514201442 pixels
Case 14
Loading Fixed Images /blue/weishao/vi.sivaraman/Tasks/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Regi
stration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif Moving Image/blue/weishao/vi.sivaraman/Task
s/Image_Registration/2D-Registration/RetinaRegNet/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_5/FA/Raw_FA_3
_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```



Note: 345 point correspondences were identified by the model for stage-1

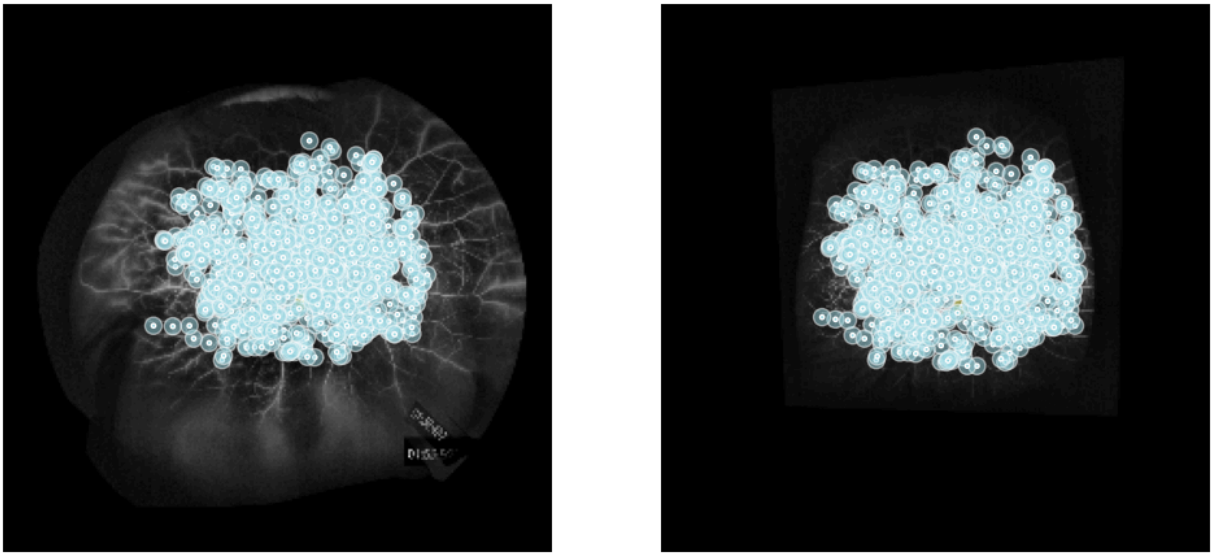
```
Homography Matrix:
[[ 5.41266463e-01  3.80911478e-02  2.05871494e+02]
 [-7.15149844e-02  6.18030040e-01  1.62308720e+02]
 [-1.17037118e-04  5.57401212e-05  1.00000000e+00]]
```




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

Stage-2 Point Correspondences

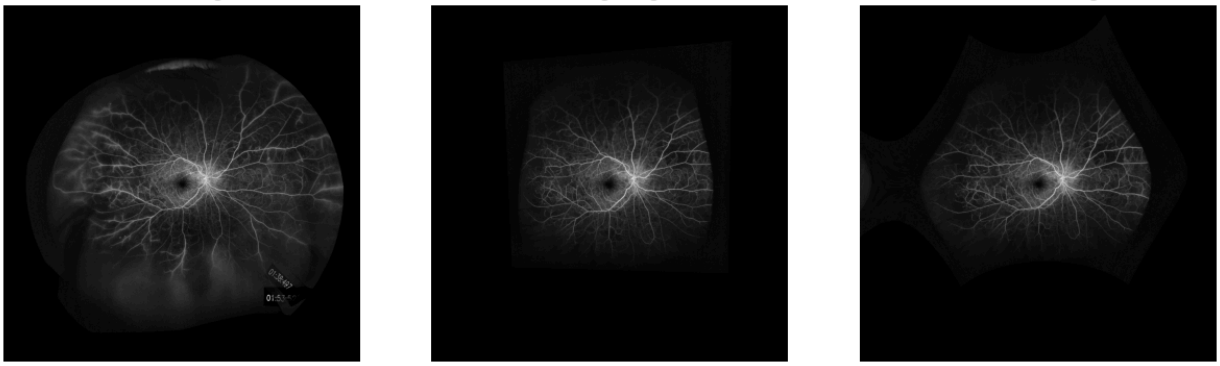
Fixed Image | Moving Image



Note: 851 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

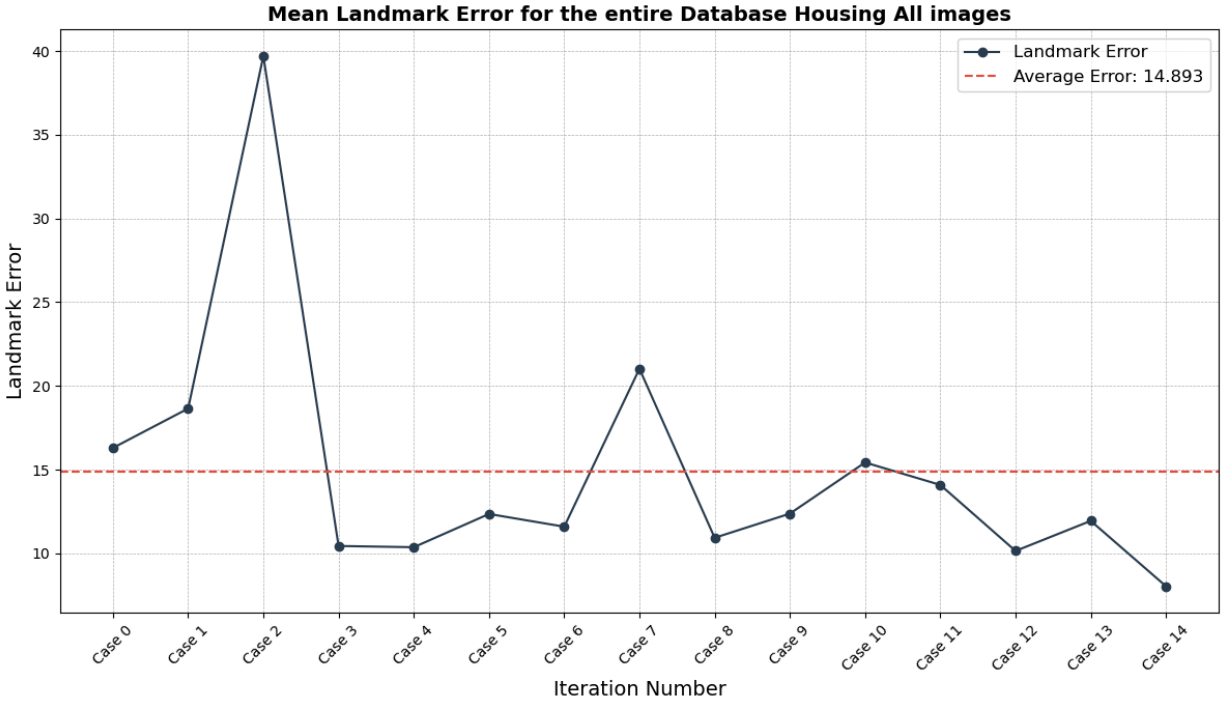


Final Registration Results by Composing Transformations Estimated in Two Stages

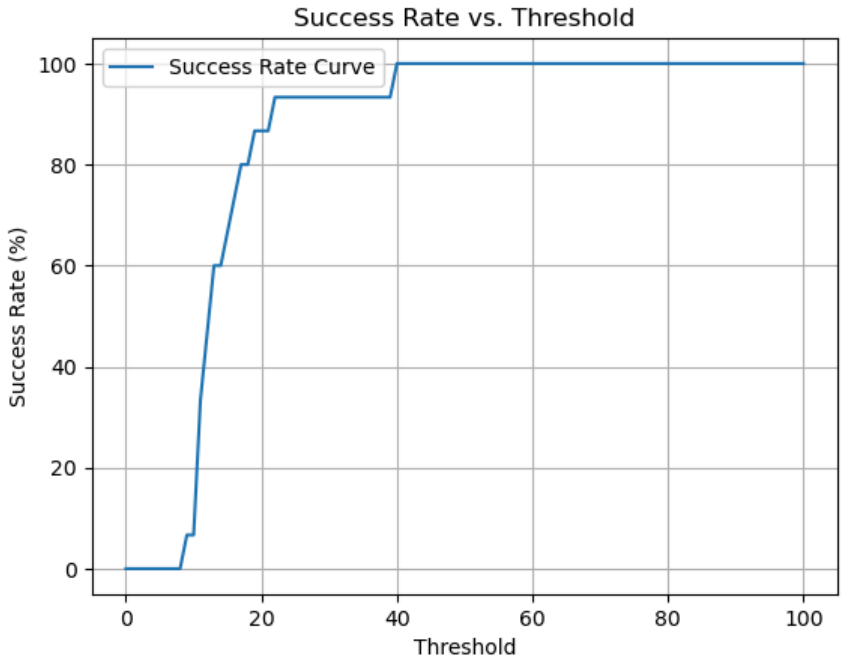
Fixed Image | Moving Image | Deformed Image

```
Mean Landmark Error for Case 14 Before Registration is 669.1965551074313 pixels
Mean Landmark Error for Case 14 After Registration is 8.040423289539016 pixels
```

In [20]:
```
plot_landmark_errors(landmark_errors,os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results'),'Al
```



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




```
AUC: 0.8553333333333332
```