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
```

```
2024-01-30 15:08:13.536613: I tensorflow/core/platform/cpu_feature_guard.cc:182] Thi
s TensorFlow binary is optimized to use available CPU instructions in performance-cr
itical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operation
s, rebuild TensorFlow with the appropriate compiler flags.
```

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

In [3]:
```
#shutil.rmtree(os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Result
```

In [4]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Homogr
os.makedirs(path,exist_ok=True)
```

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

In [6]:
```
path = os.path.join(os.getcwd(),'FLoRI21_DataPort_Image_Registration_Results/Homogr
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={2024},
  url={https://openreview.net/forum?id=ypOiXjdfnU}
}
```

In [7]:

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

    This model extends the original `UNet2DConditionModel` to incorporate additiona
    such as encoder hidden states, attention mask, and cross-attention keyword argu
    """
    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, chan
            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 s
            class_labels (Optional[torch.Tensor], default=None): Class labels tenso
            timestep_cond (Optional[torch.Tensor], default=None): Timestep conditio
            attention_mask (Optional[torch.Tensor], default=None): Mask to avoid at
            cross_attention_kwargs (Optional[dict], default=None): Keyword argument

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

        # By default samples have to be AT least a multiple of the overall upsampli
        # The overall upsampling factor is equal to 2 ** (# num of upsampling layea
        # However, the upsampling interpolation output size can be forced to fit an
        # 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 `defau
        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 timestep
            # 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
        # 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_em

            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_bloc
                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=e

        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.r

        # 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.ha
            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_sam
            )

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

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

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

    Provides a one-step stable diffusion process, integrating the VAE encoding and
    """
    @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 gene
            prompt_embeds (Optional[torch.FloatTensor], default=None): Precomputed
            callback (Optional[Callable], default=None): Callback function invoked
            callback_steps (int, default=1): Frequency of invoking the callback.
            cross_attention_kwargs (Optional[dict], default=None): Keyword argument

        Returns:
            dict: Dictionary containing output from U-Net.
        """
        device = self._execution_device
        latents = self.vae.encode(img_tensor).latent_dist.sample() * self.vae.confi
        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 i
```

```
    """
    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 diffusi
        """
        unet = MyUNet2DConditionModel.from_pretrained(sd_id, subfolder="unet")
        onestep_pipe = OneStepSDPipeline.from_pretrained(sd_id, unet=unet, safety_c
        onestep_pipe.vae.decoder = None
        onestep_pipe.scheduler = DDIMScheduler.from_pretrained(sd_id, subfolder="sc
        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,
            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 averag

        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
        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 [8]:
```
class DFT:
    """
```

```
    RetinaRegNet (RetinaRegNetwork) utilizes DFT (Diffusion Features) for identifyi
    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

        This function mimics numpy's `unravel_index` functionality, which is used t
        into a tuple of coordinate indices for an array of given shape. This is use
        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 d

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

        Example:
            >>> obj = MyClass()
            >>> obj.unravel_index(22, (5, 5))
            (4, 2)
            >>> obj.unravel_index(52, (7, 8))
            (6, 4)

        Note:
            This function operates under the assumption that indexing starts from 0
        """
        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
        """
        Compute pooled and stacked feature maps.

        Parameters:
        - feature_map (torch.Tensor): Input feature map.
        - hierarchy_range (int, optional): Depth of hierarchical pooling. Defaults
        - 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
        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_m
    """
    Computes 2D correlation maps between selected points in one feature map and

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

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

    Returns:
        torch.Tensor: A tensor of shape (NumPoints, H2, W2) where each slice co
                      for each point in `pts_list`.
    Notes:
        The function assumes that the first dimension of feature_map1 and featu
        This method uses batch matrix multiplication and vector normalization f
        Running this method on a GPU is recommended due to its computational an
    """
    # Convert the input tensors to float16
    feature_map1 = feature_map1.to(dtype=torch.float16)
    feature_map2 = feature_map2.to(dtype=torch.float16)
    _, C, H, W = feature_map2.shape

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

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

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

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

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

        # 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
        """
        Compute the maximum locations in the batched correlation maps between two f

        Parameters:
        - pts_list (list of tuples): List of points for which the correlation maps
        - 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(feat
        enhanced_feature_map2 = self.compute_pooled_and_combining_feature_maps(feat
        # Compute the batched correlation maps
        batched_correlation_maps = self.compute_batched_2d_correlation_maps(pts_lis

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

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

        x, y = zip(*[self.unravel_index(idx.item(), (H2, W2)) for idx in max_indice
        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='bilinea
            gc.collect()
            torch.cuda.empty_cache()
            trg_ft = nn.Upsample(size=(self.img_size, self.img_size), mode='bilinea
        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

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

        Parameters:
            feature_map1 (torch.Tensor): The first feature map, used as the base fo
            feature_map2 (torch.Tensor): The second feature map, used for reverse c
            iccl (float): The maximum allowed distance (inverse consistency criteri
                          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
            - rmaxs (list of floats): The maximum correlation values at these point
            - rspts (list of tuples): The corresponding points in the second featur
                                      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(
        x_prime_y_prime = max_indices_ST
        max_indices_TS, max_values_TS = self.compute_correlation_map_max_locations(
        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 inde
            if np.sqrt((pt[1] - max_idx.cpu()[0]) ** 2 + (pt[0] - max_idx.cpu()[1])
                pnts.append((int(pt[0]), int(pt[1])))
                rmaxs.append(max_values_ST[i].cpu().item())  # Assuming max_values_
                rspts.append((x_prime_y_prime[i][1].cpu().item(), x_prime_y_prime[i
        return pnts, rmaxs, rspts
```

In [9]:

```
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 inte
    """
    # Compute the upper, lower, left, and right boundary
    upper_boundary = next((i for i, row in enumerate(image) if np.mean(row) > mean_
    lower_boundary = next((i for i, row in enumerate(image[::-1]) if np.mean(row) >

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

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

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 o

    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 2
    - 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 thre
            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
                    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
    """
    Selects a specified number of random points from an image, ensuring that each p
    meeting a defined intensity threshold within the image. The image is resized to
    are chosen randomly, with each potential point undergoing validation against cr

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

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

    Notes:
        The function converts the image to grayscale and resizes it to img_size x i
        points near the image boundary by applying a boundary offset calculated fro
        Each point must be centered in a window (defined by 'window_size') where al
        greater than or equal to 5. If the function fails to find a suitable 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
```

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

            # 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
            break  # Break outer loop if max attempts is reached without finding a

    return pts
```

In [10]:

```
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 imag
    and then converts it back to RGB. It uses OpenCV for the CLAHE operation and PI
    conversions.

    Parameters:
        imag (np.array): The input image array. Expected to be in format suitable f
        clip (float): The clipping limit for the CLAHE algorithm, which controls th
                      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 parame
          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 t

    Parameters:
    - landmark_errors: List of landmark errors including outliers.
    """
    landmark_errors_sorted = sorted(landmark_errors) # includes all outliers as wel
    # 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 <= t
        success_rate = successful_count / len(landmark_errors_sorted)
        success_rates.append(success_rate * 100) # convert to percentage
    # Plot the curve
    plt.plot(thresholds, success_rates, label="Success Rate Curve")
    plt.xlabel("Threshold")
    plt.ylabel("Success Rate (%)")
    plt.title("Success Rate vs. Threshold")
    plt.legend()
    plt.grid(True)
    plt.show()
    # Compute AUC
    auc = np.sum(success_rates) / 10000 # normalize to 0-1
    print("AUC:", auc)

def plot_landmark_errors(landmark_errors,rpth,chr='All'):
    """
    Plots a graph of landmark errors over successive iterations to provide a visual
    across samples. This function is designed to help in the assessment of registra
    or computer vision tasks by plotting each landmark error against its iteration
    displays the average landmark error across all iterations.

    Parameters:
        landmark_errors (list of float): A list containing numerical errors for eac
                                         Outliers (e.g., errors set to 10000) are a
        rpth (str): Path where the resulting plot image will be saved.
        chr (str, optional): Characteristic or description to include in the plot t
                             used. Defaults to 'All'.

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

    Notes:
        - This plot is useful for tracking improvements or deteriorations in landma
        - It automatically filters out error values set to 10000, considering them
        - The function saves the plot in the directory specified by `rpth` and name
    """
    #excluding outliers
    landmark_errors =[x for x in landmark_errors if x!=10000]
    # Generate sample numbers (or indices) based on the number of errors recorded
    samples = list(range(1, len(landmark_errors) + 1))
    # Calculate the average landmark error
    avg_error = sum(landmark_errors) / len(landmark_errors)
    # Plotting
    plt.figure(figsize=(12, 7))
```

```
    plt.plot(samples, landmark_errors, marker='o', linestyle='-', color='#2C3E50',
    plt.axhline(y=avg_error, color='#E74C3C', linestyle='--', label=f"Average Error
    plt.title("Landmark Error vs. Iteration for Database Housing model {} images".f
    plt.xlabel("Iteration Number", fontsize=14)
    plt.ylabel("Landmark Error", fontsize=14)
    plt.xticks(samples, [f"Iteration {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 keypoints_visualization(image, landmarks, img_size, rpth, num):
    """
    Visualize keypoints on a given image.

    Parameters:
    - image (ndarray): The input image on which the keypoints are to be visualized.
    - landmarks (list or array-like): Collection of keypoints to be drawn on the im
    - img_size (tuple): The size (width, height) of the image.
    - rpth (str): The path where the image with keypoints will be saved.
    - num (int or str): An identifier for the saved image file.

    Notes:
    - The function assumes the existence of a coordinates_rescaling function.
    - The displayed image has the title 'Chosen Keypoints' and does not have any ax
    """

    # Rescale the coordinates of the landmarks
    rescaled_landmarks = coordinates_rescaling(landmarks, img_size, img_size, 256)

    # Define color map based on the number of points
    num_points = len(rescaled_landmarks)
    if num_points > 15:
        cmap = plt.get_cmap('tab20')  # Changed from 'tab10' to 'tab20' for more co
    else:
        cmap = ListedColormap(["red", "yellow", "blue", "lime", "magenta", "indigo"
                               "darkgreen", "maroon", "black", "white", "chocolate"
    colors = np.array([cmap(i) for i in range(num_points)])

    # Create the figure and axis
    fig, ax1 = plt.subplots(1, 1, figsize=(6, 6))

    # Set title and turn off the axis
    ax1.set_title('Chosen Keypoints')
    ax1.axis('off')

    # Show the image
    ax1.imshow(image, cmap='gray')  # Assuming image is grayscale; remove cmap if i

    # Define the radii for the circles
    radius1, radius2 = 4, 1

    # Plot each rescaled landmark on the image
    for point, color in zip(rescaled_landmarks, colors):
        x, y = point
```

```
        # Create two circles at each point
        circ1 = plt.Circle((x, y), radius1, facecolor=color, edgecolor='white', alp
        circ2 = plt.Circle((x, y), radius2, facecolor=color, edgecolor='white')
        # Add the circles to the axis
        ax1.add_patch(circ1)
        ax1.add_patch(circ2)

    # Save the figure
    plt.savefig(os.path.join(rpth, 'Keypoints_Visualization_' + str(num) + '.png'))

    # Display the plot
    plt.show();

def image_point_correspondences(images,img_size,landmarks1,landmarks2,landmarks3,rp
    """
    Displays and compares point correspondences between three images using given la

    This function visualizes three images side-by-side with their respective landma
    of corresponding landmarks across the images is marked with the same color for
    of correspondences. The function is designed to handle visualization for studie
    registration or similar tasks where landmark matching is crucial.

    Parameters:
        images (list of str): File paths to the three images (source, target, and r
        img_size (int): The size to which images should be resized, specified as wi
        landmarks1 (list of tuples): Landmark points on the first image (source ima
        landmarks2 (list of tuples): Corresponding landmark points on the second im
        landmarks3 (list of tuples): Corresponding landmark points on the third ima
        rpth (str): Path where the resultant visualization should be saved.
        num (int): An identifier number used to differentiate the output file name.

    Returns:
        None: The function directly displays the image using matplotlib and saves t

    Notes:
        - The function uses OpenCV for reading and resizing images, and the 'clahe'
        - Matplotlib is used for visualizing the images and landmarks. The colormap
          of landmarks; if there are more than 15 landmarks, a cyclic colormap is u
        - This function is particularly useful for visualizing transformations and
          similar fields where point correspondence is critical.
        - It is assumed that the order of landmarks in `landmarks1`, `landmarks2`,
    """
    image1 = cv2.resize(cv2.imread(images[0]),(256,256))
    image2 = cv2.resize(cv2.imread(images[1]),(256,256))
    #### enhance image contrast for visulaiztion
    image1 = clahe(image1, 3)
    image2 = clahe(image2, 3)
    keypoints_visualization(image1,landmarks1,img_size,rpth,num)
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,256)
    landmarks3 = coordinates_rescaling(landmarks3,img_size,img_size,256)
    assert len(landmarks2) == len(landmarks3), f"points lengths are incompatible: {
    num_points = len(landmarks2)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
    ax1.set_title('Source Image')
    ax2.set_title('Target 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"
                               "maroon", "black", "white", "chocolate", "gray", "bl
    colors = np.array([cmap(x) for x in range(num_points)])
    radius1, radius2 = 4, 1
    for point1, point2, color in zip(landmarks2, landmarks3, colors):
        x1, y1 = point1
        circ1_1 = plt.Circle((x1, y1), radius1, facecolor=color, edgecolor='white',
        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',
        circ2_2 = plt.Circle((x2, y2), radius2, facecolor=color, edgecolor='white')
        ax2.add_patch(circ2_1)
        ax2.add_patch(circ2_2)
    plt.savefig(os.path.join(rpth,'RetinaRegNet_Keypoints_Estimation_Results'+str(n
    plt.show();

def original_image_point_correspondences(images, img_size, landmarks1, landmarks2,
    """
    Visualize point correspondences across three images typically representing fixe
    and deformed states in image processing tasks. This function plots and saves th
    showing landmark points overlaid on each image. The images are enhanced using C
    and the landmarks are scaled according to a given image size.

    Parameters:
        images (list of np.array): A list containing three images (as numpy arrays)
                                   and deformed image states.
        img_size (tuple): The original size (width, height) of the images before re
        landmarks1 (list of tuples): Landmark coordinates for the first image.
        landmarks2 (list of tuples): Landmark coordinates for the second image.
        landmarks3 (list of tuples): Landmark coordinates for the third 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.

    Raises:
        AssertionError: If the lengths of landmarks1, landmarks2, and landmarks3 do

    Notes:
        - This function uses CLAHE for contrast enhancement of the images.
        - The landmarks are resized to fit a 256x256 image scale for visualization.
        - A colormap is used to differentiate points; if the number of points excee
          otherwise a specific 15-color map is applied.
    """

    assert len(landmarks1) == len(landmarks2) == len(landmarks3), "All landmarks li
    num_points = len(landmarks1)
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

    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],(256,256)),1.2),cmap='gray')
    ax2.imshow(clahe(cv2.resize(images[1],(256,256)),1.2),cmap='gray')
    ax3.imshow(clahe(cv2.resize(images[2],(256,256)),1.2),cmap='gray')

    landmarks1 = coordinates_rescaling(landmarks1,img_size,img_size,256)
    landmarks2 = coordinates_rescaling(landmarks2,img_size,img_size,256)
    landmarks3 = coordinates_rescaling(landmarks3,img_size,img_size,256)

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

    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, co
        # Landmarks for Image 1
        x1, y1 = point1
        circ1_1 = plt.Circle((x1, y1), radius1, facecolor=color, edgecolor='white',
        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',
        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',
        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, 'RetinaRegNet_Original_Keypoints_Estimation_Resu
    plt.show();
```

In [11]:

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

    Args:
    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 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
    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 transfo

    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):
    """
    Deform a set of points using given quadratic coefficients.

    :param points: A list of points, where each point is a tuple (x, y).
    :param coefficients: The coefficients to use for the deformation.
    :return: A list of deformed points.
    """
    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
    """
    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
    - 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, pol
    transformed_points = coordinates_rescaling_high_scale(transformed_points,image_
```

```
    errors = np.linalg.norm(np.array(fixed_points) - transformed_points, axis=1)
    mle = np.mean(errors)
    return mle


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

    Args:
    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

    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,
        # 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**

        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

    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(landma
    return homography_matrix

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

    Parameters:
        landmarks1 (list of tuples): List of original landmark points in the source
        landmarks2 (list of tuples): List of corresponding landmark points in the t
                                     The points in landmarks2 should correspond one
        img_size (int): Original size of the images from which the landmarks were e
                        rescale points for accurate computation of the transformati
        new_img_size (int): New size to which the points will be rescaled before co
                            This should reflect the size of the image space into wh

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

    Notes:
        - Ensure that the number of points in landmarks1 and landmarks2 are equal a
        - This function involves rescaling coordinates, calculating a transformatio
          tasks where geometric transformations are necessary for alignment and reg
    """
    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(landmark
```

```
    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

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

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

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

    Notes:
        - Ensure that the number of points in landmarks1 and landmarks2 are equal a
        - This function is essential in image processing tasks where precise transf
    """
    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):
    """
    Deform the image using given third-order polynomial coefficients.

    :param image: The image to deform, as a numpy array (height, width) or (height,
    :param coefficients: The coefficients to use for the deformation.
    :return: The deformed image.
    """
    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 consiste
    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 +
               a5*x_coords**2 + a6*x_coords*y_coords + a7*y_coords**2 + a8*x_coords
    y_prime = (a11*x_coords**3 + a12*x_coords**2*y_coords + a13*x_coords*y_coords**
               a15*x_coords**2 + a16*x_coords*y_coords + a17*y_coords**2 + a18*x_co

    # 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

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



def warp_image_quadratic_matrix(image, coefficients):
    """
    Deform the image using given quadratic coefficients.

    :param image: The image to deform, as a numpy array (height, width) or (height,
    :param coefficients: The coefficients to use for the deformation.
    :return: The deformed image.
    """
    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 consiste
    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 + a
    y_prime = a7*x_coords + a8*y_coords + a9*x_coords*y_coords + a10*x_coords**2 +
```

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

    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, la
    """
    Computes the third-order polynomial transformation matrix between two sets of l
    and applies this transformation to warp one image to align with another. The fu
    displays and saves the fixed, moving, and deformed images with contrast enhance

    Parameters:
        images (list of str): List containing the file paths of two images.
        img_size (int): The size (height and width) to which the images will be res
        landmarks1 (list of tuples): Landmark points (x, y) on the first image.
        landmarks2 (list of tuples): Corresponding landmark points (x, y) on the se
        rpth (str): The directory path where the resultant images will be saved.
        num (int): Identifier number used to differentiate the output file names.
        cll (float, optional): The clipping limit for the CLAHE algorithm used in c

    Raises:
        ValueError: If the landmarks1 list is empty.

    Returns:
        tuple: A tuple containing three elements:
            - imags (list of np.array): List containing the original fixed and movi
            - imgs (list of str): File paths where the output images have been save
            - coefficients (np.array): Coefficients of the third-order polynomial t

    Notes:
        - The third-order polynomial transformation is a complex and computationall
          for fine-grained image registration tasks where simple affine transformat
        - The resultant 'Deformed Image' is the second image warped to align with t
          polynomial transformation matrix.
    """
    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,

    # Apply the transformation using third-order polynomial
```

```
    transformed_image = warp_image_third_order_polynomial(img2, coefficients_for_tr
    imags.append(transformed_image)

    # Display and save the images
    plt.figure(figsize=(10, 5))

    plt.subplot(131)
    plt.imshow(clahe(img1.astype(np.uint8),cll))
    plt.title('Fixed Image')
    plt.axis('off')

    plt.subplot(132)
    plt.imshow(clahe(img2.astype(np.uint8),cll))
    plt.title('Moving Image')
    plt.axis('off')

    plt.subplot(133)
    plt.imshow(clahe(transformed_image.astype(np.uint8),cll))
    plt.title('Deformed Image')
    plt.axis('off')

    plt.show()

    imgs.append(os.path.join(rpth, 'Deformed_Image_' + str(num) + '_.png'))
    imgs.append(os.path.join(rpth, 'Target_' + str(num) + '_.png'))
    cv2.imwrite(os.path.join(rpth, 'Target_' + str(num) + '_.png'), img1)
    cv2.imwrite(os.path.join(rpth, 'Source_' + str(num) + '_.png'), img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transformed_i
    return imags,imgs,coefficients




def compute_affine_matrix(images,img_size,landmarks1, landmarks2,rpth,num,cll=5):
    """
    Computes an affine transformation matrix based on provided landmarks from two i
    this transformation to visually compare the source, target, and transformed ima

    This function takes pairs of landmarks from two images and computes the affine
    that best maps the source image to align with the target image. It then applies
    the source image and displays both the original (source and target) and the tra
    The images are displayed after contrast enhancement and are saved to the specif

    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 res
        landmarks1 (list of tuples): Landmark points (x, y) on the source image.
        landmarks2 (list of tuples): Corresponding landmark points (x, y) on the ta
        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 contr

    Returns:
        tuple: Contains two items:
            - imgs (list of str): File paths where the images are saved.
            - affine_matrix (numpy.ndarray): The computed 3x3 affine transformation
```

```
    Raises:
        ValueError: If the landmarks list is empty, indicating insufficient data to

    Notes:
        - This function uses OpenCV for image processing tasks including reading, r
        - The affine transformation matrix is computed using a least squares method
        - This function is useful for image registration tasks where visual compari
    """
    imgs=[]
    img1 = cv2.resize(cv2.imread(images[0]),(img_size,img_size))
    img2 = cv2.resize(cv2.imread(images[1]),(img_size,img_size))

     # 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.

    # Display the original and transformed images
    plt.figure(figsize=(10, 5))


    plt.subplot(131)
    plt.imshow(clahe(img1,cll))

    plt.title('Fixed Image')
    plt.axis('off')

    plt.subplot(132)
    plt.imshow(clahe(img2,cll))
```

```
    plt.title('Moving Image')
    plt.axis('off')

    plt.subplot(133)
    plt.imshow(clahe(transformed_image,cll))

    plt.title('Deformed Image')
    plt.axis('off')

    plt.show();
    imgs.append(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'))
    imgs.append(os.path.join(rpth,'Target_'+str(num)+'_.png'))
    cv2.imwrite(os.path.join(rpth,'Target_'+str(num)+'_.png'),img1);
    cv2.imwrite(os.path.join(rpth,'Source_'+str(num)+'_.png'),img2);
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transformed_i

    return imgs,affine_matrix

def warp_quadratic_matrix(images,img_size,landmarks1, landmarks2,rpth,num,cll=5):
    """
    Computes a quadratic transformation matrix from source to target landmarks and
    to the source image. The transformed source image is displayed alongside the or
    and all images are saved to disk.

    This function takes pairs of corresponding landmarks from the source and target
    transformation matrix. This matrix is then used to warp the source image to mat
    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 res
        landmarks1 (list of tuples): Landmark points (x, y) on the source image.
        landmarks2 (list of tuples): Corresponding landmark points (x, y) on the ta
        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 contr

    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 ori
            - quadratic_matrix (numpy.ndarray): The computed quadratic transformati

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

    Notes:
        - The function uses OpenCV for image processing tasks such as reading, resi
        - The quadratic transformation matrix is computed using a least squares met
        - Matplotlib is used for visualizing the before and after effects of the tr
        - This function is particularly useful in applications such as image regist
    """
    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

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

    quadratic_matrix_for_image_deformed = compute_quadratic_matrix(landmarks2, land

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

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

    # Display the original and transformed images
    plt.figure(figsize=(10, 5))


    plt.subplot(131)
    plt.imshow(clahe(img1,cll))

    plt.title('Fixed Image')
    plt.axis('off')

    plt.subplot(132)
    plt.imshow(clahe(img2,cll))

    plt.title('Moving Image')
    plt.axis('off')

    plt.subplot(133)
    plt.imshow(clahe(transformed_image.astype(int),cll))

    plt.title('Deformed Image')
    plt.axis('off')

    plt.show();
    imags.append(transformed_image)
    imgs.append(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'))
    imgs.append(os.path.join(rpth,'Target_'+str(num)+'_.png'))
    cv2.imwrite(os.path.join(rpth,'Target_'+str(num)+'_.png'),img1);
    cv2.imwrite(os.path.join(rpth,'Source_'+str(num)+'_.png'),img2);
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transformed_i
    return imgs,imags,quadratic_matrix

def compute_and_apply_homography(images, img_size, landmarks1, landmarks2, rpth, nu
    """
    Computes the homography transformation matrix based on landmark correspondences
    and applies this transformation to the source image. The function displays the
    target images along with the transformed source image. It also saves these imag
```

```
    Parameters:
        images (list of str): Paths to the source and target images.
        img_size (int): The size to which both images will be resized (assumed squa
        landmarks1 (list of tuples): Landmark points (x, y) from the source image.
        landmarks2 (list of tuples): Corresponding landmark points (x, y) from the
        rpth (str): The directory path where the resultant images will be saved.
        num (int): An identifier number used to differentiate the output file names
        cll (float): The clipping limit for the CLAHE algorithm used in contrast en

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

    Raises:
        ValueError: If the list of landmarks is empty, indicating that there are no
    Notes:
        - The function uses OpenCV for image reading, resizing, and applying the ho
          Matplotlib is used for displaying the images.
        - Ensure the landmarks are accurately defined as their correspondence direc
        - Homography transformations are particularly useful for applications in im
    """
    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],

    # Display the original and transformed images
    plt.figure(figsize=(10, 5))

    plt.subplot(131)
    plt.imshow(clahe(img1,cll))
    plt.title('Fixed Image')
    plt.axis('off')

    plt.subplot(132)
    plt.imshow(clahe(img2,cll))
    plt.title('Moving Image')
    plt.axis('off')
```

```
    plt.subplot(133)
    plt.imshow(clahe(transformed_image.astype(np.uint8),cll))
    plt.title('Deformed Image')
    plt.axis('off')

    plt.show();

    imags.append(transformed_image)
    imgs.append(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'))
    imgs.append(os.path.join(rpth,'Target_'+str(num)+'_.png'))
    cv2.imwrite(os.path.join(rpth, 'Target_' + str(num) + '_.png'),img1)
    cv2.imwrite(os.path.join(rpth, 'Source_' + str(num) + '_.png'),img2)
    cv2.imwrite(os.path.join(rpth,'Deformed_Image_'+str(num)+'_.png'),transformed_i
    return imgs,imags,homography_matrix
```

In [12]:

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

    Args:
    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):
    """
    Estimate affine transformation matrix using point correspondences.

    Args:
    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):
    """
    Estimate 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 (se

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

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

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

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

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

    return updated_set1, updated_set2
```

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

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

    Returns:
        tuple of lists: Returns two lists (updated_set1, updated_set2) containing t
                        `set1` and `set2` respectively.
    Notes:
        - Ensure that `set1` and `set2` are of equal length and that the points cor
          as any misalignment could result in incorrect calculations and poor resul
        - This function is typically used in image processing and computer vision t
          alignment and transformation of point sets between images are required, e
          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
        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):
    """
    Filter outliers based on a specified condition.

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

    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 c
    - criteria (str, optional): The criteria to use for filtering outliers. Options
    - thresh (int, optional): Threshold value used in the outlier removal process.

    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 homograph
    - If 'affine' is chosen, it removes outliers based on affine transformation err
    """
    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
    return computed,original
```

In [13]:

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

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

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


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

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

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

```
    return scaled_points

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

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

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

    Notes:
        - This function uses OpenCV's `createCLAHE` method to apply the CLAHE algor
          first converted to grayscale as CLAHE is typically applied to single-chan
          visualization of detail.
        - The images are processed in-place and saved in the same directory as the
          prefixed to their original filenames.
        - It is recommended to adjust the `clip` parameter based on the specific re
          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 speci

    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='bilinea
    return uniform_feature_maps


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

    This function processes images to generate feature maps at multiple resolutions

    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_dis
            pts = pts+select_random_points(images[0],N,multi_img_size*(i+1),offset,
            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(ima
            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,im
    return src_ft,trg_ft

def landmarks_condition_check(orig_images, img_size, t, uft, landmarks1, landmarks2
    """
    Iteratively attempts to improve image registration quality by enhancing image c
    until certain quality conditions are met or a maximum number of attempts is rea
    for image contrast enhancement and uses various feature transformation and scal
```

```
    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
        landmarks1 (list of tuples): Initial landmarks as (x, y) coordinates in the
        landmarks2 (list of tuples): Target landmarks as (x, y) coordinates in the
        max_tries (int, optional): Maximum number of attempts to improve image regi
        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.
        offset (float, optional): Offset used in point selection to avoid edge effe
        window_size (int, optional): Size of the window used in point selection. De
        iccl (float, optional): Inverse consistency criteria limit used in landmark
        outlier_cond (str, optional): Condition used to determine outliers. Default
        thresh (float, optional): Threshold used for filtering outliers. Defaults t

    Returns:
        tuple: Depending on the success of the registration process, this function
            - The original images and the best set of landmarks found, or
            - The original images and a set of default landmarks if conditions are

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

    Notes:
        - This function is particularly useful in medical imaging or computer visio
          registration is crucial for further analysis.
        - The effectiveness of the registration process depends heavily on the qual
        - CLAHE and other image processing techniques may not always produce the de
          are of poor quality or the initial landmarks are inaccurately defined.
    """
    imgs,lim,land_marks1, land_marks2,list_landmarks_2, list_sim_scores,list_landma
    tries, ch, = 0, 0
    assert len(landmarks1) == len(landmarks2), f"Points lengths are incompatible: {
    landmarks2,landmarks1 = filter_outlier_cond(landmarks2,landmarks1,outlier_cond,
    print(len(landmarks2))
    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_si
            dft = DFT(images, img_size, pts)
            src_ft,trg_ft = dft.feature_upsampling(RetinaRegNet_Intialization(image
            land_marks1,sim_score, land_marks2 = dft.feature_maps(src_ft,trg_ft,icc
            del src_ft
            del trg_ft
            torch.cuda.empty_cache()
            gc.collect()
            land_marks2,land_marks1 = filter_outlier_cond(land_marks2,land_marks1,o
```

```
            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 [14]:

```
def folder_structure(path):
    """
    Create 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 value
    """
    result_lists = {f'Subject_{i + 1}': [] for i in range(len(nfn))}
    for i in fls:
        if '_'.join(map(str,i.split('.')[0].split('_')[-2:])) in nfn:
            result_lists[str('_'.join(map(str,i.split('.')[0].split('_')[-2:])))] .
        else:
            continue
    return result_lists

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

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

    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
```

```python
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

    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 += [os.path.join(pth,str(i),fnn[2],result_lists[i][j])] * N
                a.append(os.path.join(pth,str(i),fnn[2],result_lists[i][j]))
                #a.append(result_lists[i][j])
            else:
                c.append(os.path.join(pth,str(i),fnn[0],result_lists[i][j]))
                #c.append(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
```

```python
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)
    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 imag
    - img_shape (int, optional): Desired image dimension for rescaling. Default is

    Returns:
    - tuple: Scaled coordinates for the first and second images.
    """
    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
    return (H2,W2),max(max(H1,W1),max(H2,W2)),scaled_fixed_points,scaled_moving_poi



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 image paths.
    - fixed_points (list): List of fixed points.
    - moving_points (list): List of moving points.
    - img_shape (int): Desired image dimension for rescaling.

    Returns:
    - tuple: Scaled fixed and moving points.
    """
    fixed_image_size,max_image_size,fixed_pointss,moving_pointss,scaled_moving_poin
    for i in range(len(images)):
        fhs,mhs,fpnts,mpnts,scmpnts = coordinates_processing(images[i][0],images[i]
        fixed_image_size.append(fhs)
        max_image_size.append(mhs)
        fixed_pointss.append(fpnts)
        moving_pointss.append(mpnts)
        scaled_moving_points.append(scmpnts)
    return fixed_image_size,max_image_size,fixed_points,fixed_pointss,moving_points

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

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

    Returns:
    - tuple: Images, fixed points, and moving points extracted from the dataset.
    """
    fls,nfn,fnn=[],[],[]
    pth = os.path.join(os.getcwd(),path,'data')
    for i in 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 [15]:

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

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

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

    Notes:
    The function uses the SDFeaturizer from the 'stabilityai/stable-diffusion-2-1'
    from each image. After processing all images, the extracted features are concat
    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 [16]:

```
def main(images,rpth,ifn,img_size=256,up_ft_indices = 1,timestep = 75,N=50,offset=0
     """
    Perform image registration and point correspondence using a series of processin

    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.
    - img_size (int, optional): Size of the input images (default is 256).
    - up_ft_indices (int, optional): Up-sampling factor for feature indices (defaul
    - 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
    - window_size (int, optional): Size of the window for feature extraction (defau
    - 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 'affi
    - thresh (int, optional): Threshold value for outlier removal (default is 20).
    - max_tries (int, optional): Maximum number of attempts for matching features (
    - num (int, optional): Number of iterations for matching features (default is 5
    - clip (float, optional): Clip parameter for image enhancement (default is 1.0)
    - multi_ch (bool, optional): Flag indicating whether to use multi-channel proce
    - multi_iter (int, optional): Number of iterations for multi-channel processing
    - multi_img_size (int, optional): Size of images for multi-channel processing (

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

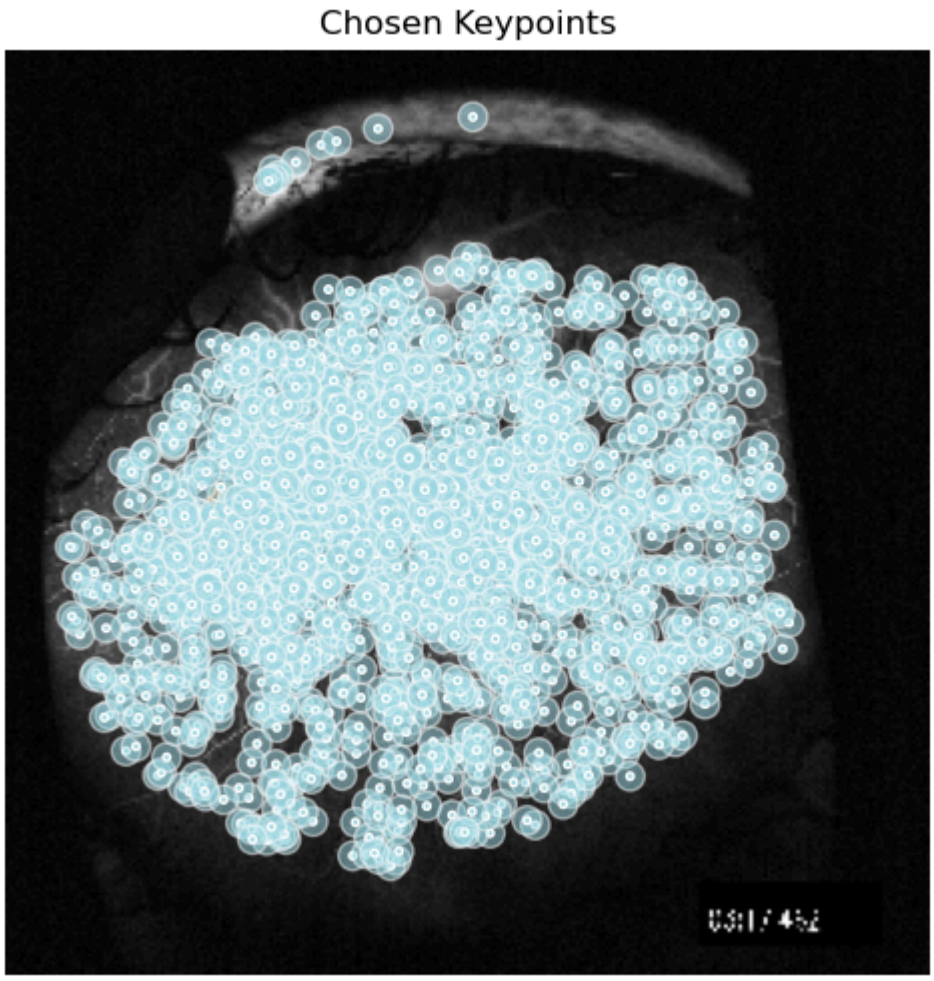
```
    - computed (list): List of computed image points after registration.

    Note:
    - This function performs various processing steps including feature extraction,
      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
    """
    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_
    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
    if len(computed)!=0:
        image_point_correspondences(images,img_size,pts,original,computed,rpth,ifn)
        return original,computed
    else:
        print("Image Registration is Unsuccessful for the presented Images due to u
        return [],[]
    torch.cuda.empty_cache()
```

In [17]:
```
img_size= 1024
images,fixed_points,moving_points = data_preprocessing('FLoRI21_DataPort')
fixed_image_size,max_image_size,fixed_points,scaled_fixed_points,scaled_moving_poin
```

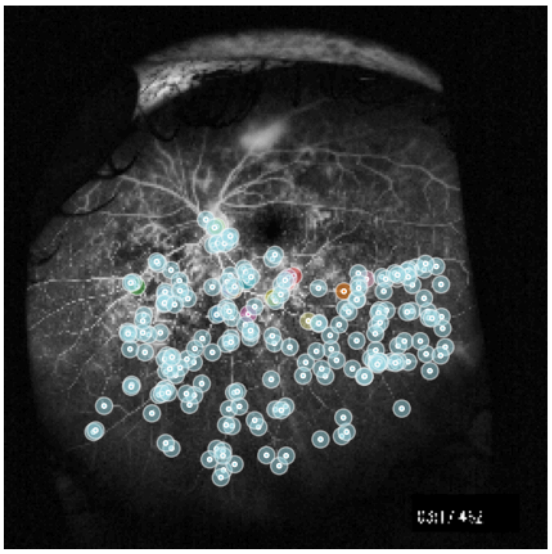
In [18]:
```
landmark_errors=[]
for i in range(len(images)):
    print("Iteration {}".format(i+1))
    print("Loading Source Images {0} ,Target Image{1} to the framework".format(imag
    original_low_res,computed_low_res = main(images[i],os.path.join(os.getcwd(),'FL
    imags,imgs,homography_matrix_low_res = compute_and_apply_homography(images[i][:
    if len(homography_matrix_low_res) !=0:
        transformed_points_hom = transform_points_homography(scaled_moving_points[i
        transformed_points_high_res_hom =  coordinates_rescaling(transformed_points
        original_low_res,computed_low_res = main(imags,os.path.join(os.getcwd(),'FL
        imag,imgs,quadratic_matrix_low_res = compute_third_order_polynomial_matrix_
        if len(quadratic_matrix_low_res) !=0:
            ## rescaled version for dispaly purposes
            transformed_points_poly = transform_points_third_order_polynomial(trans
            original_image_point_correspondences(imag,img_size, scaled_fixed_points
            ### Original Version for computation of errors
            polynomial_matrix = transform_points_third_order_polynomial_matrix(orig
            error = compute_landmark_error_fixed_space(polynomial_matrix,fixed_poin
            print("Recorded Landmark Error for Iteration {0} is {1}".format(i+1,err
            landmark_errors.append(error)
        else:
            landmark_errors.append(10000)
```

```
    else:
        landmark_errors.append(10000)
```

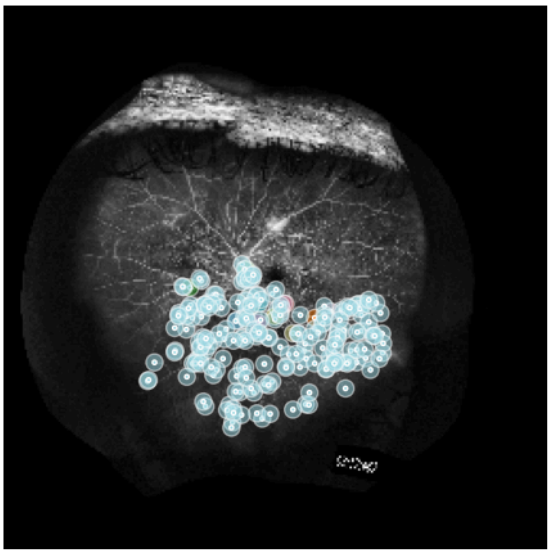
```
Iteration 1
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif ,Target Image/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/FA/Raw_FA_1_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
/blue/weishao/vi.sivaraman/conda/envs/VBS_HRC/lib/python3.11/site-packages/torch/nn/modules/conv.py:459: UserWarning: Applied workaround for CuDNN issue, install nvrtc.so (Triggered internally at ../aten/src/ATen/native/cudnn/Conv_v8.cpp:80.)
  return F.conv2d(input, weight, bias, self.stride,
/scratch/local/22569018/ipykernel_1623912/2877224606.py:74: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  points_indices = torch.tensor(pts_list)
208
```



Source Image

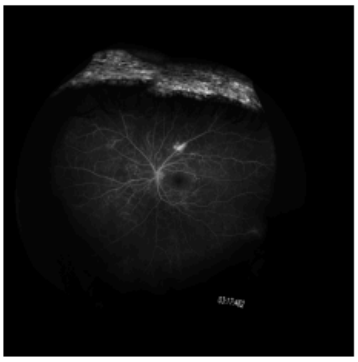


Target Image



```
Homography Matrix:
[[ 5.61061207e-01 -1.41432074e-01  2.50466914e+02]
 [ 7.48336292e-02  4.40905276e-01  2.41829394e+02]
 [-1.93660443e-06 -1.73865167e-04  1.00000000e+00]]
```

Fixed Image

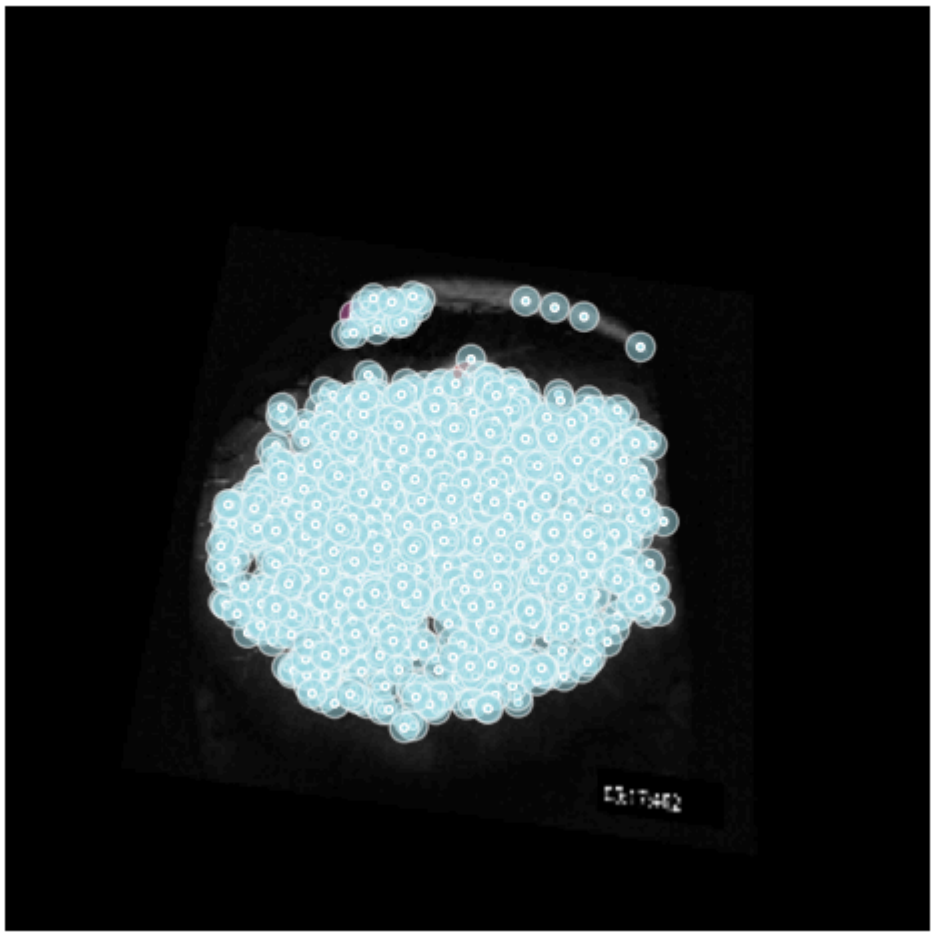


Moving Image

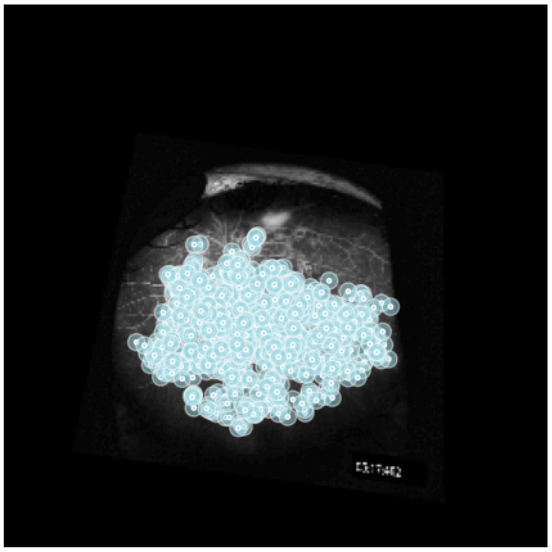
Deformed Image

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
710
```

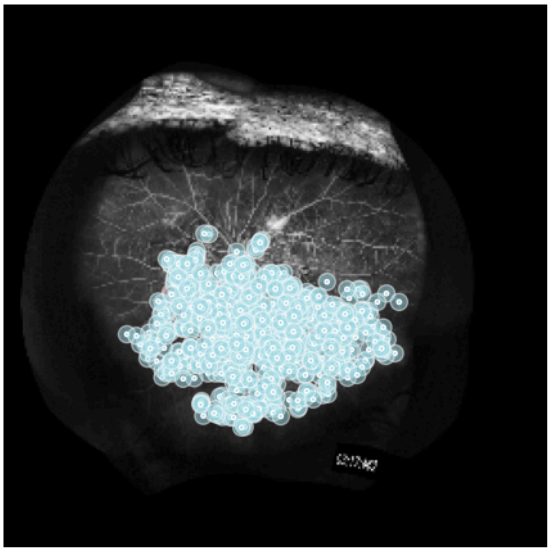
Chosen Keypoints

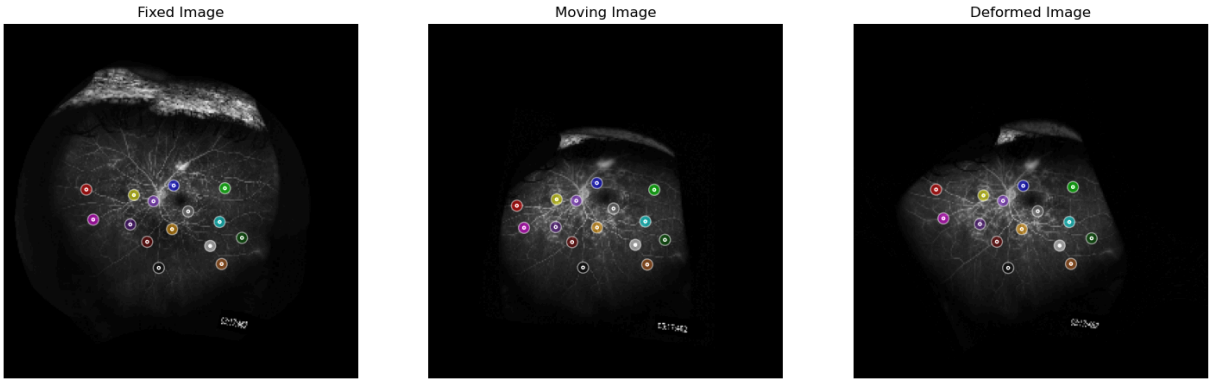


Source Image



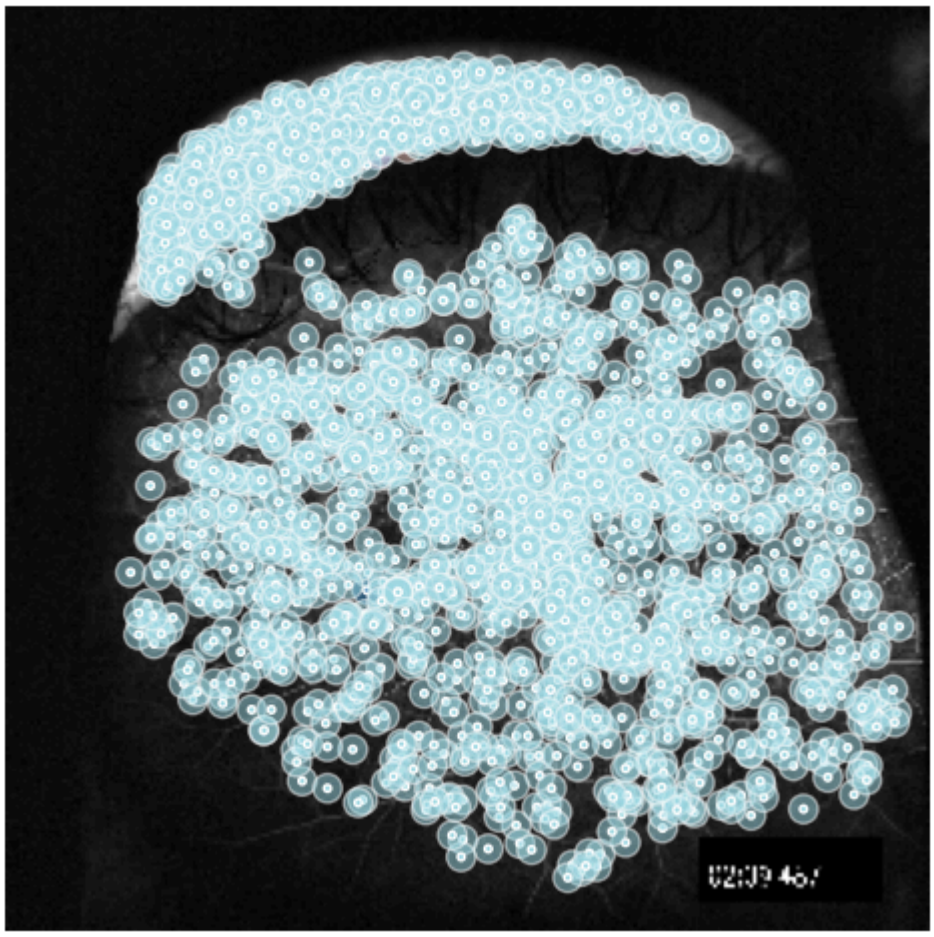
Target Image



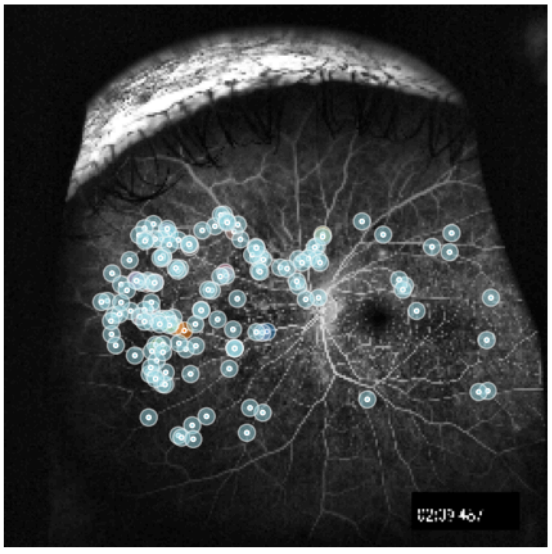


Recorded Landmark Error for Iteration 1 is 8.749038446032909
Iteration 2
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif ,Target Image/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21_DataPort/data/Subject_2/FA/Raw_FA_2_Subject_2.tif to the framework
Loading pipeline components...: 0%| | 0/6 [00:00<?, ?it/s]
153

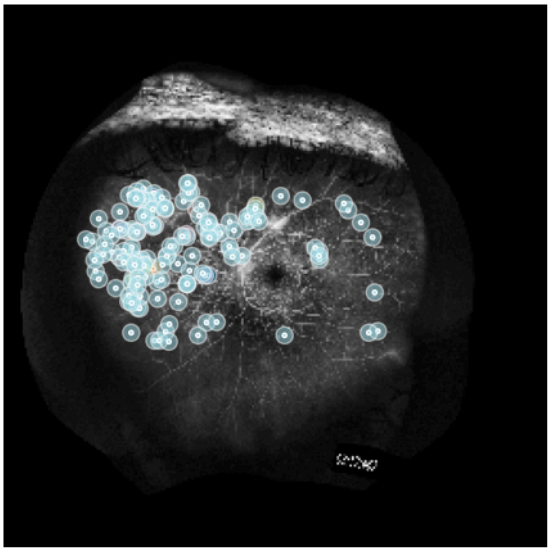
Chosen Keypoints



Source Image

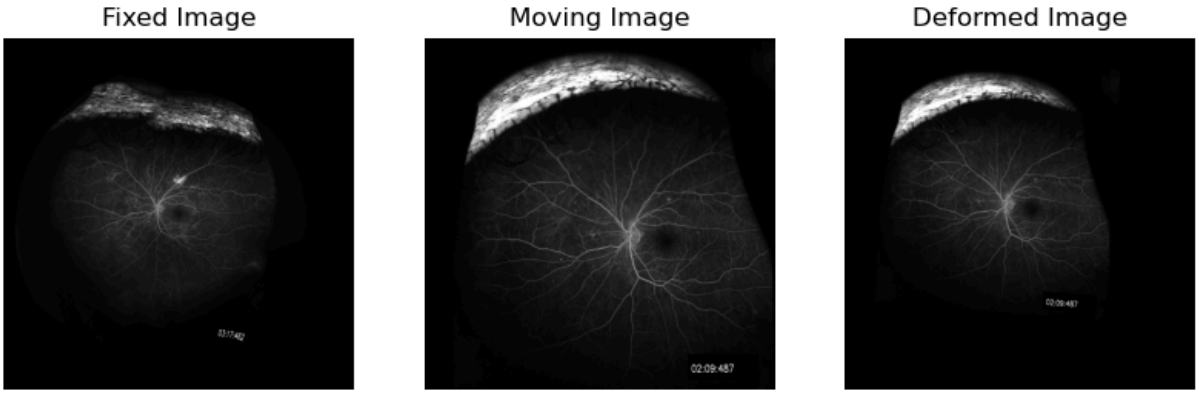


Target Image

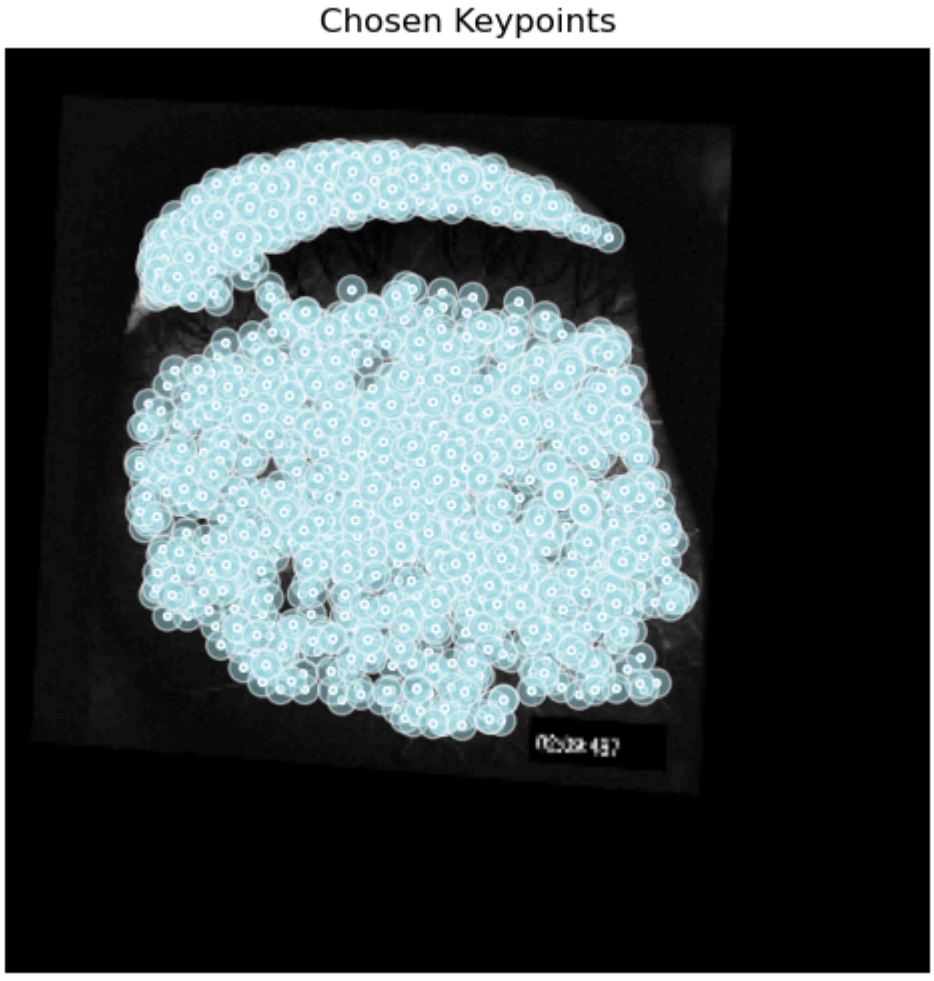


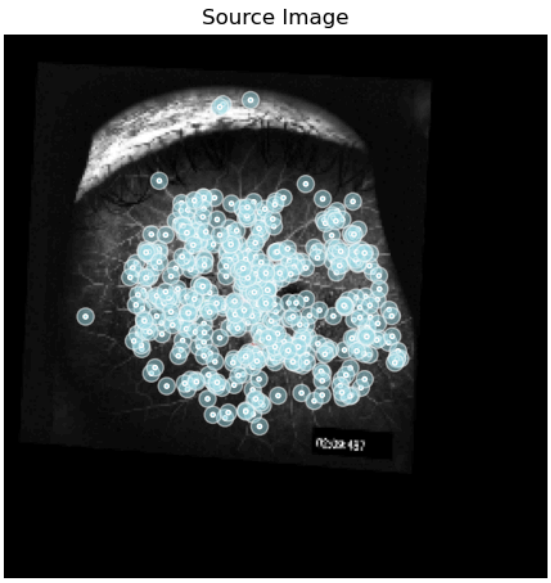
Homography Matrix:

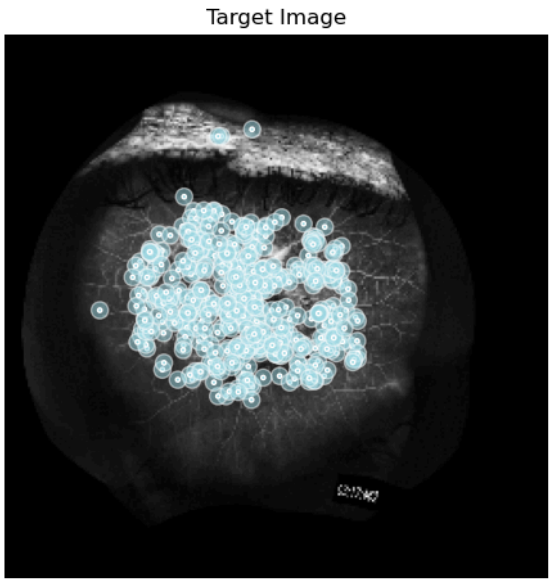
```
[[ 6.97182568e-01 -3.30054245e-02  6.34819242e+01]
 [ 2.98053207e-02  7.00984237e-01  5.23725037e+01]
 [-3.50091534e-05  3.82990449e-06  1.00000000e+00]]
```

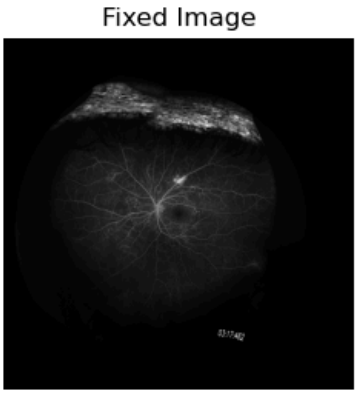


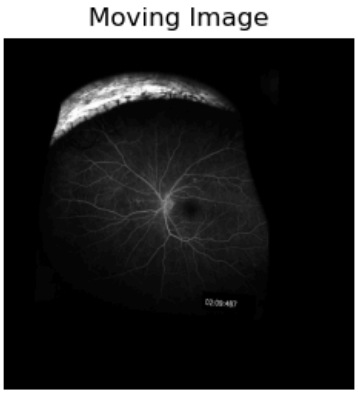

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
485
```

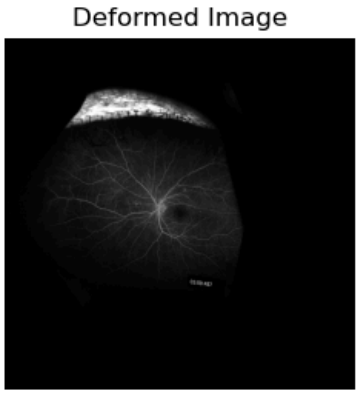


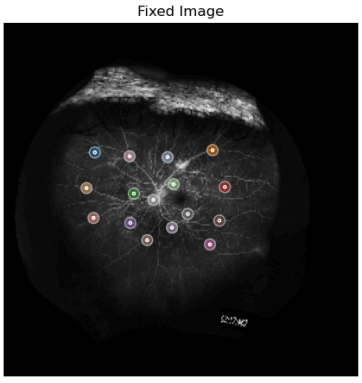


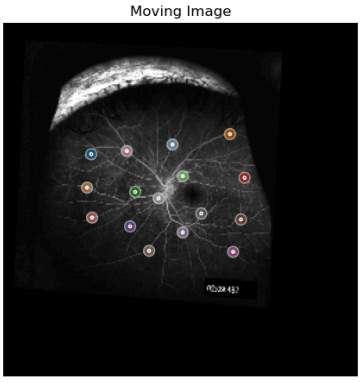


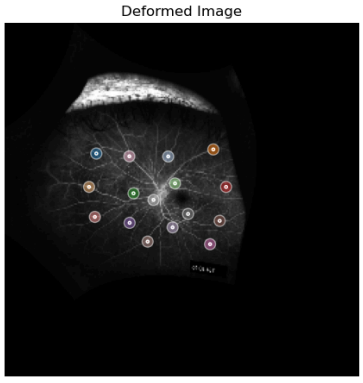






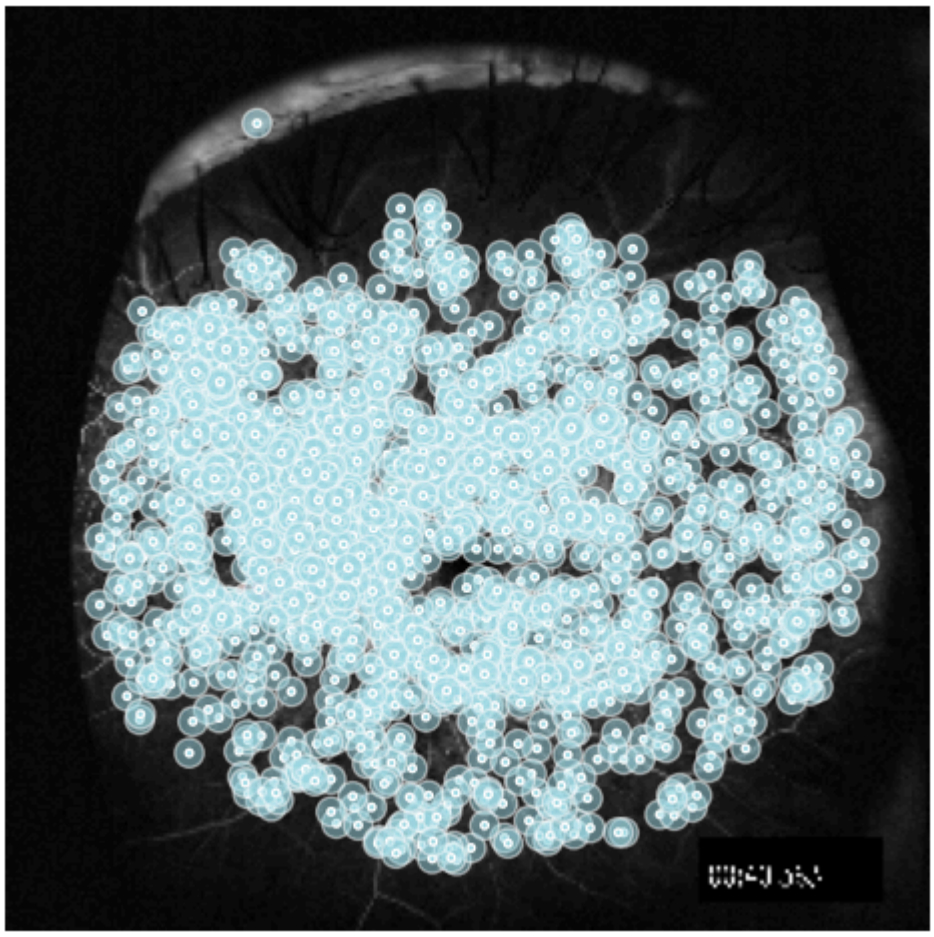






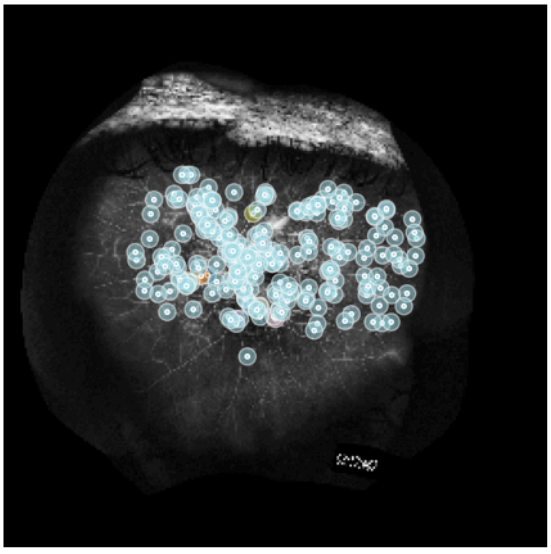

```
Recorded Landmark Error for Iteration 2 is 10.66290389511247
Iteration 3
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_2/FA/Raw_FA_3_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
239
```

Chosen Keypoints



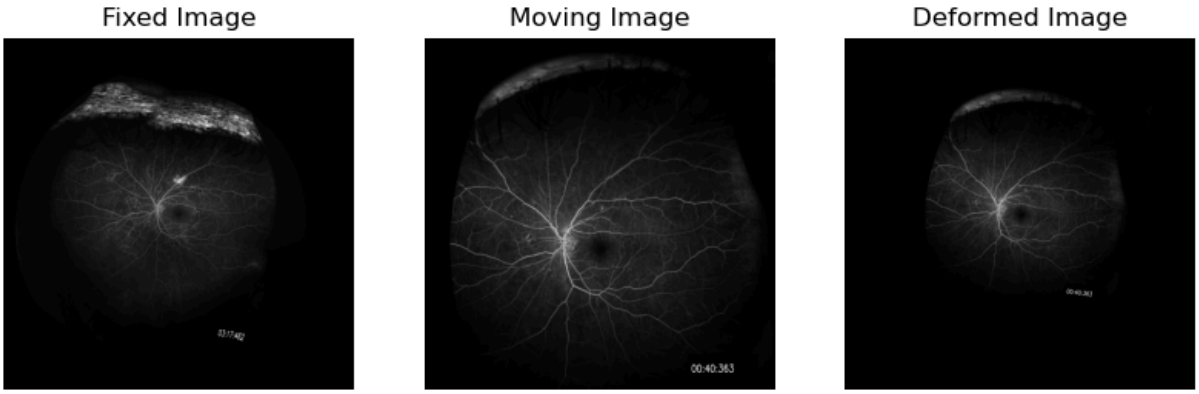
Source Image

Target Image

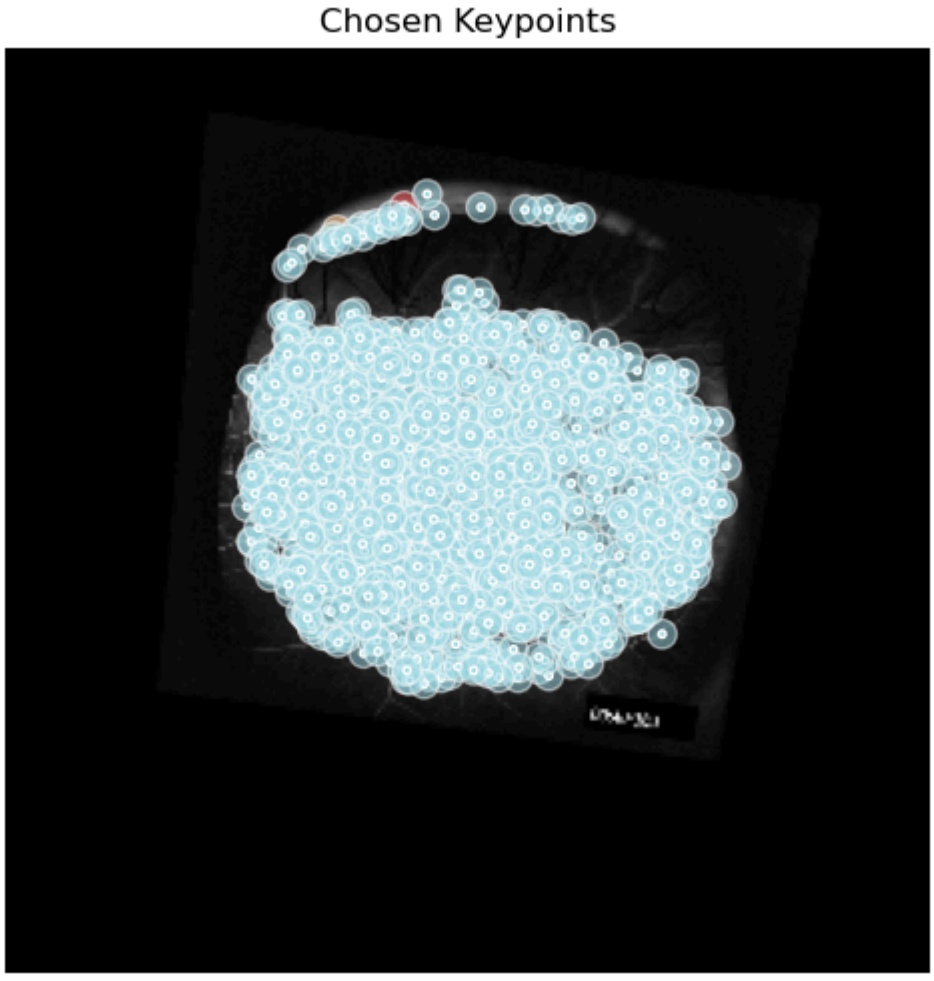


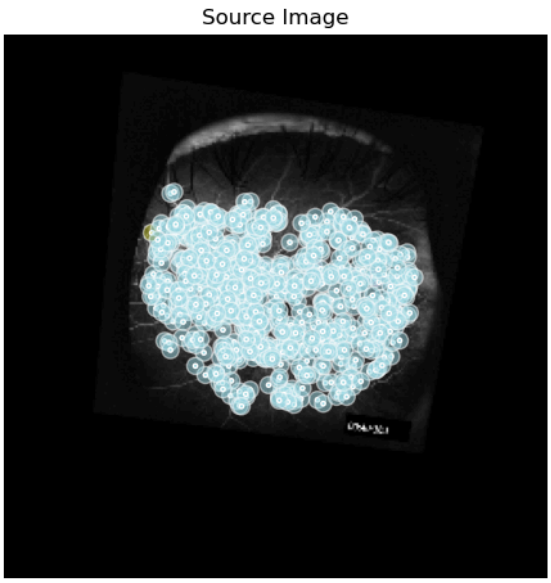
Homography Matrix:

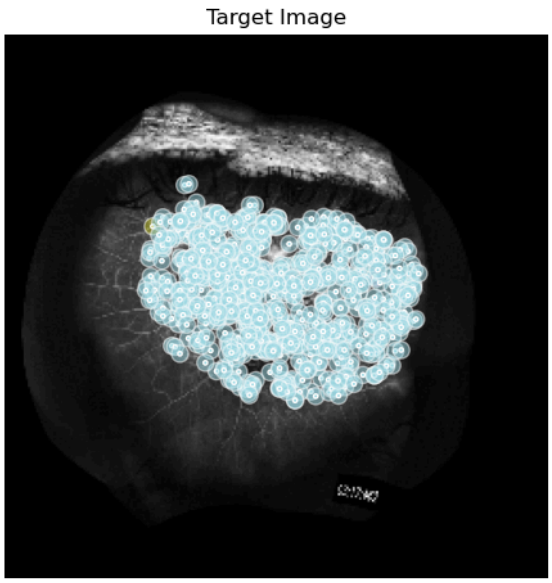
```
[[ 6.90923060e-01 -3.71556824e-02  2.23843744e+02]
 [ 1.06891836e-01  6.97986371e-01  6.98996409e+01]
 [ 2.92173489e-05  1.00562885e-04  1.00000000e+00]]
```

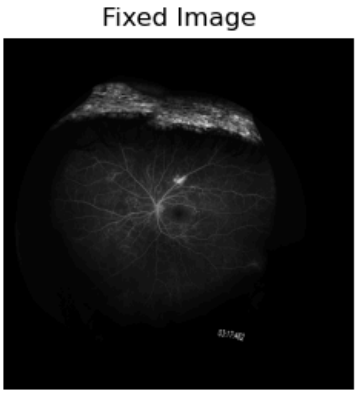


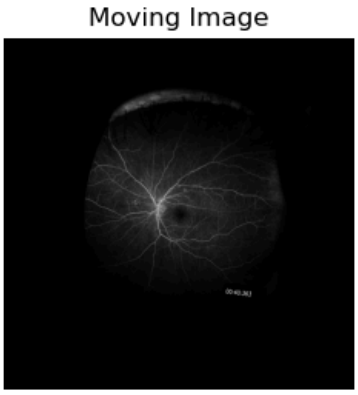

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
759
```

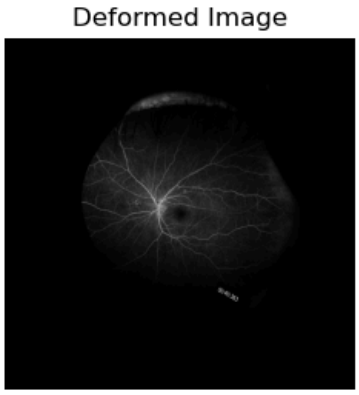


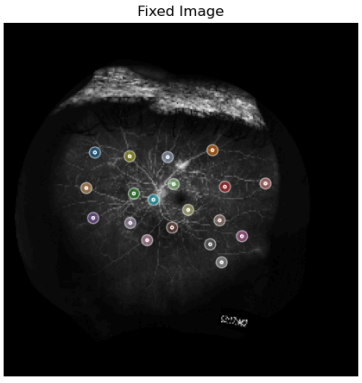


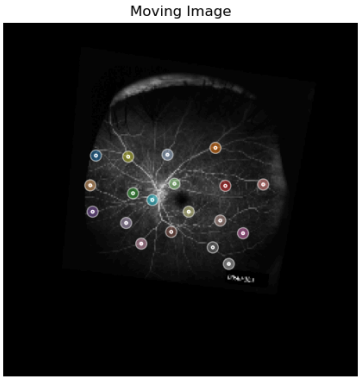


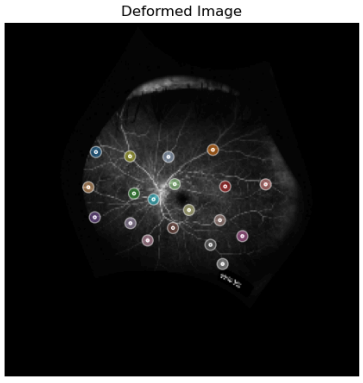






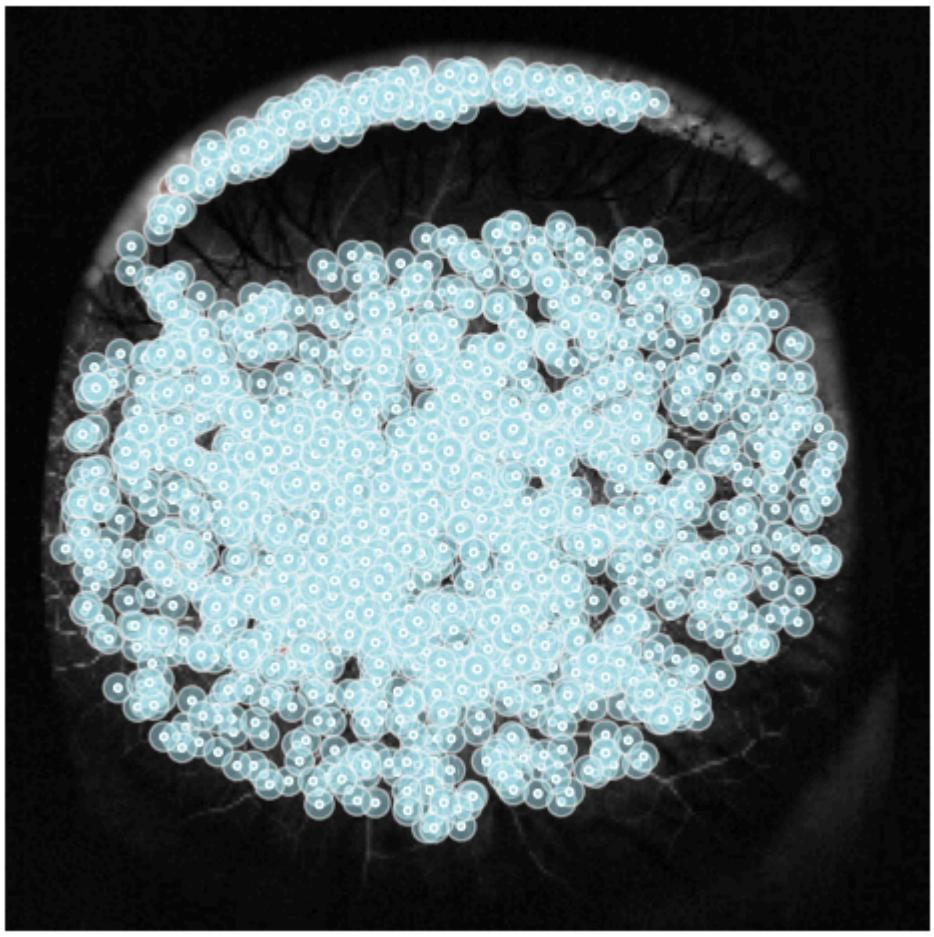




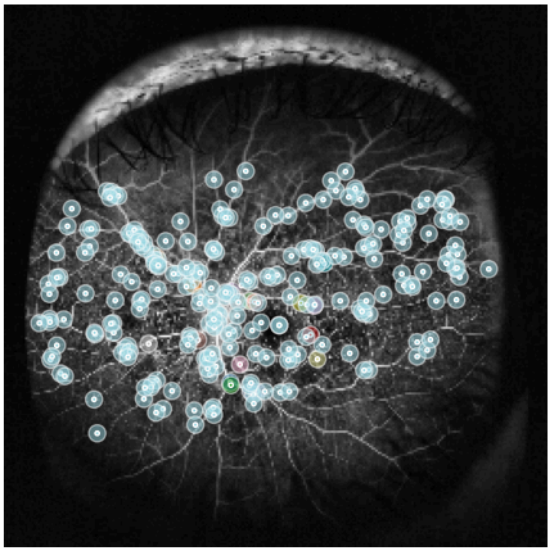



```
Recorded Landmark Error for Iteration 3 is 12.173661032414424
Iteration 4
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_2/Montage/Montage_Subject_2.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_2/FA/Raw_FA_4_Subject_2.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
246
```

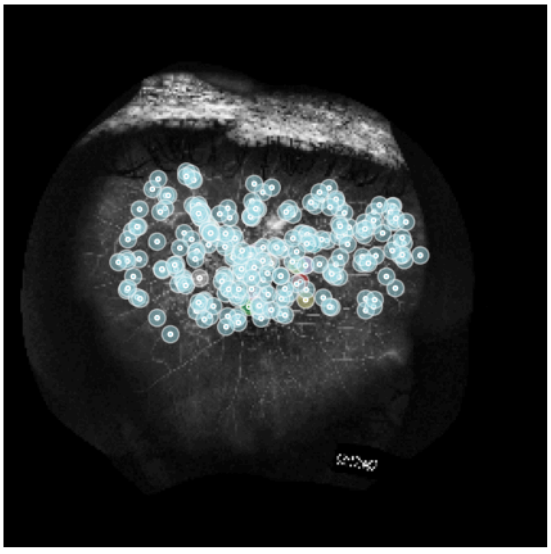
Chosen Keypoints



Source Image

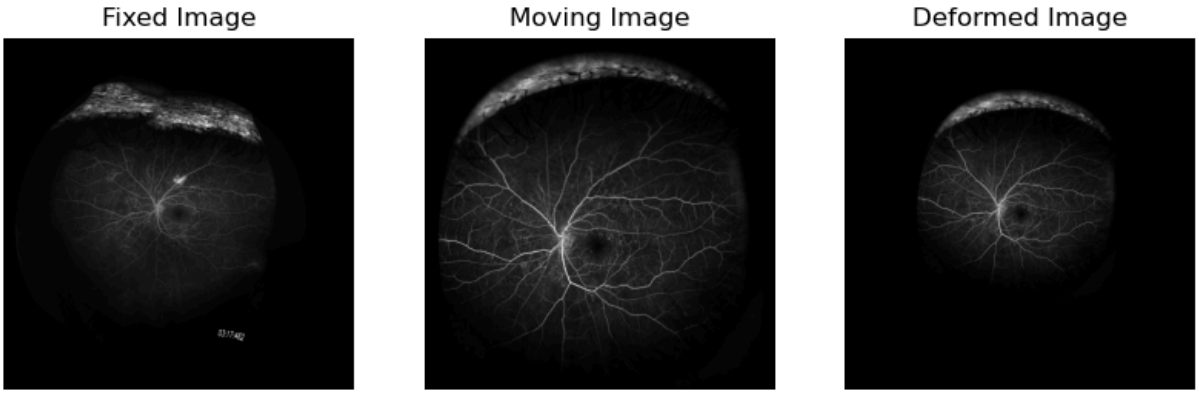


Target Image

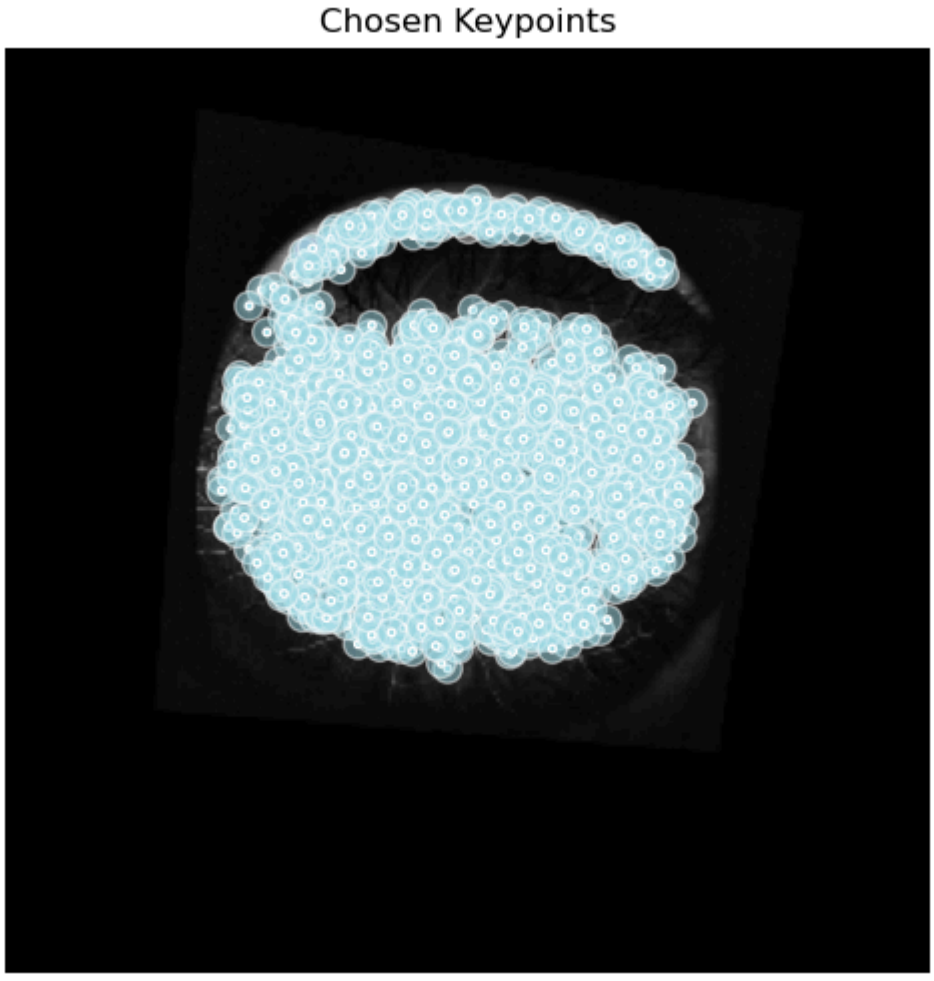


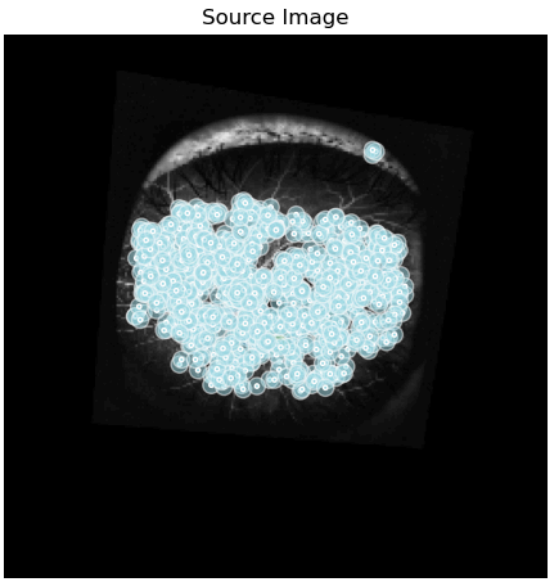
Homography Matrix:

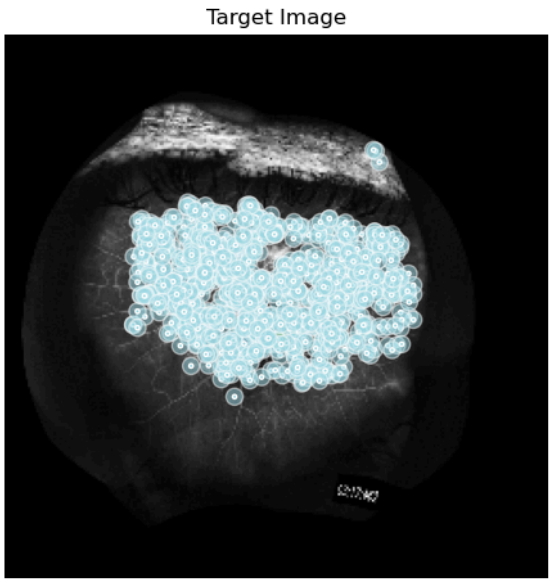
```
[[ 7.43350656e-01 -3.32135363e-02  2.14034752e+02]
 [ 1.29448692e-01  7.13142268e-01  6.70666933e+01]
 [ 1.01583656e-04  8.64135296e-05  1.00000000e+00]]
```

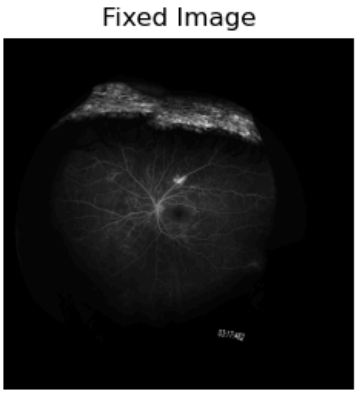


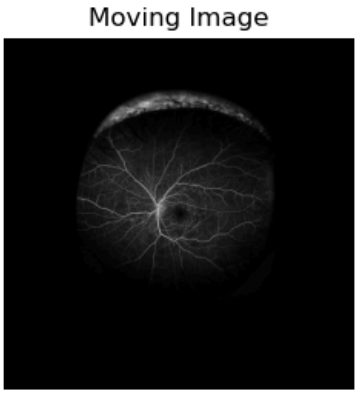

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
930
```

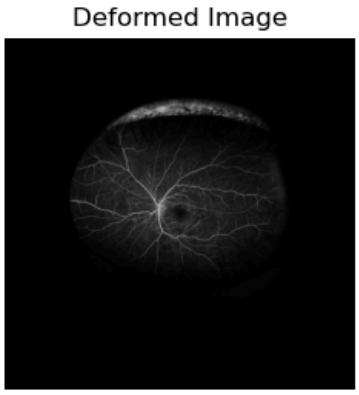


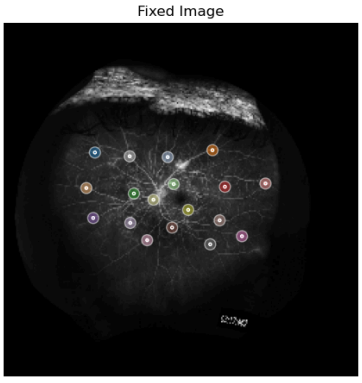


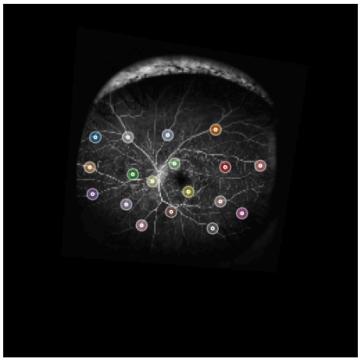


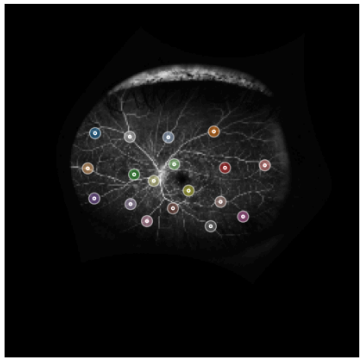






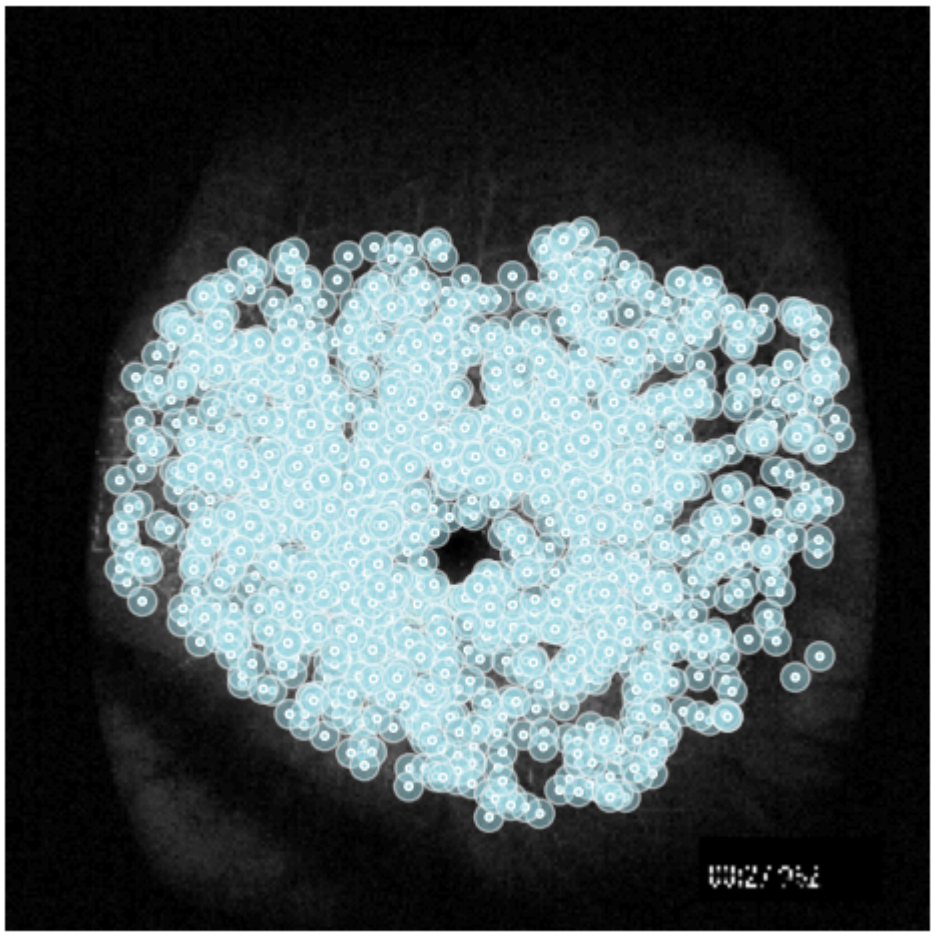




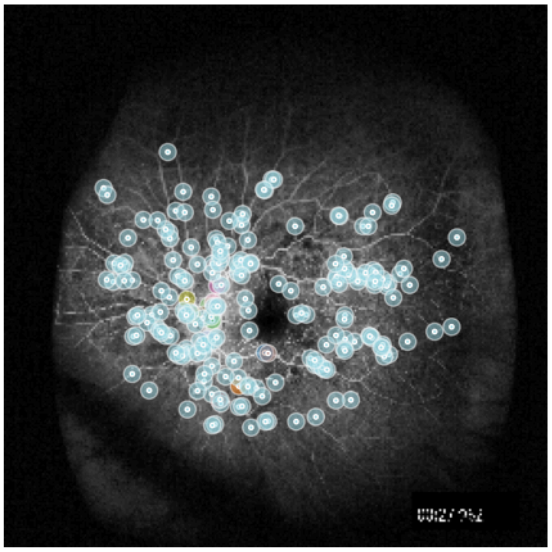



```
Recorded Landmark Error for Iteration 4 is 12.930191960013286
Iteration 5
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_4/FA/Raw_FA_1_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
207
```

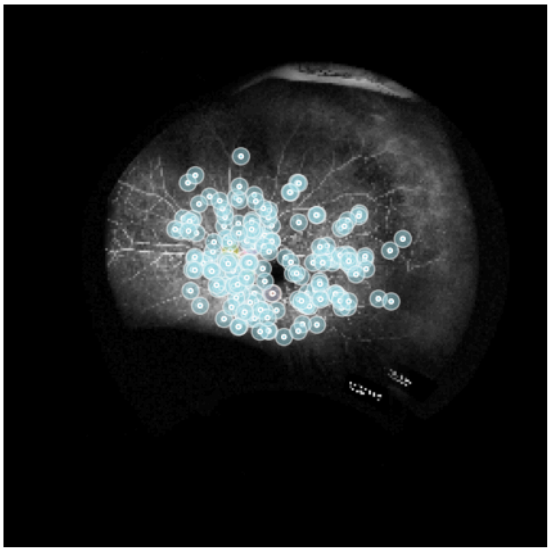
Chosen Keypoints



Source Image



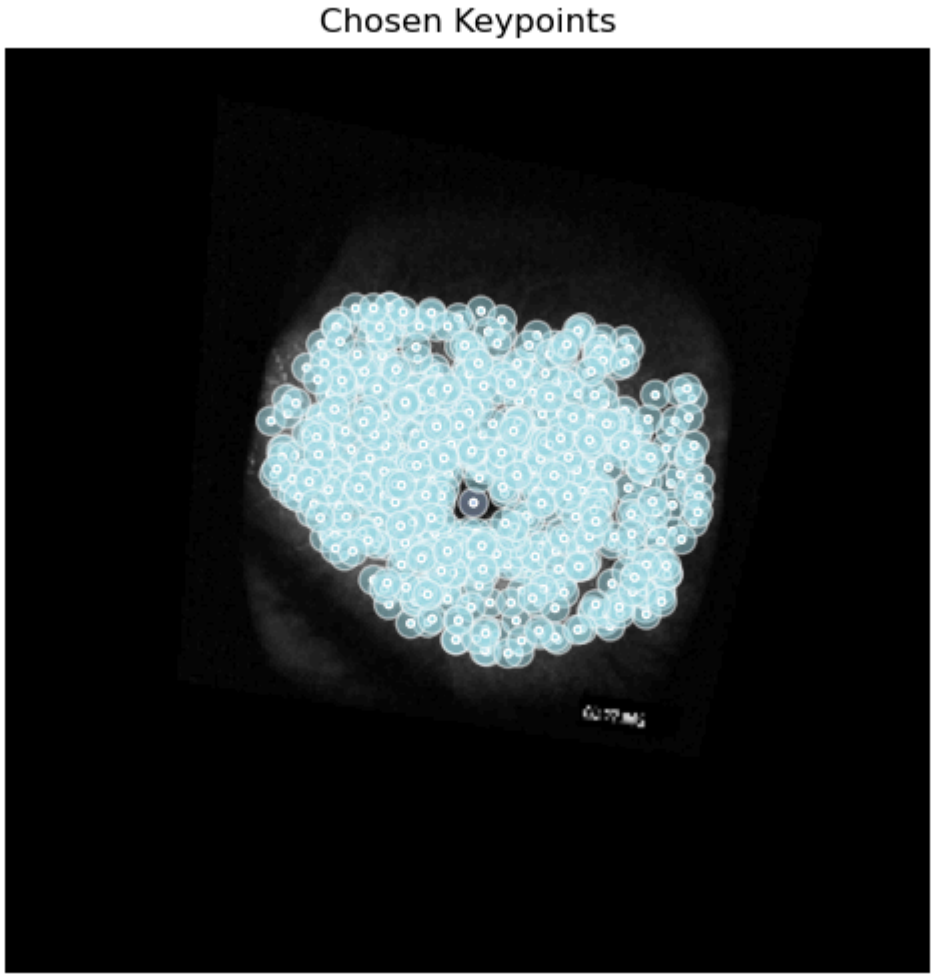
Target Image

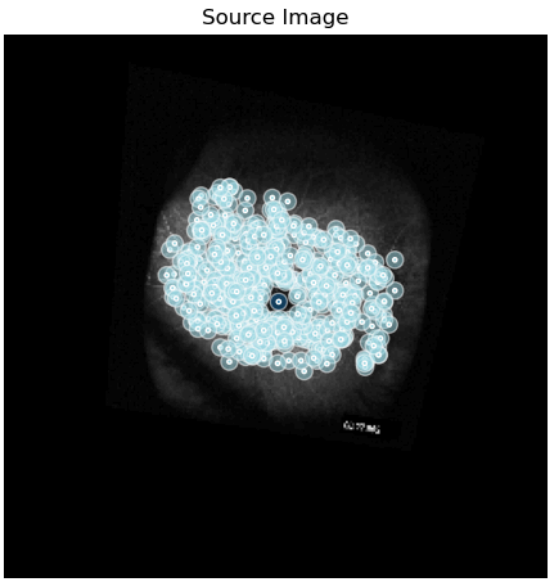


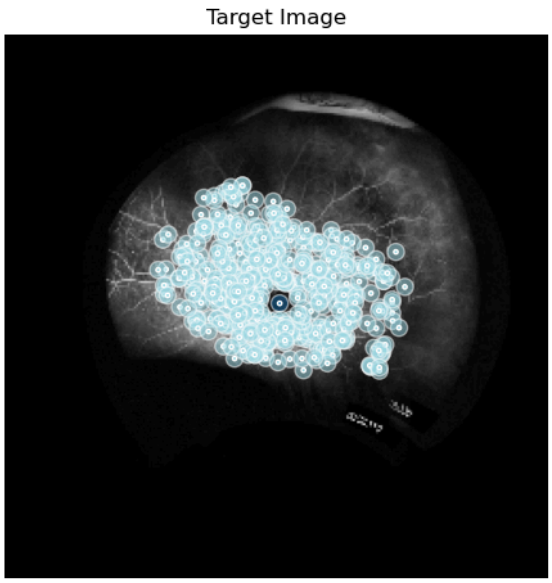
Homography Matrix:

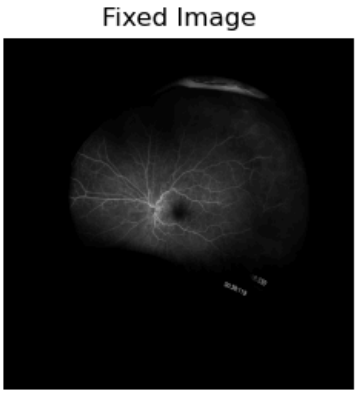
```
[[ 7.10751419e-01 -1.05944534e-02  2.36522902e+02]
 [ 1.49087923e-01  7.52782504e-01  5.38568251e+01]
 [ 6.36777932e-05  1.74113686e-04  1.00000000e+00]]
```

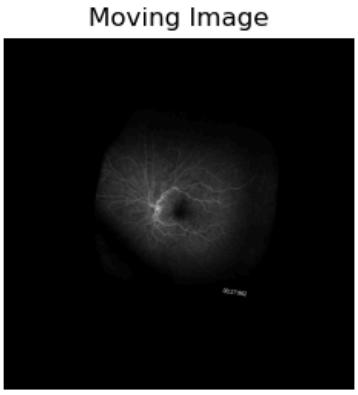

```
Maximum attempts reached, unable to find sufficient points with the specified criter
ia.
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
497
```

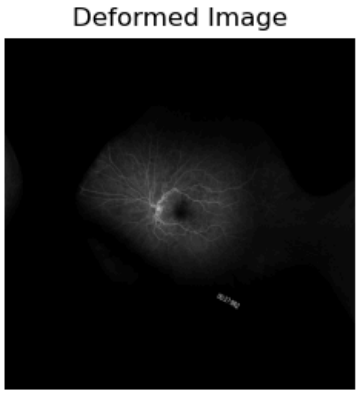


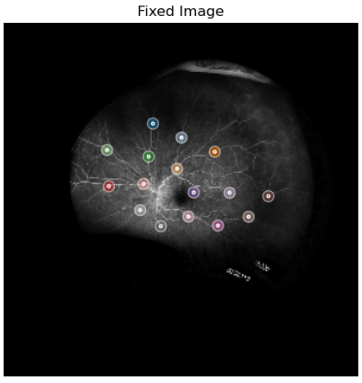


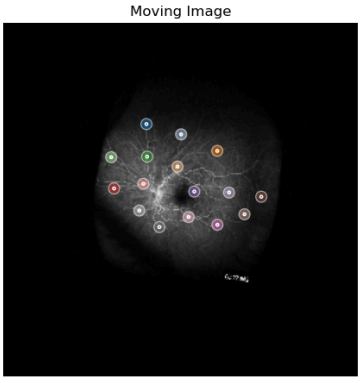


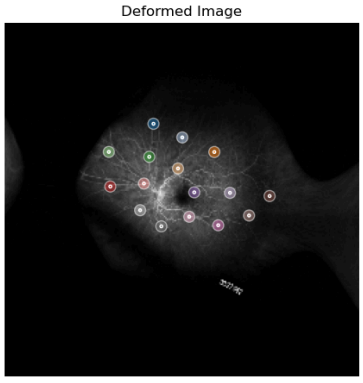






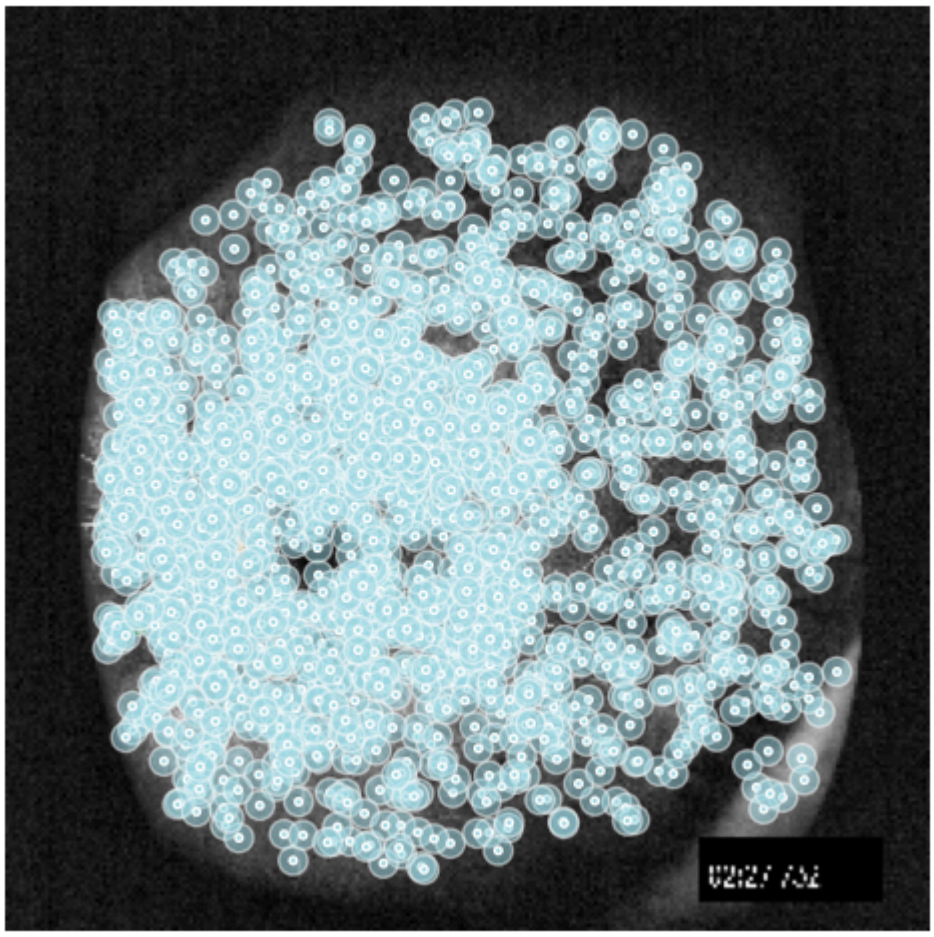




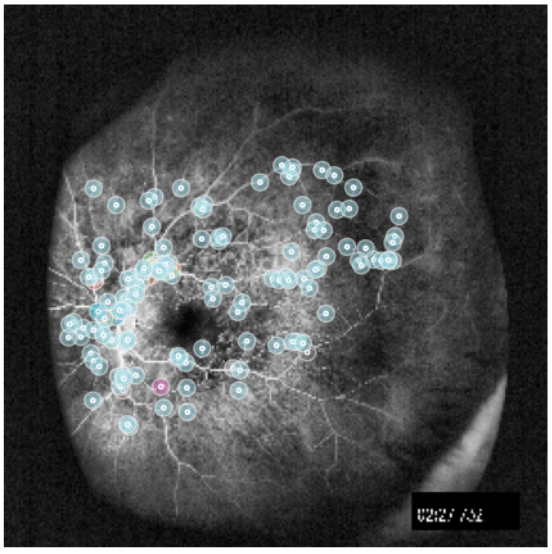



```
Recorded Landmark Error for Iteration 5 is 6.759529169360167
Iteration 6
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_4/FA/Raw_FA_2_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
138
```

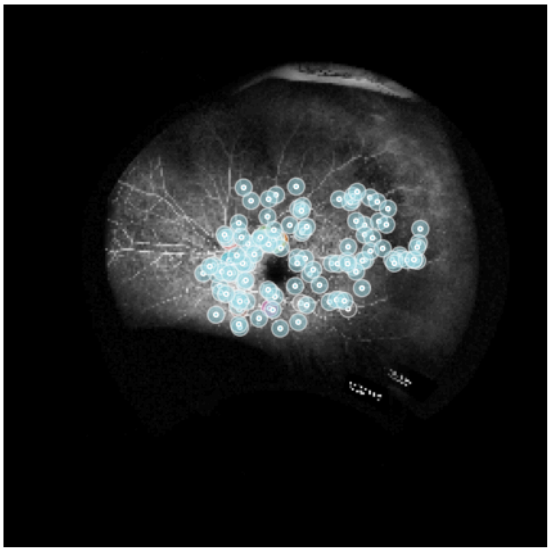
Chosen Keypoints



Source Image

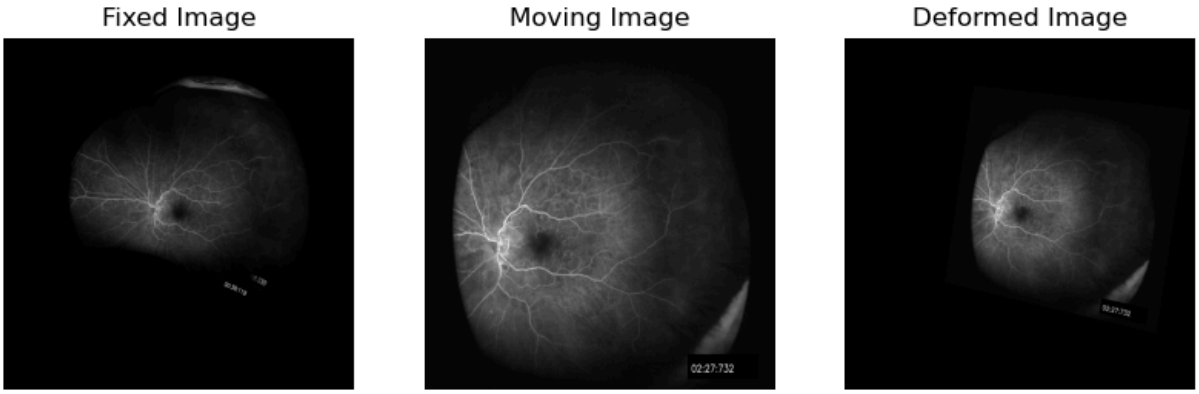


Target Image

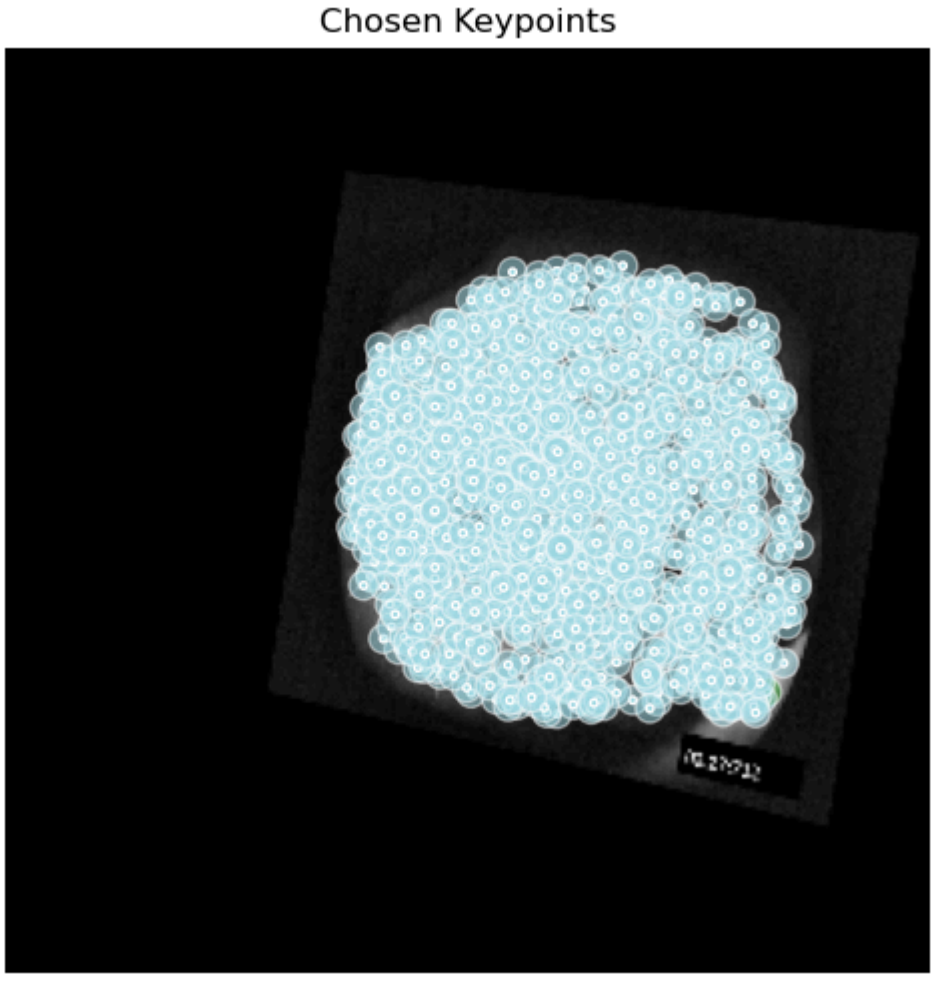


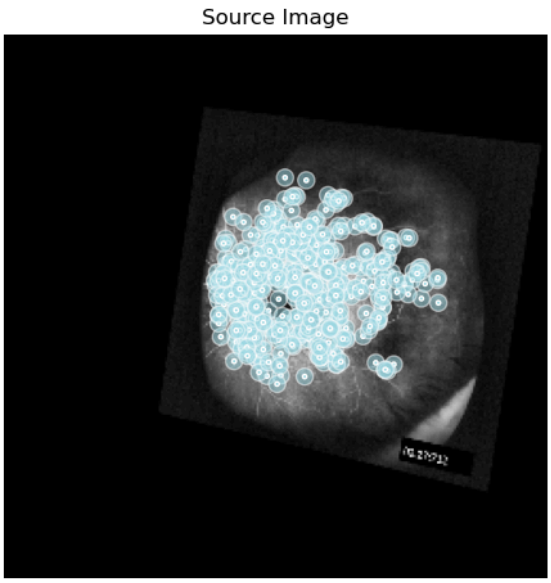
Homography Matrix:

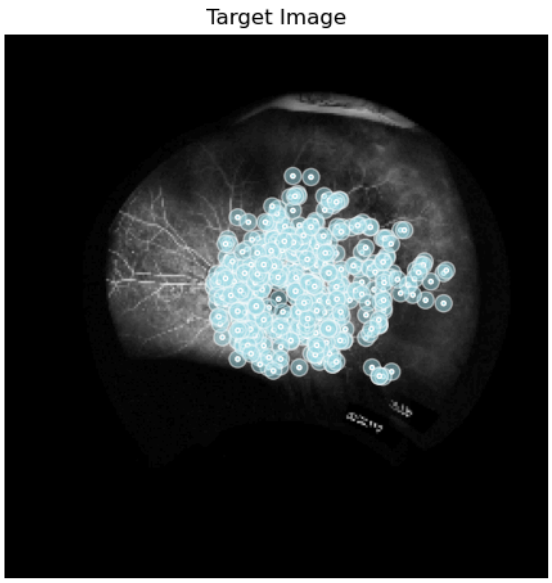
```
[[ 5.00755555e-01 -8.19523012e-02  3.76277891e+02]
 [ 4.43979272e-02  5.66042031e-01  1.36262431e+02]
 [-1.18480918e-04  4.92888940e-06  1.00000000e+00]]
```

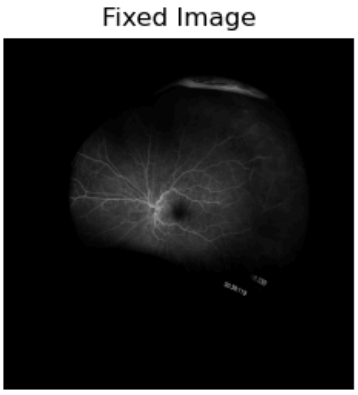


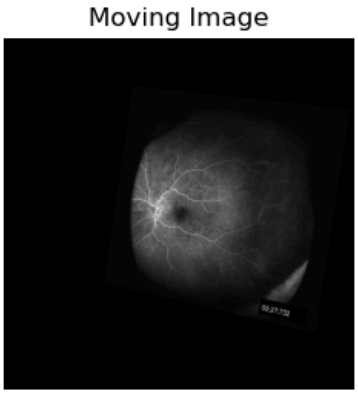

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
507
```

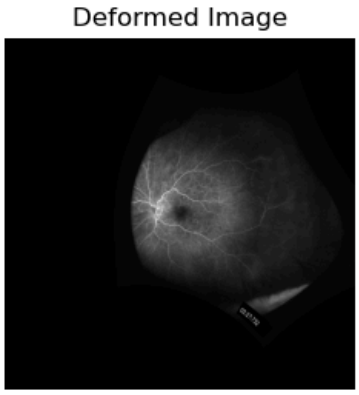


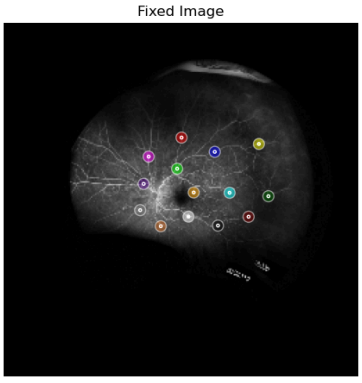


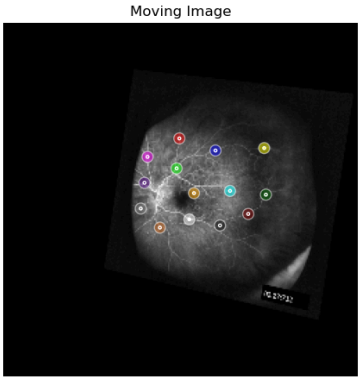


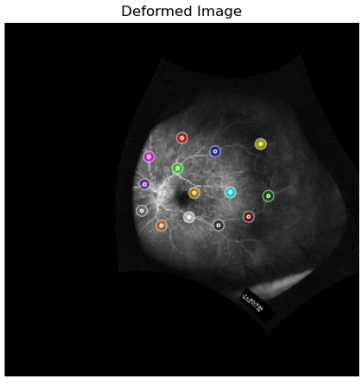






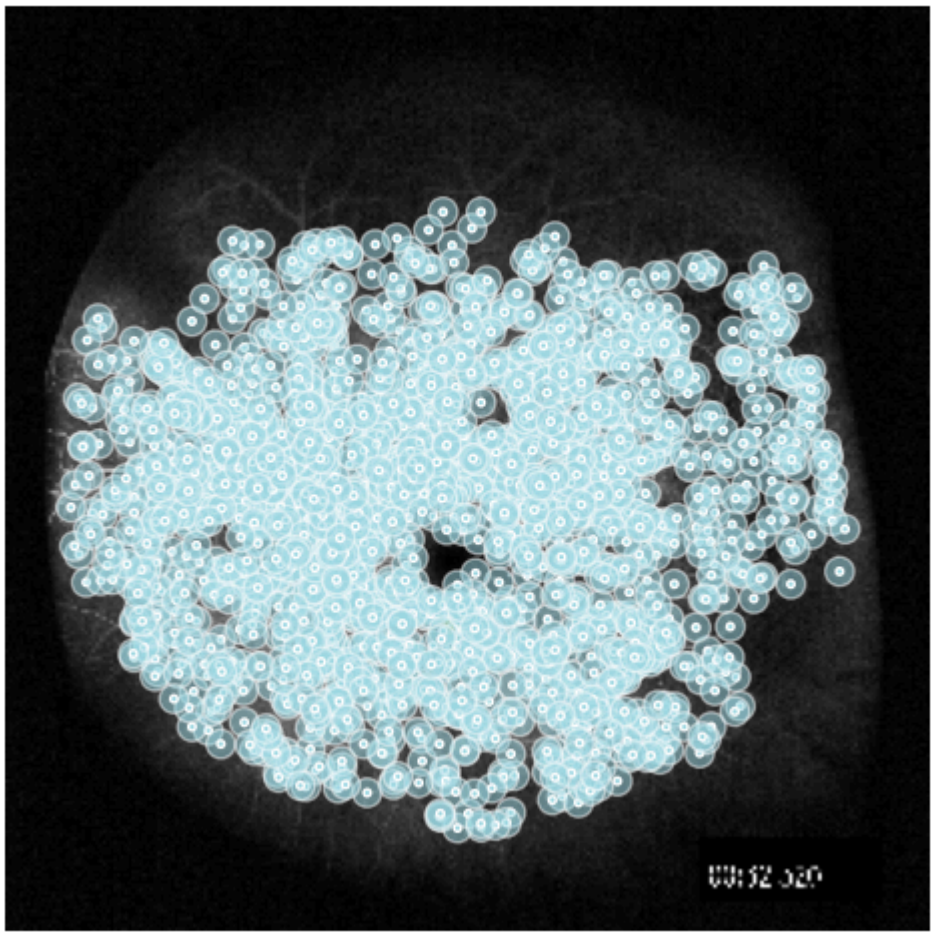




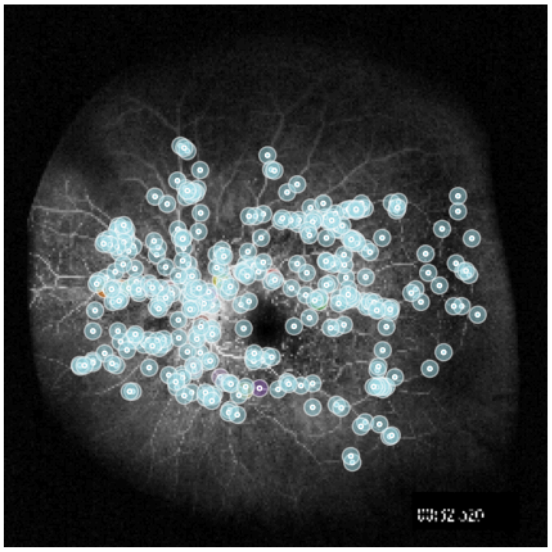



```
Recorded Landmark Error for Iteration 6 is 7.769554363418775
Iteration 7
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_4/Montage/Montage_Subject_4.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_4/FA/Raw_FA_3_Subject_4.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
346
```

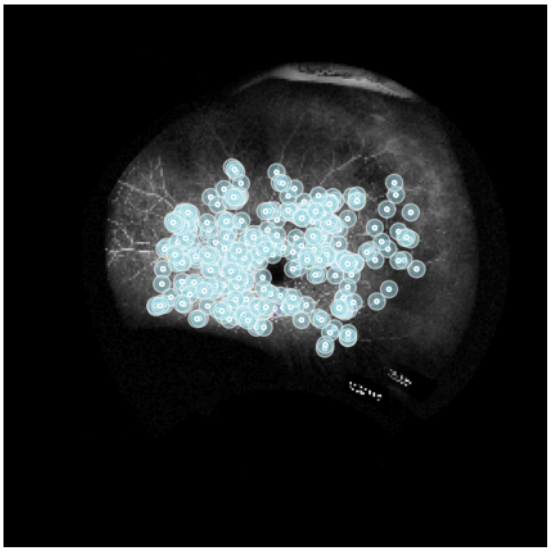
Chosen Keypoints



Source Image



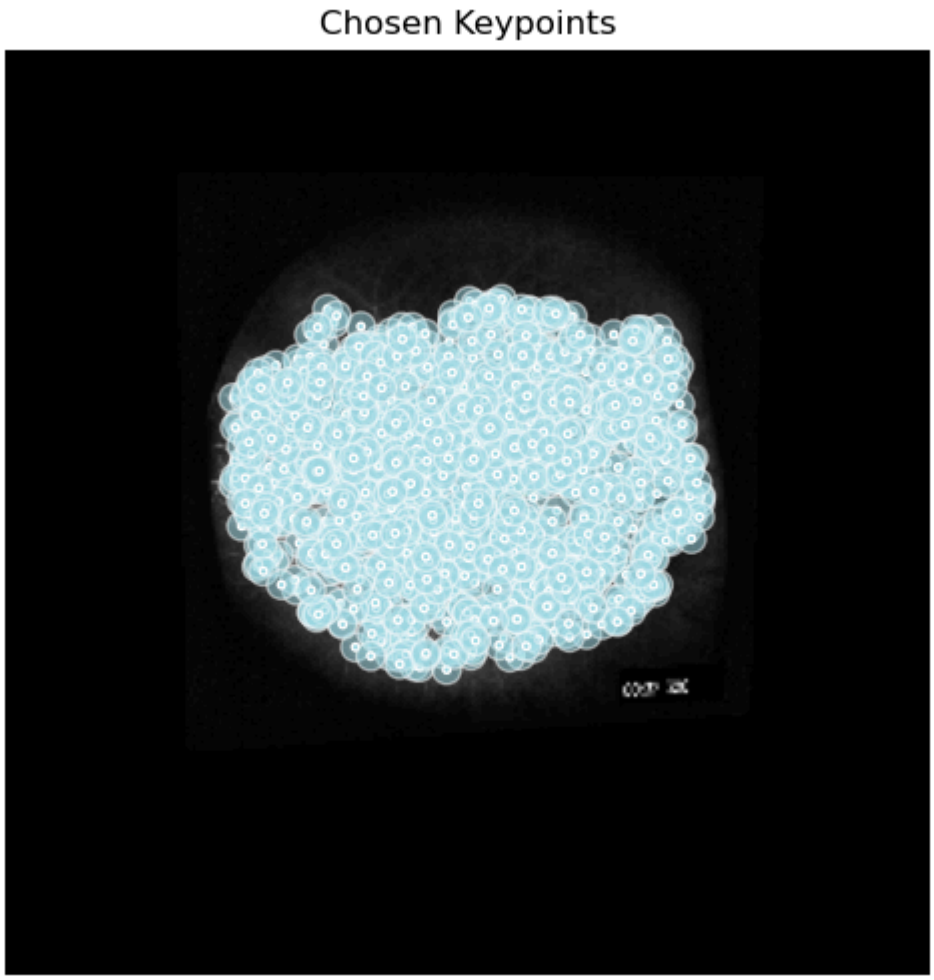
Target Image

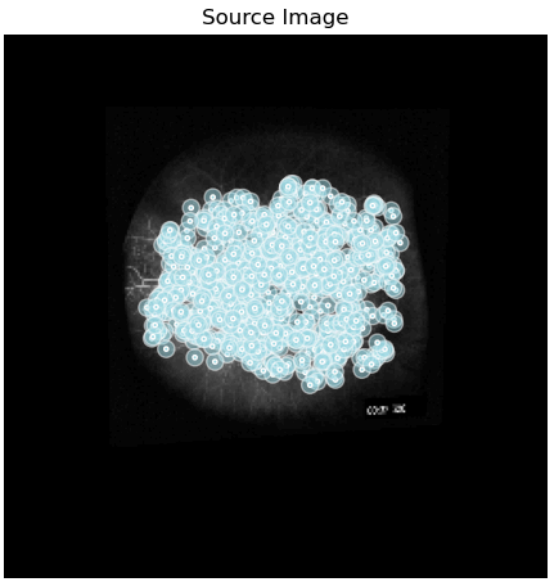


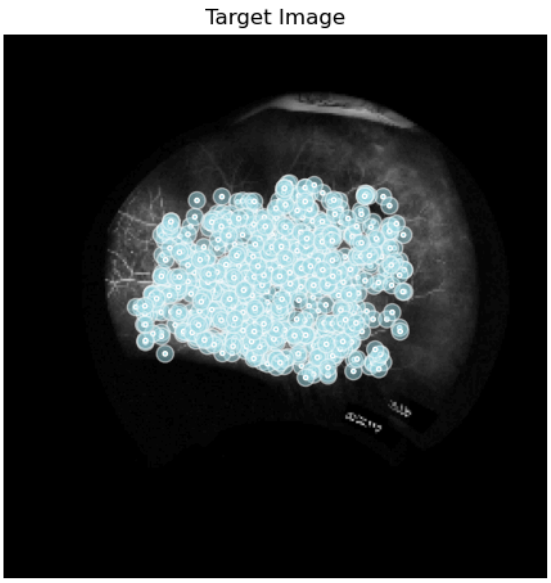
Homography Matrix:

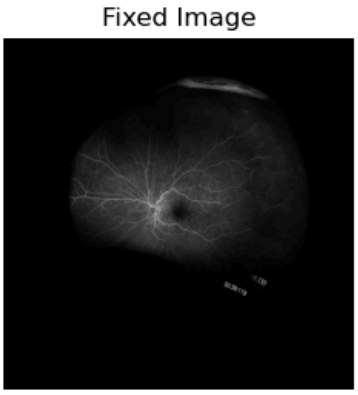
```
[[7.03264585e-01 1.98972435e-02 1.90334265e+02]
 [1.67179000e-02 6.60301667e-01 1.35100557e+02]
 [8.23555659e-05 4.32838897e-05 1.00000000e+00]]
```

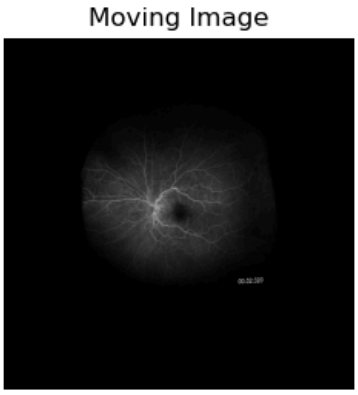

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
840
```

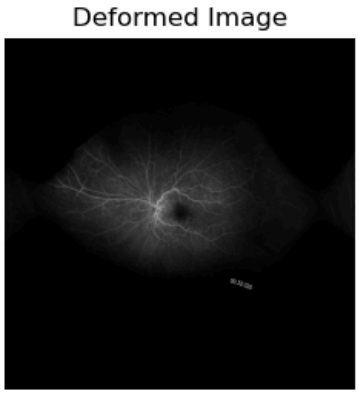


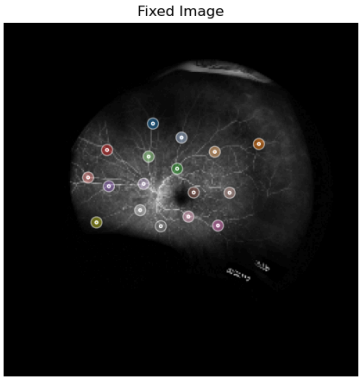


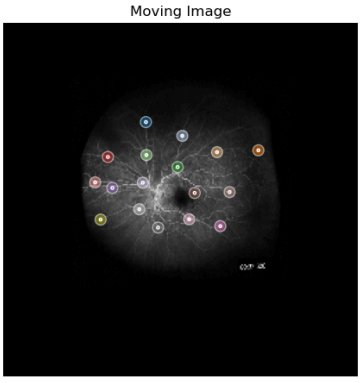


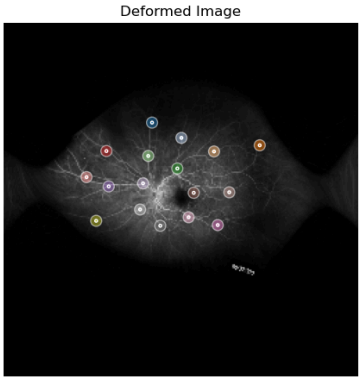






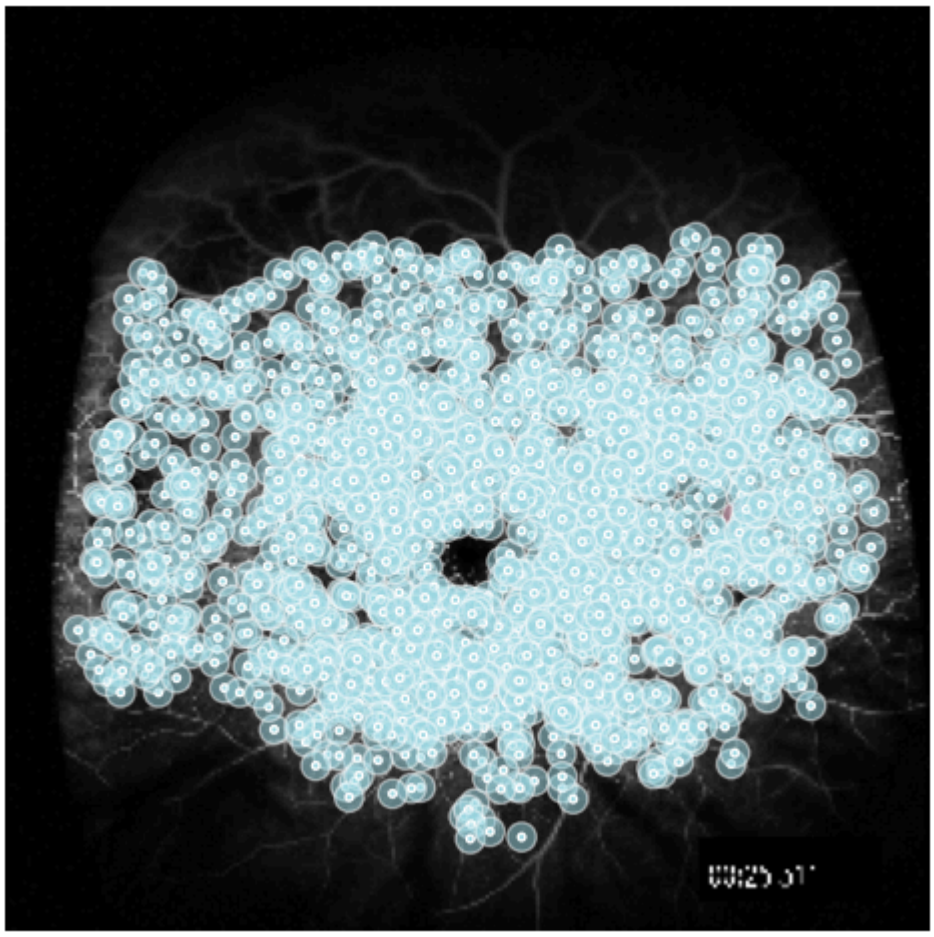




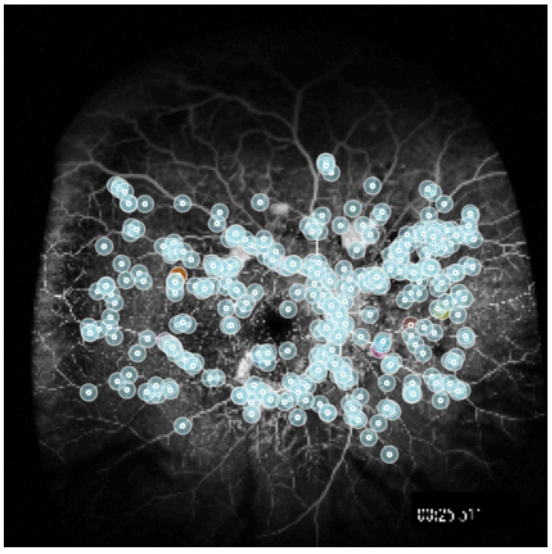



```
Recorded Landmark Error for Iteration 7 is 14.708587917366152
Iteration 8
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_3/Montage/Montage_Subject_3.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_3/FA/Raw_FA_1_Subject_3.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
361
```

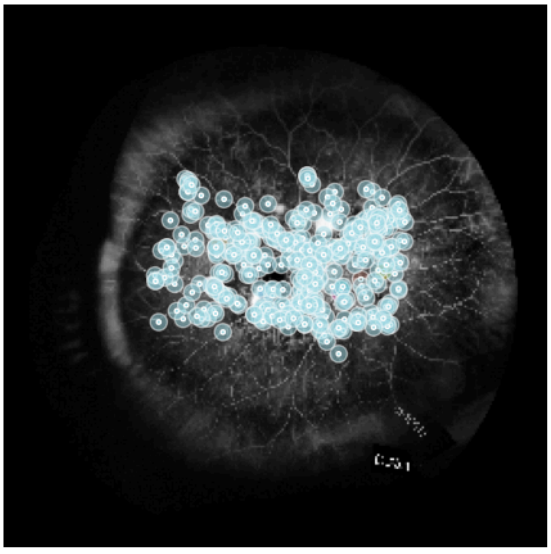
Chosen Keypoints



Source Image



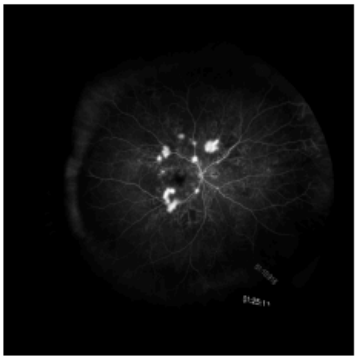
Target Image



Homography Matrix:

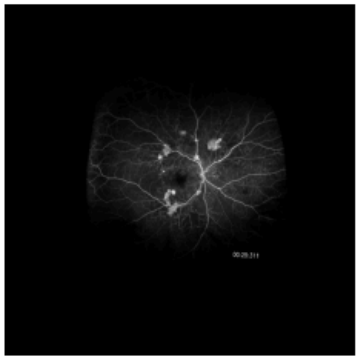
```
[[ 5.95139254e-01  3.16401047e-02  1.99392701e+02]
 [-1.22940162e-02  6.66896211e-01  1.25509137e+02]
 [-6.18260419e-05  9.10222909e-05  1.00000000e+00]]
```

Fixed Image



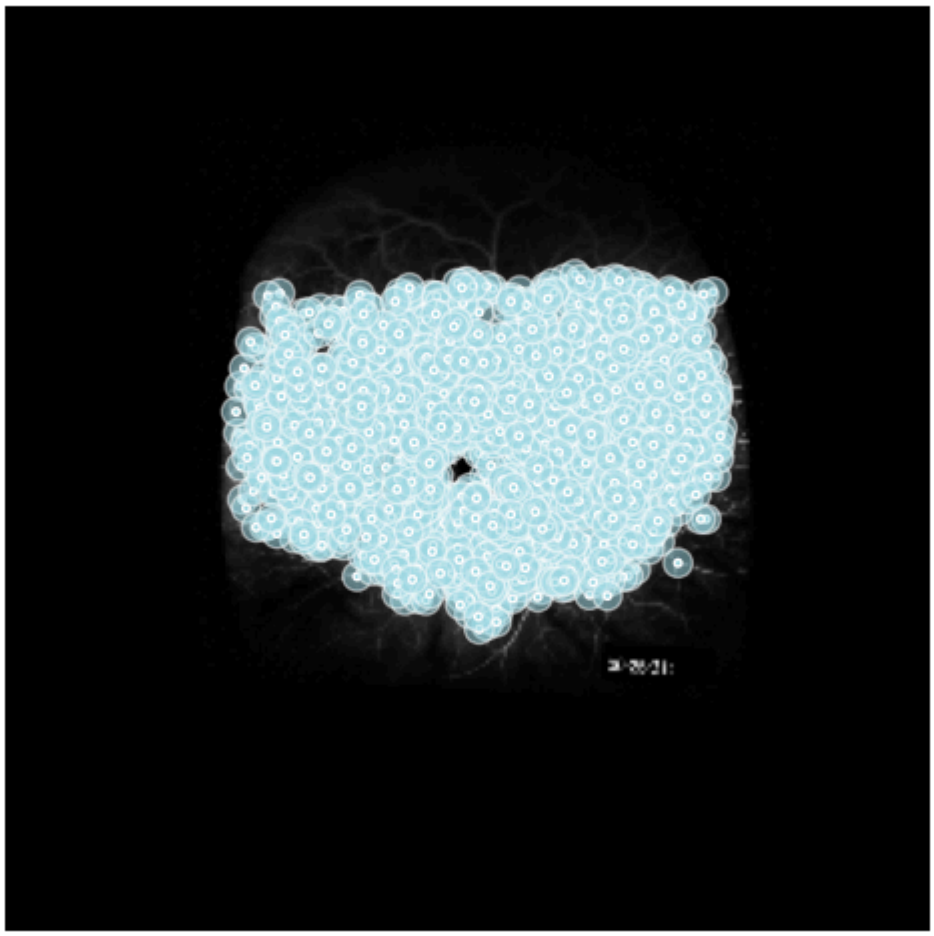
Moving Image

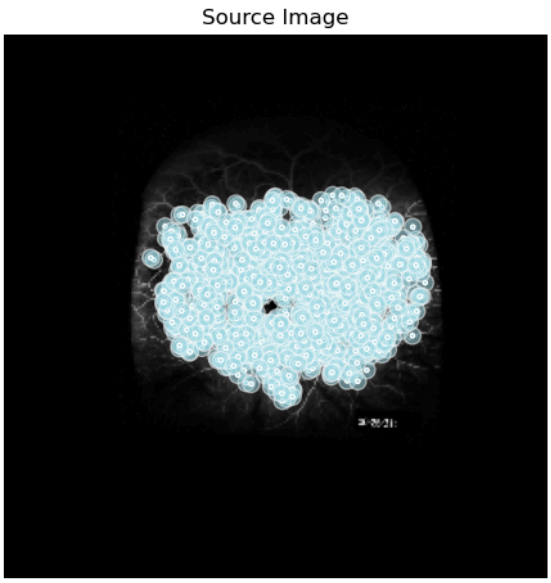
Deformed Image

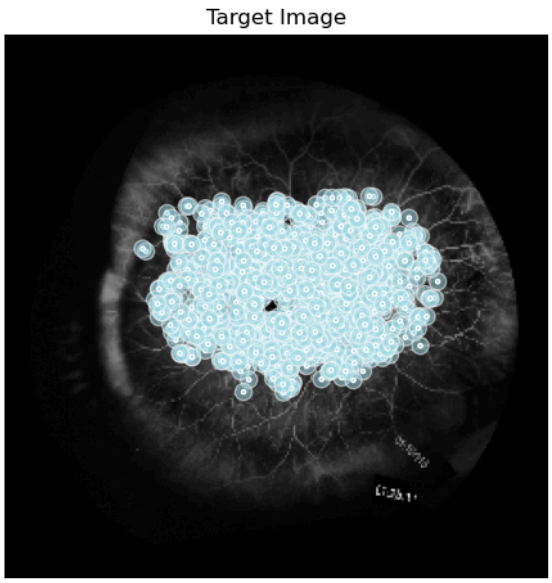


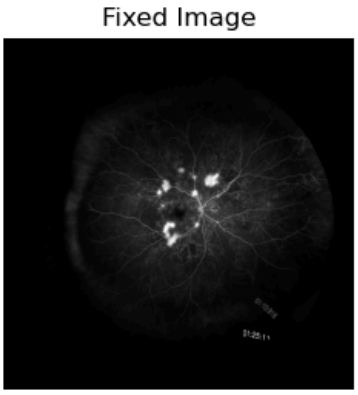
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
1208
```

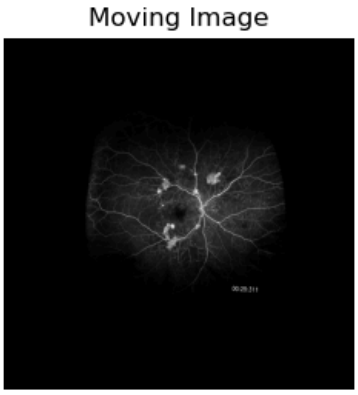
Chosen Keypoints

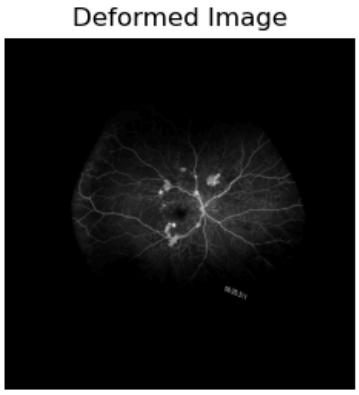


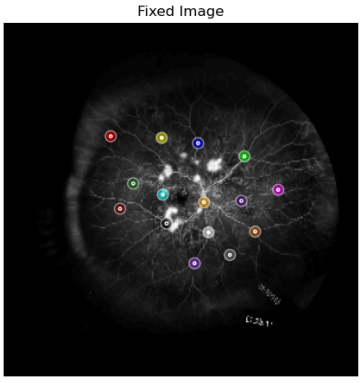


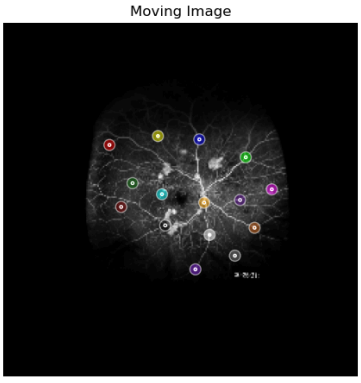


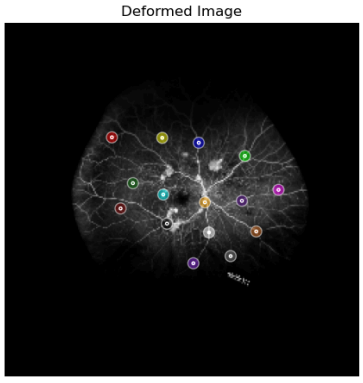






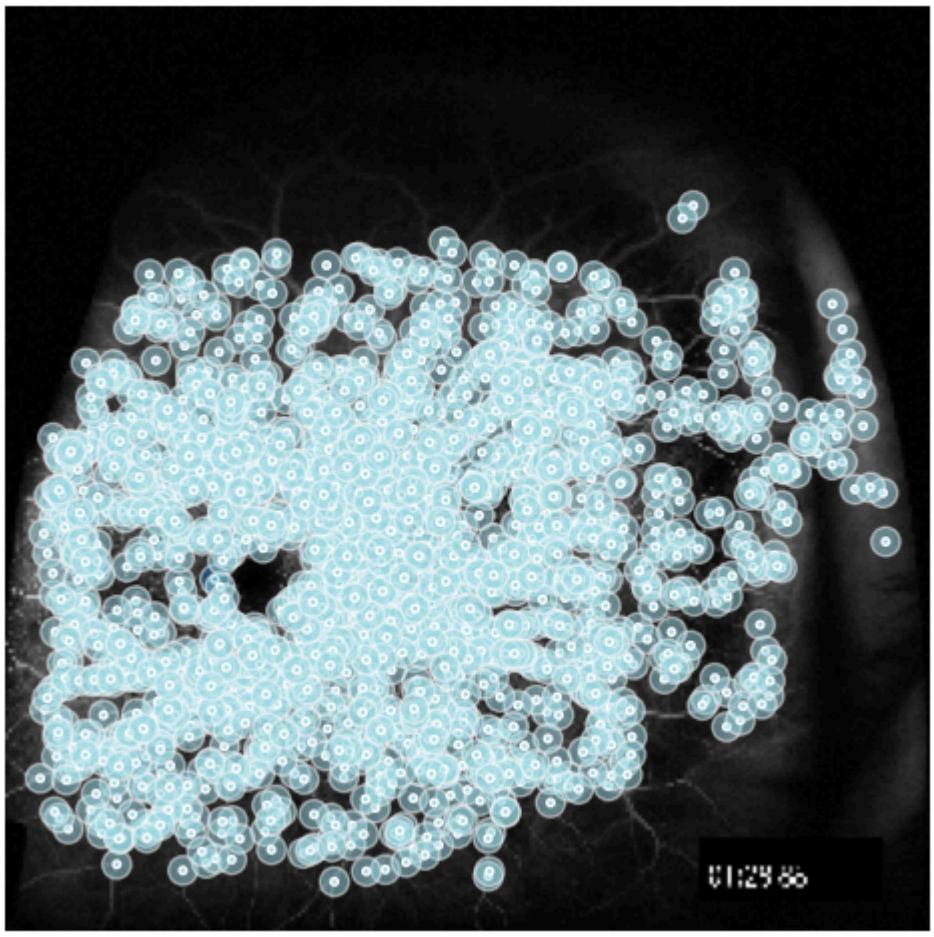




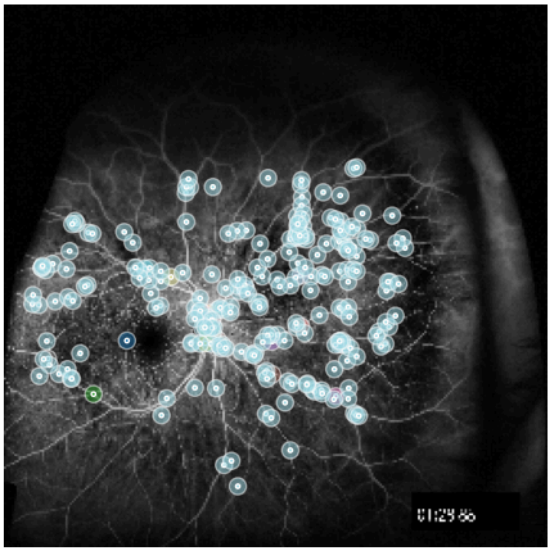



```
Recorded Landmark Error for Iteration 8 is 19.67213024057926
Iteration 9
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_3/Montage/Montage_Subject_3.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_3/FA/Raw_FA_2_Subject_3.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
240
```

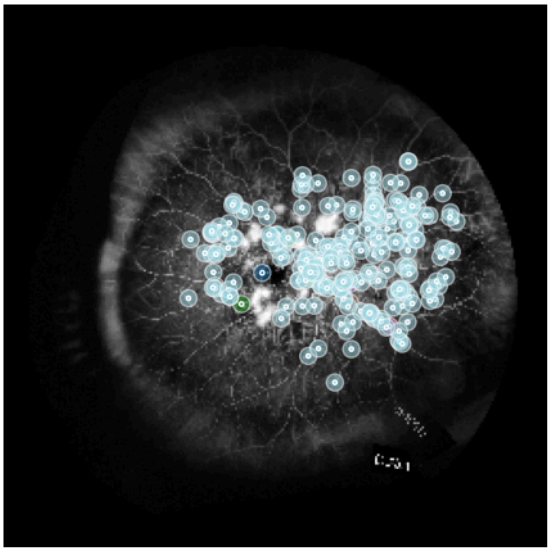
Chosen Keypoints



Source Image



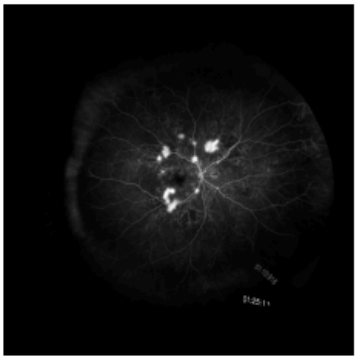
Target Image



Homography Matrix:

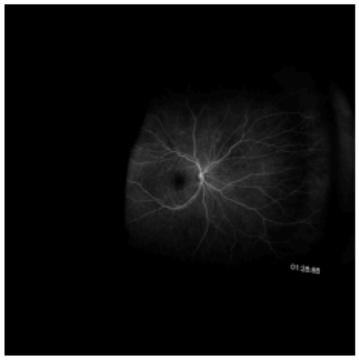
```
[[ 4.71634809e-01  1.56899535e-02  3.60032279e+02]
 [-7.84514268e-02  6.46447194e-01  1.17906857e+02]
 [-2.38292830e-04  7.63805793e-05  1.00000000e+00]]
```

Fixed Image



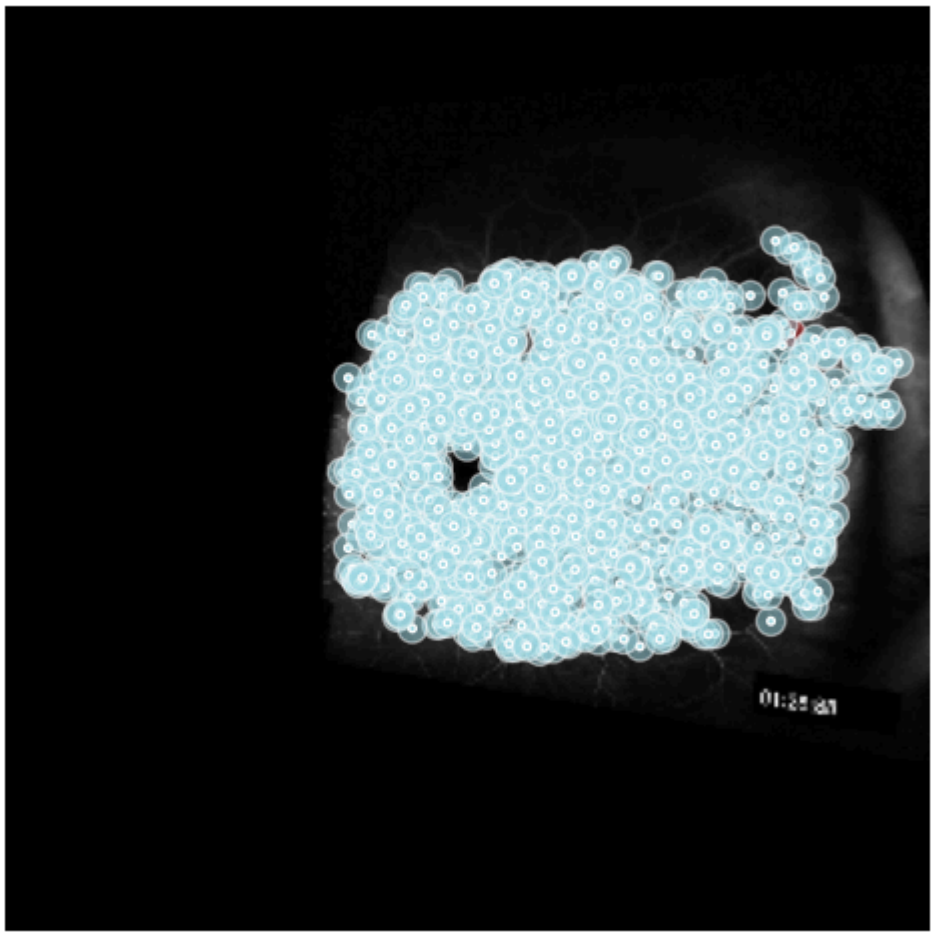
Moving Image

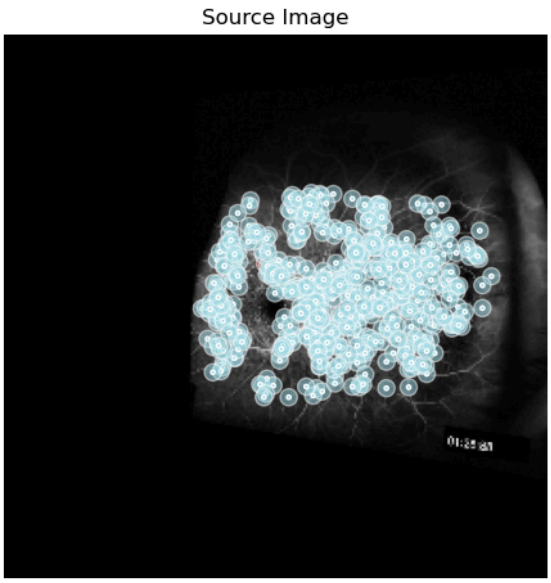
Deformed Image

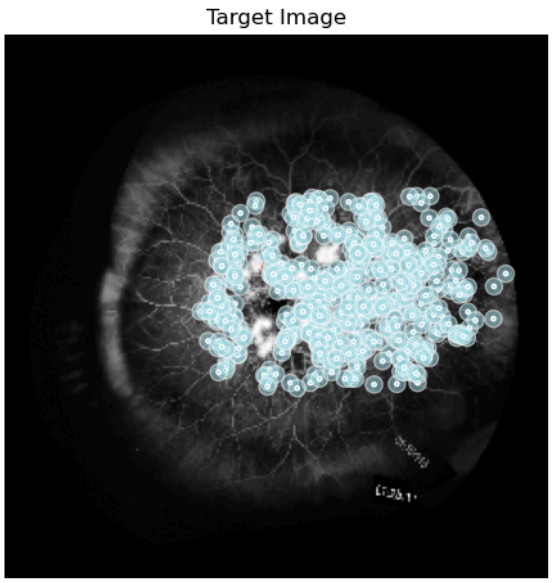


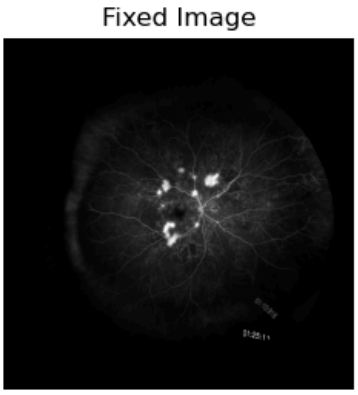
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
573
```

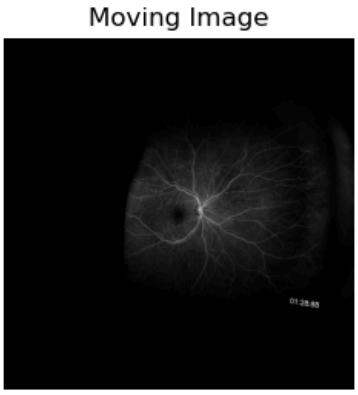
Chosen Keypoints

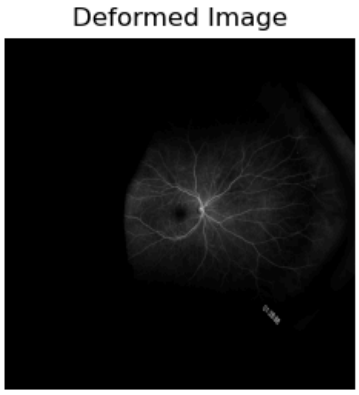


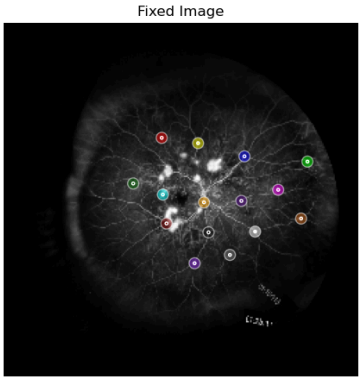


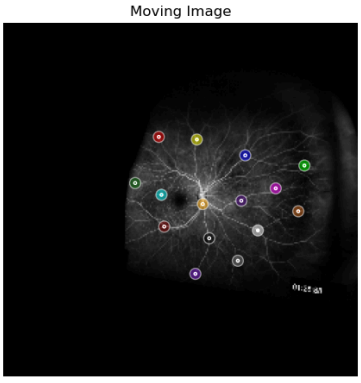


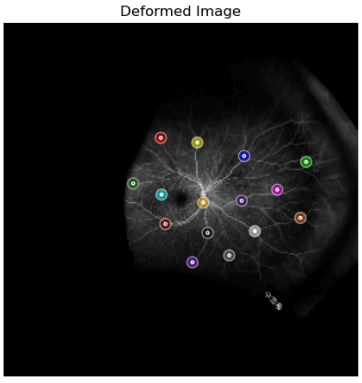






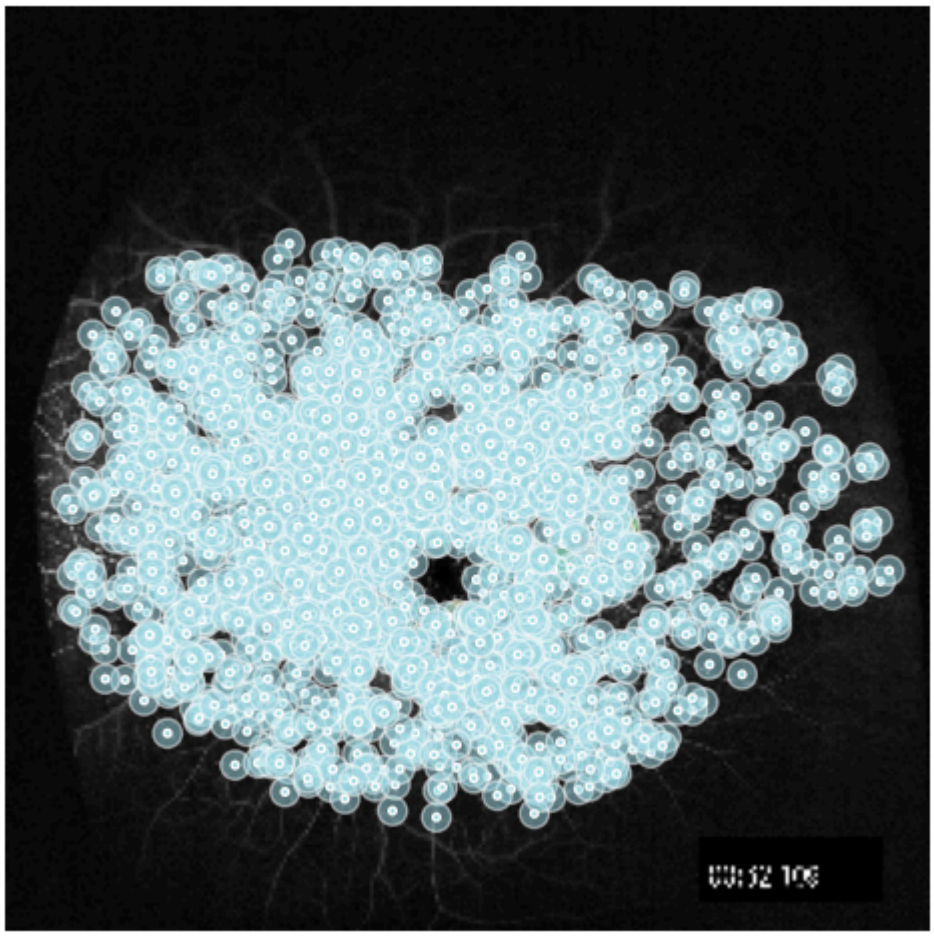




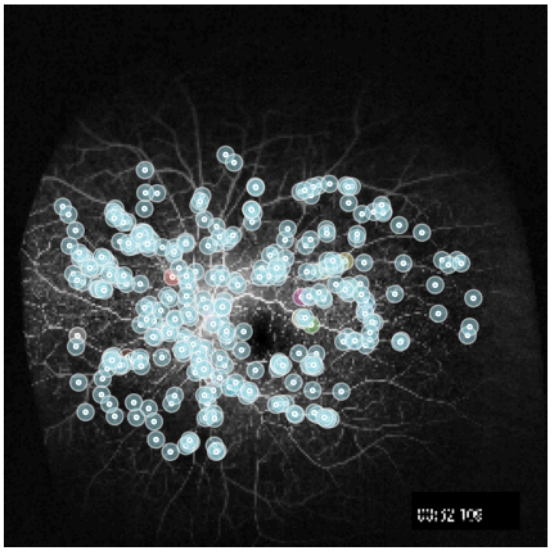



```
Recorded Landmark Error for Iteration 9 is 11.877064069272913
Iteration 10
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_1/FA/Raw_FA_1_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
331
```

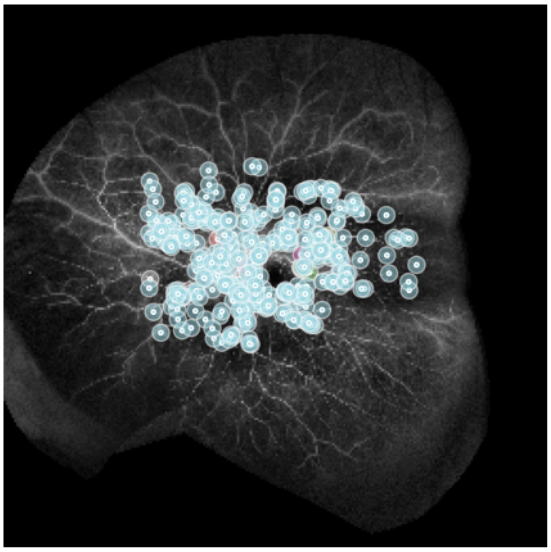
Chosen Keypoints



Source Image

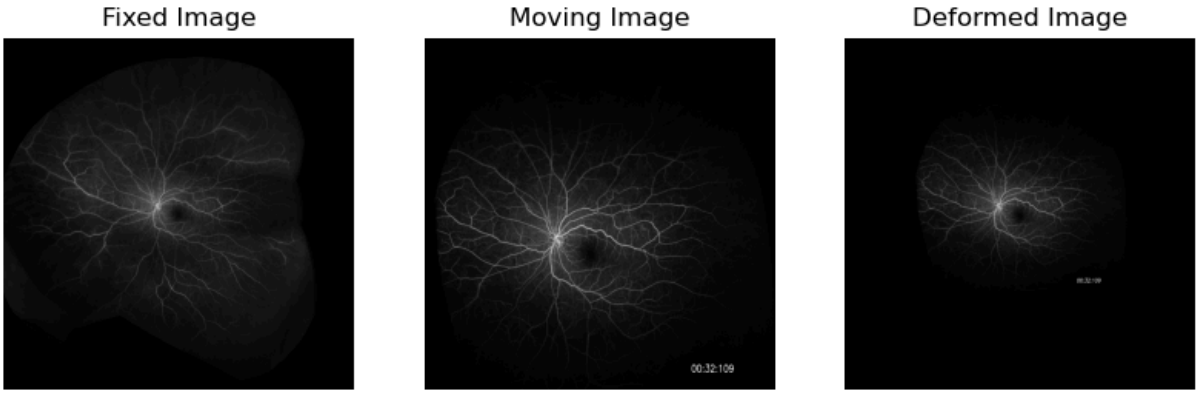


Target Image

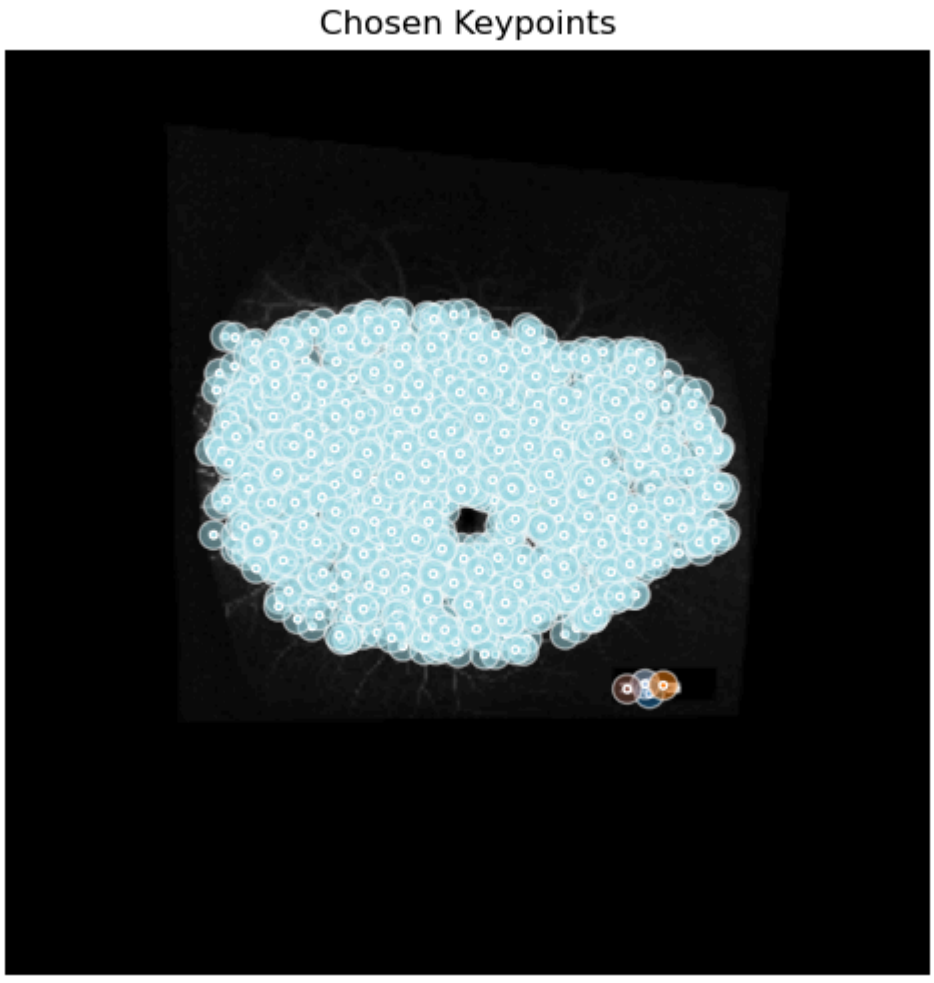


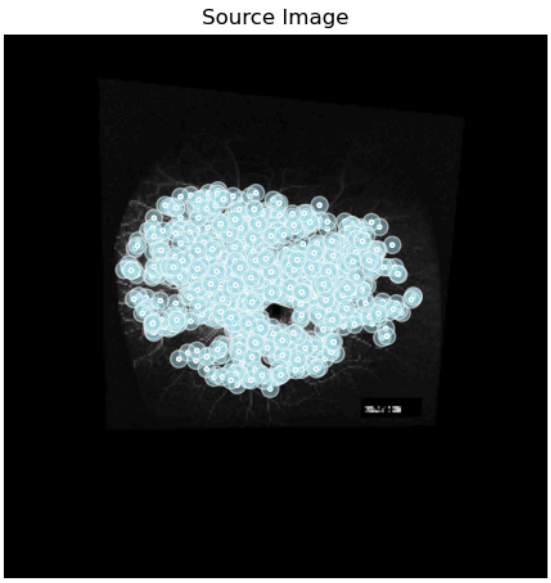
Homography Matrix:

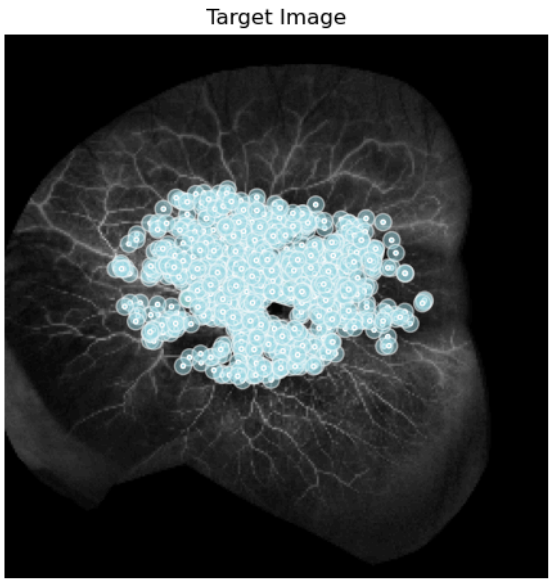
```
[[7.92116801e-01 3.93065893e-02 1.78313558e+02]
 [9.35385789e-02 7.39573886e-01 8.18451278e+01]
 [1.37492727e-04 1.25432925e-04 1.00000000e+00]]
```

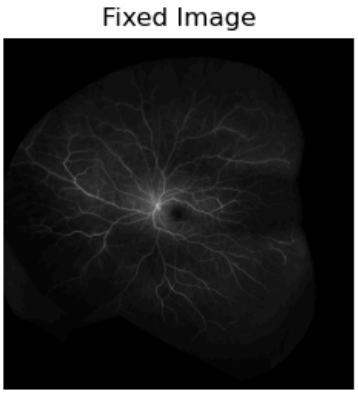


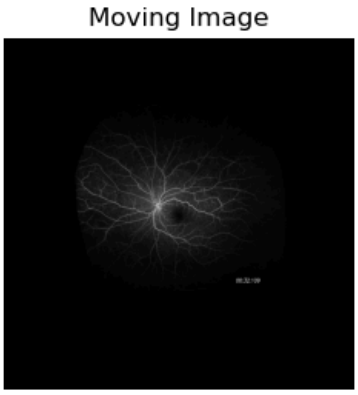

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
882
```

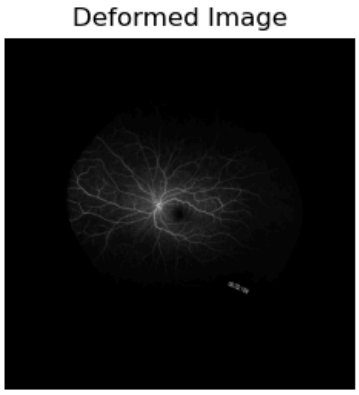


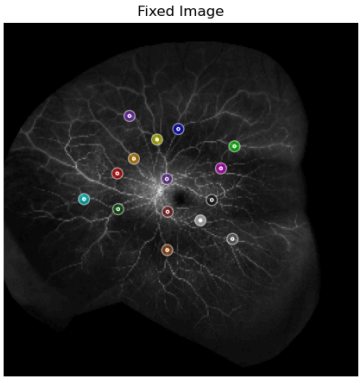


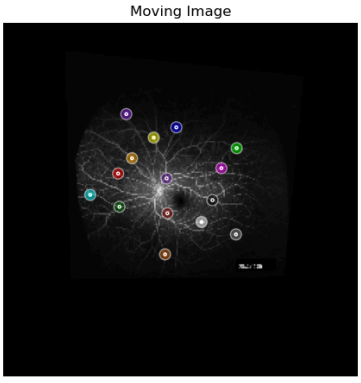


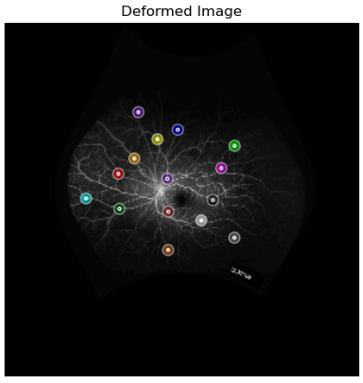






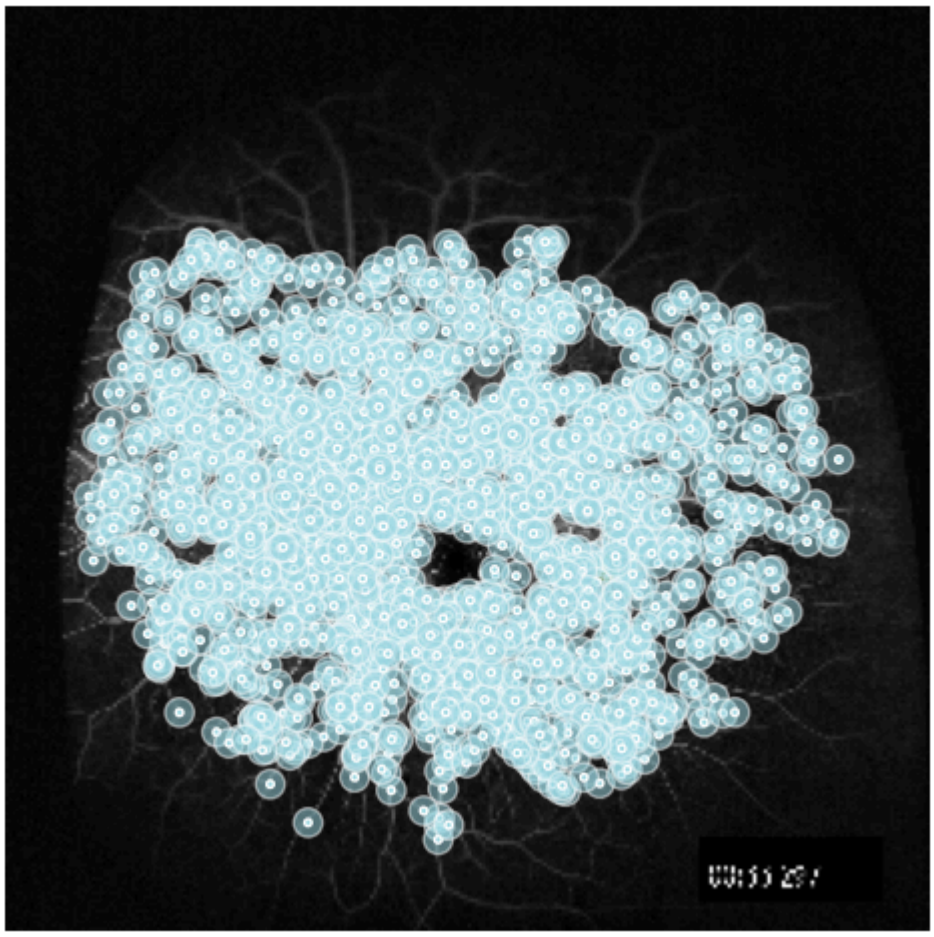




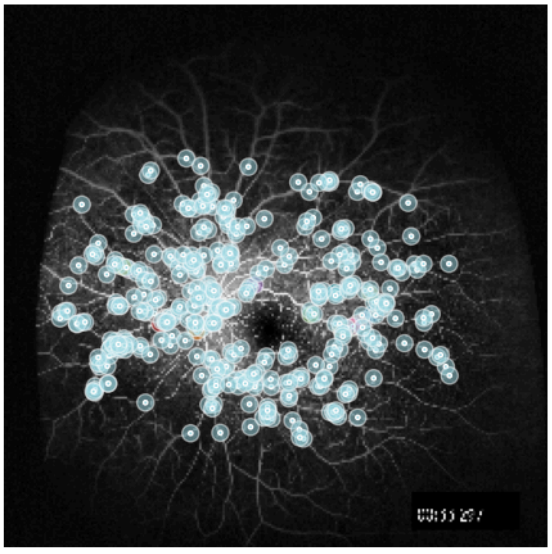



```
Recorded Landmark Error for Iteration 10 is 16.697139474300744
Iteration 11
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_1/FA/Raw_FA_2_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
358
```

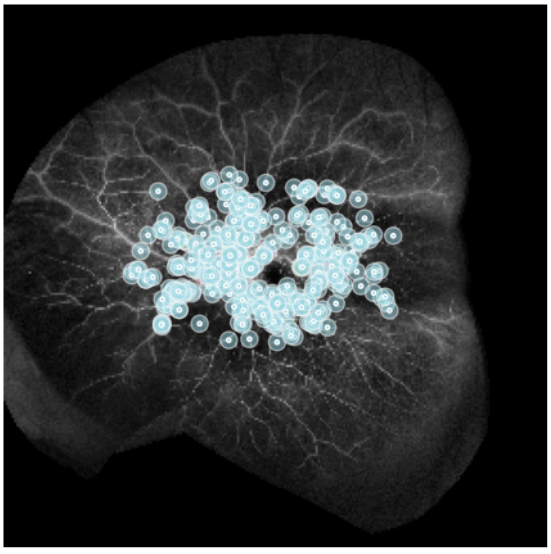
Chosen Keypoints



Source Image



Target Image

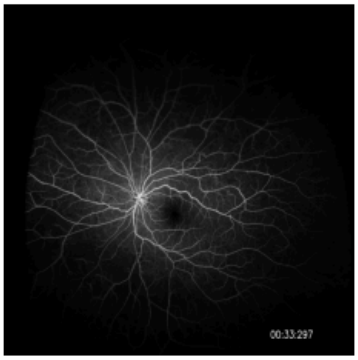


Homography Matrix:

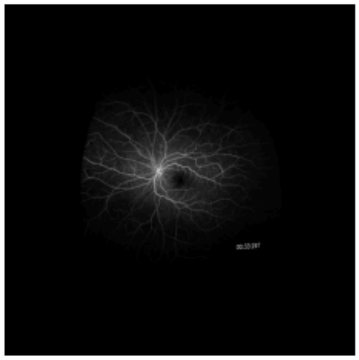
```
[[ 7.62427676e-01 -7.93999848e-03  1.81686540e+02]
 [ 6.90747642e-02  6.44927869e-01  1.28282144e+02]
 [ 1.71301682e-04  8.61617945e-07  1.00000000e+00]]
```

Fixed Image

Moving Image

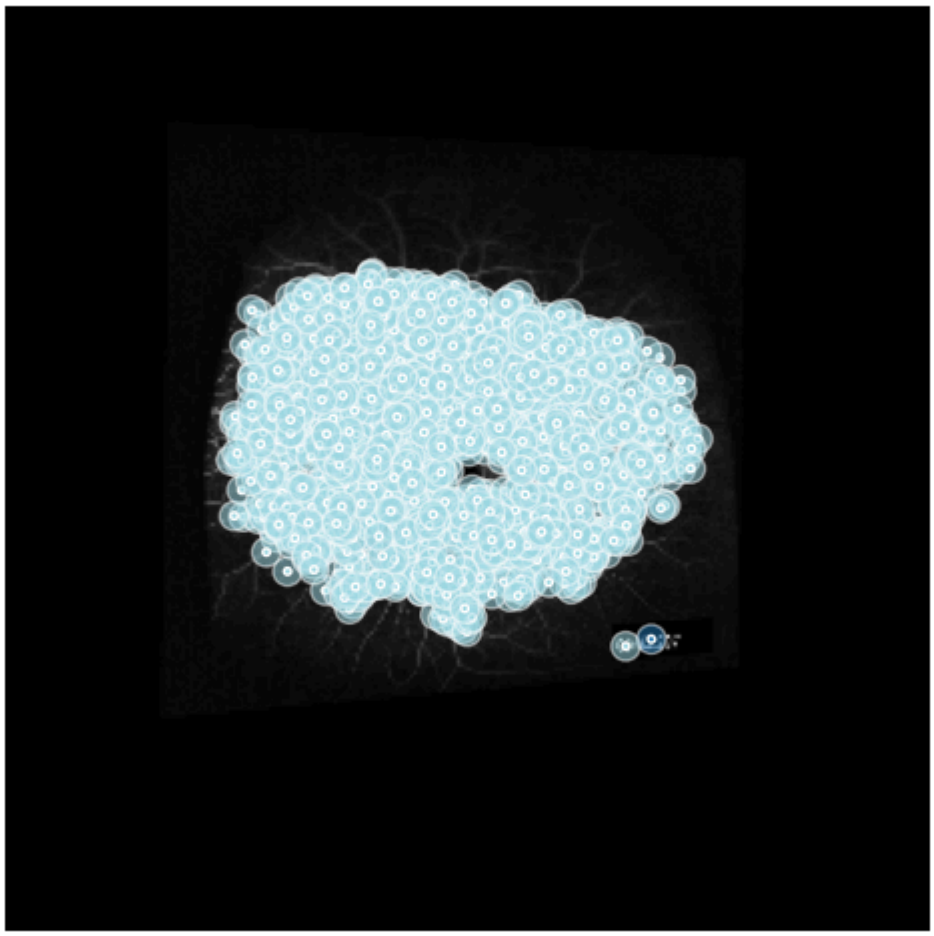


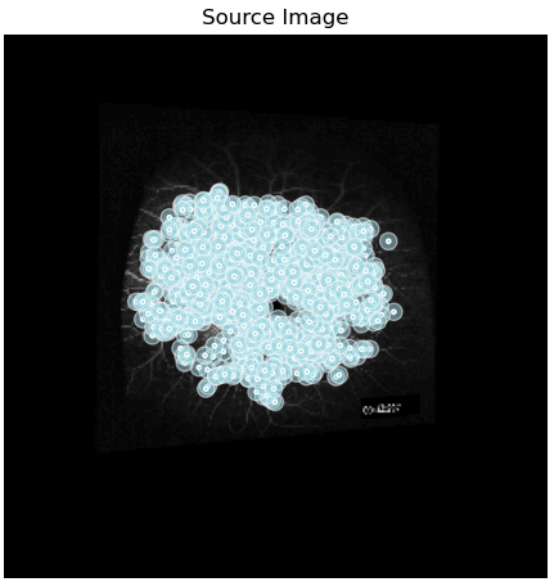
Deformed Image

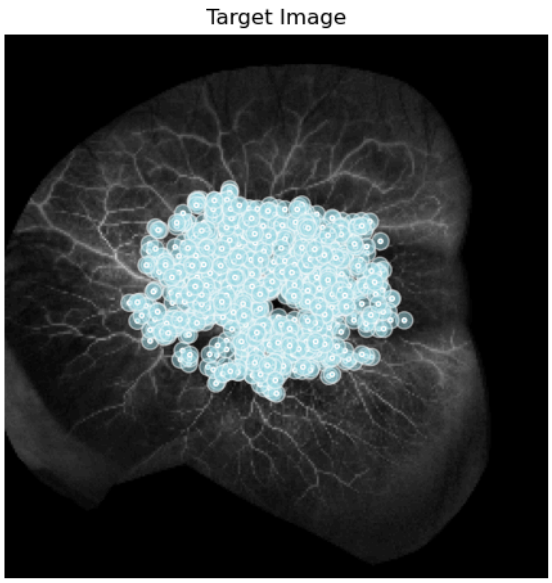


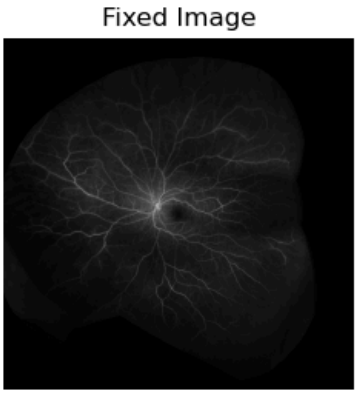
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
1019
```

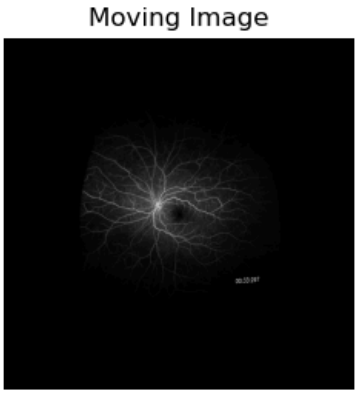
Chosen Keypoints

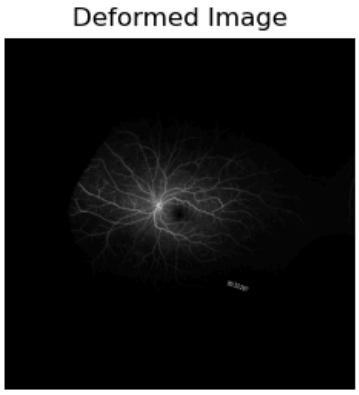


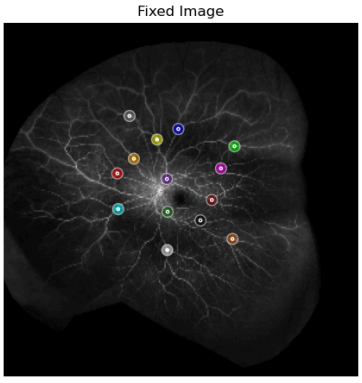


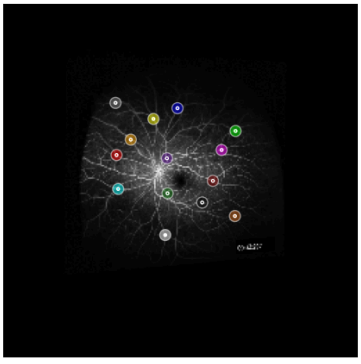


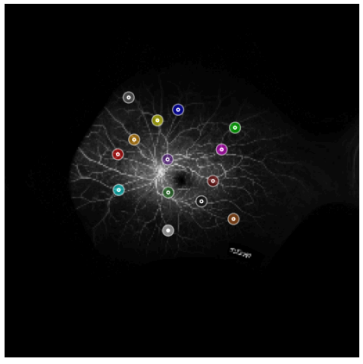






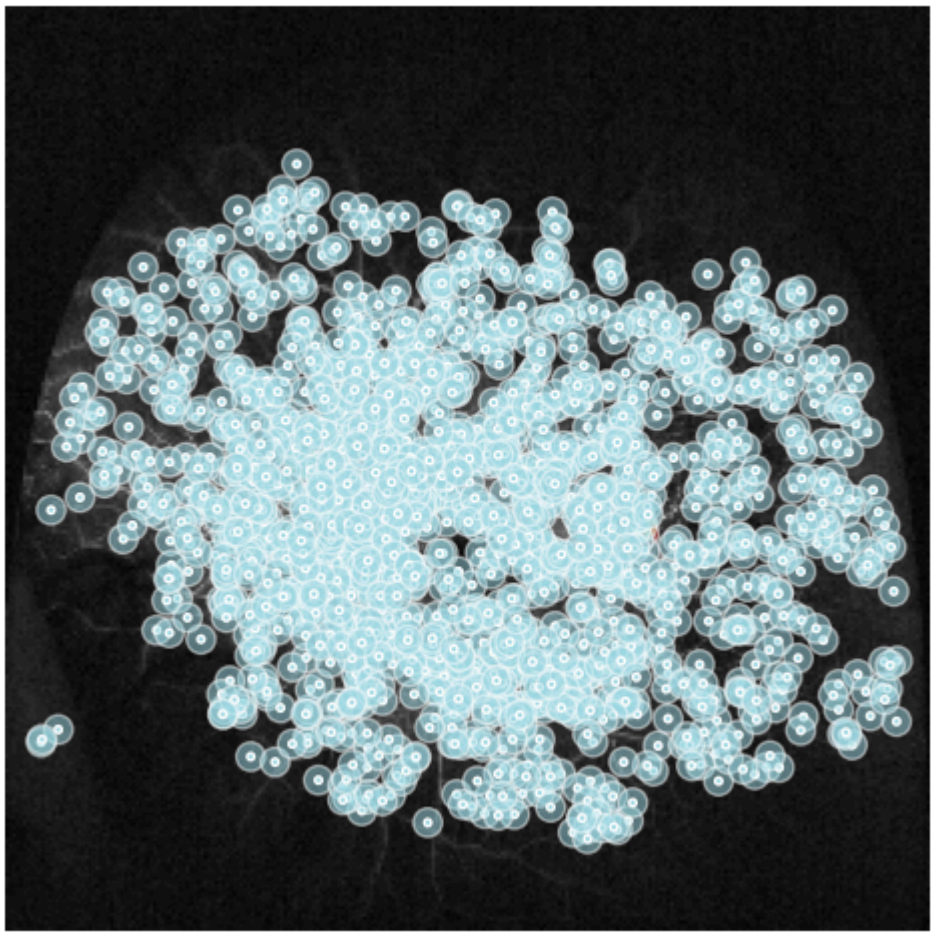




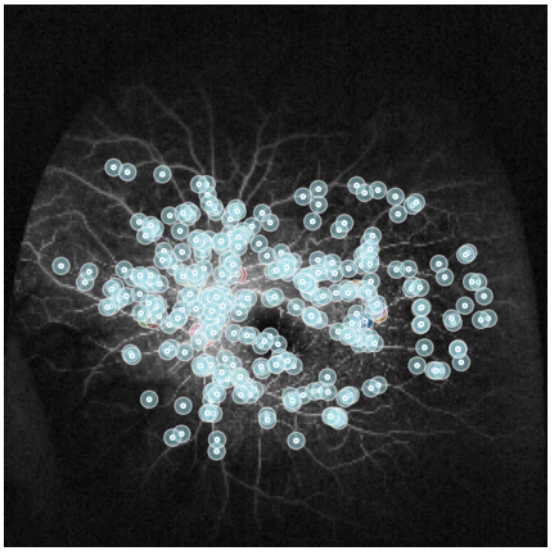



```
Recorded Landmark Error for Iteration 11 is 18.30264639174361
Iteration 12
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_1/Montage/Montage_Subject_1.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_1/FA/Raw_FA_3_Subject_1.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
337
```

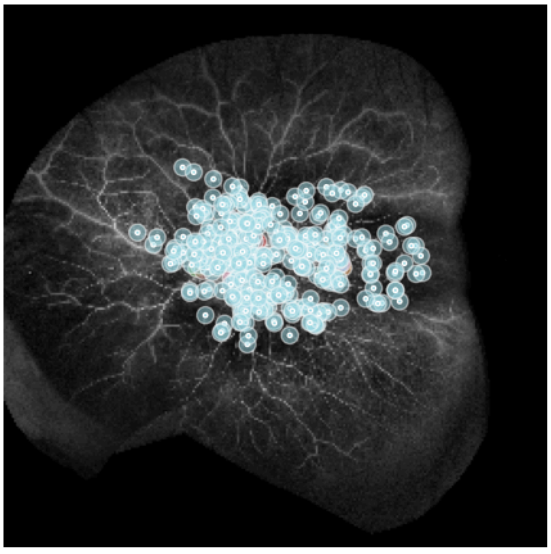
Chosen Keypoints



Source Image

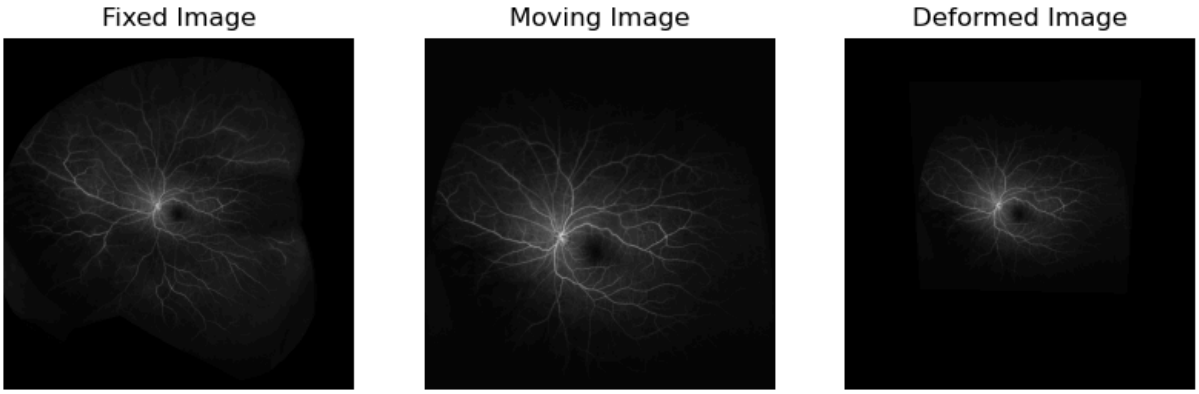


Target Image

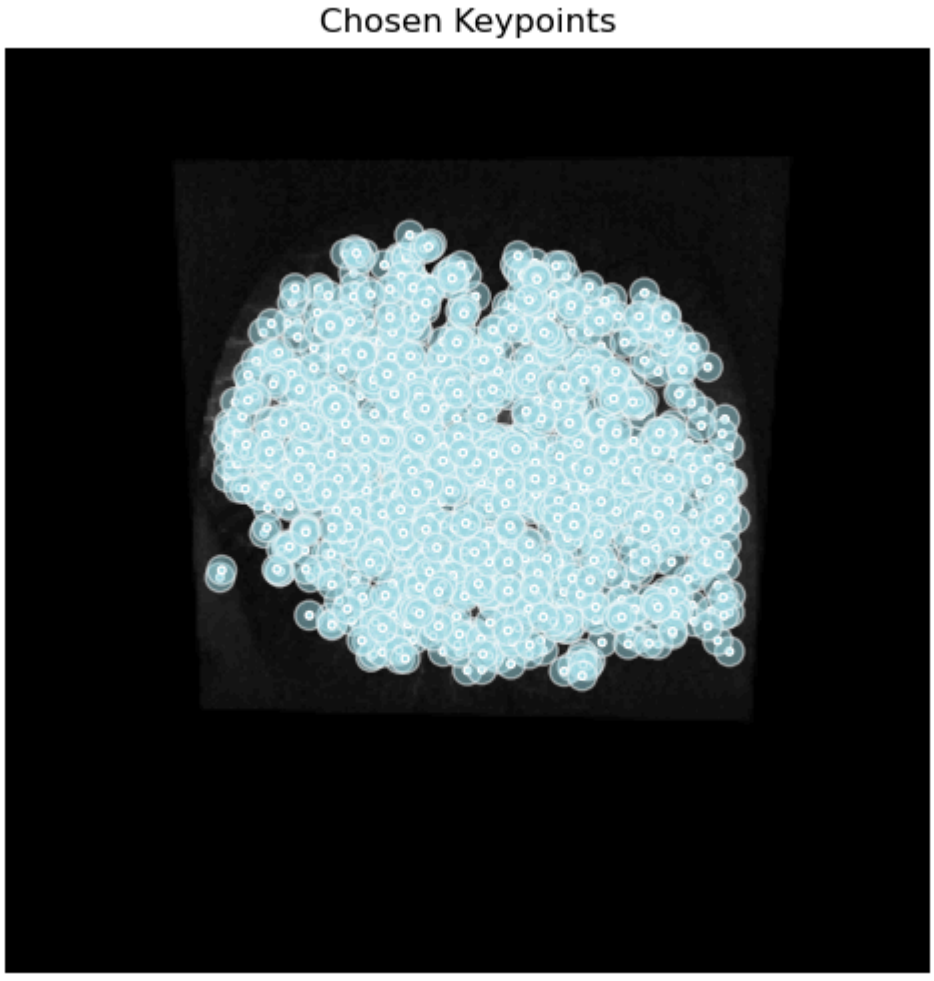


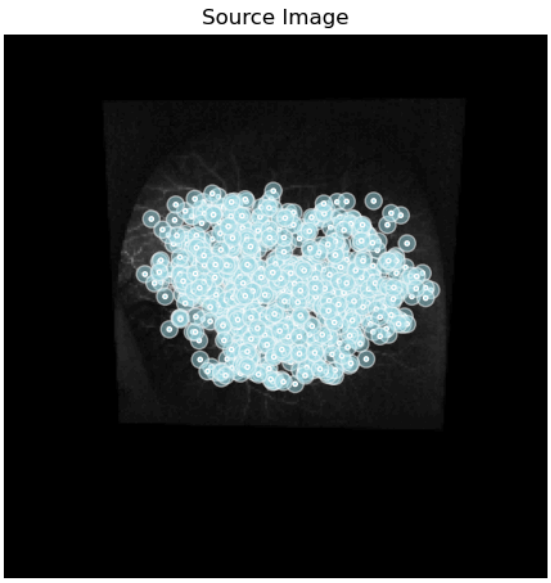
Homography Matrix:

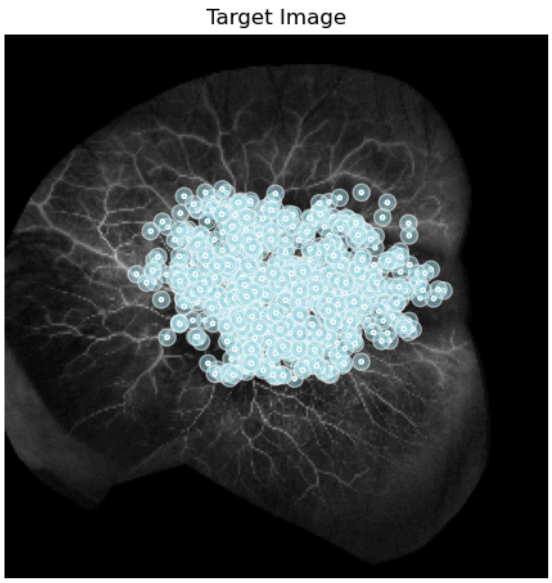
```
[[ 6.39416053e-01  5.71105029e-02  1.85722523e+02]
 [-9.25708648e-03  6.79310184e-01  1.25490572e+02]
 [-3.32240681e-05  1.20027981e-04  1.00000000e+00]]
```

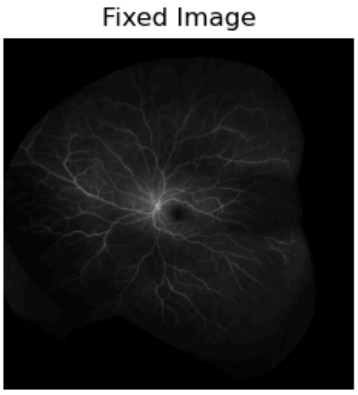


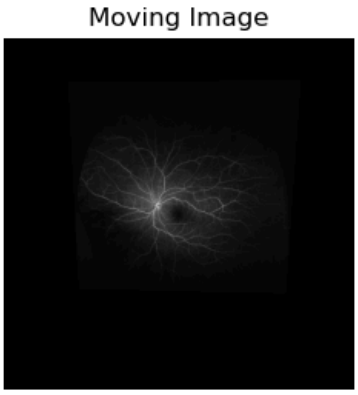

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
781
```

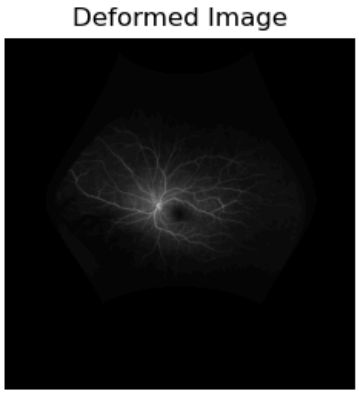


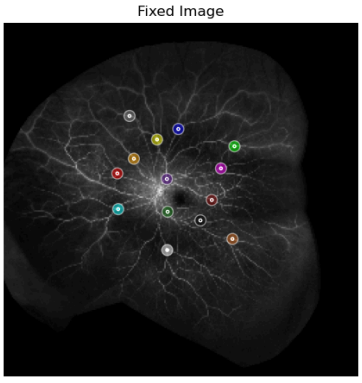


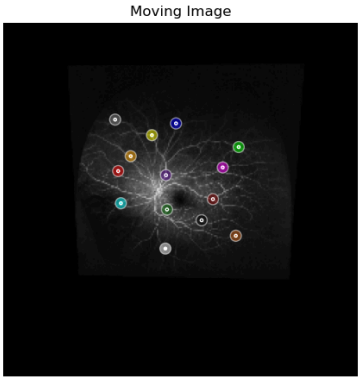


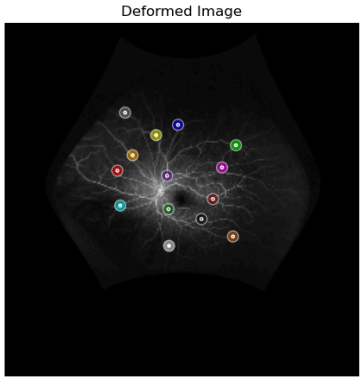






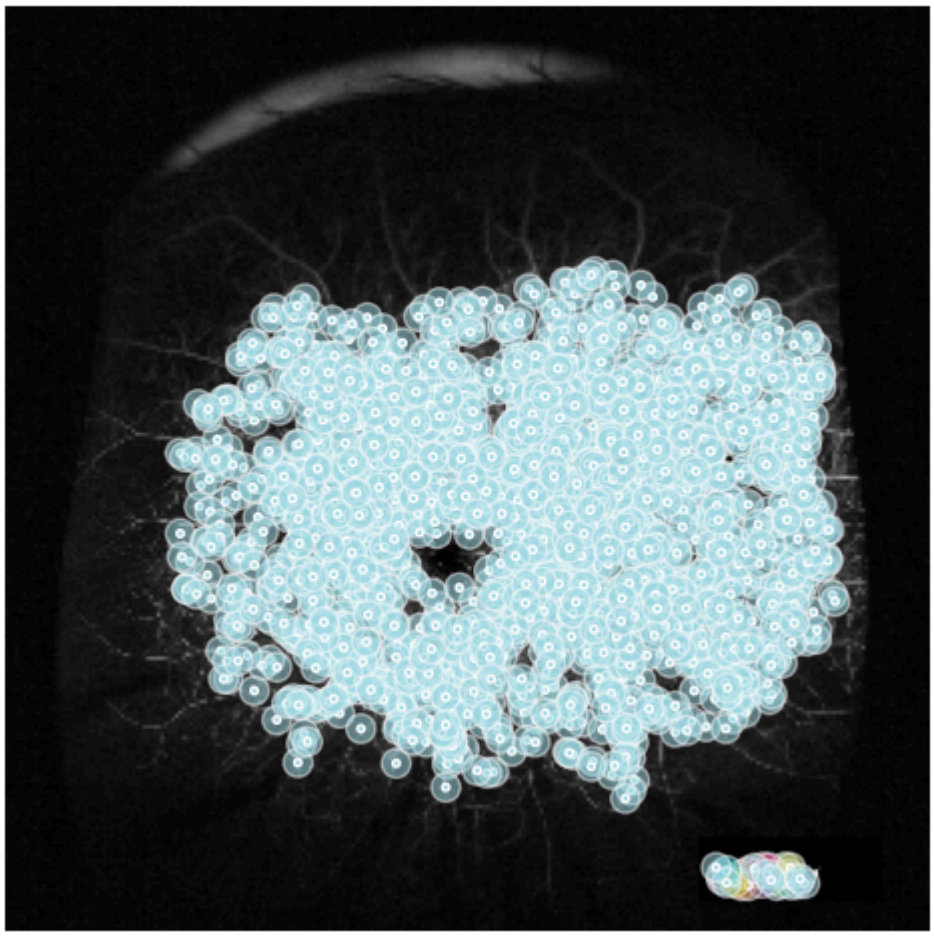




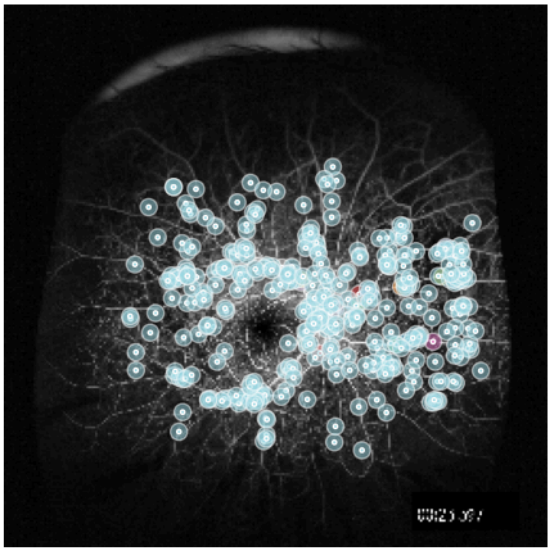



```
Recorded Landmark Error for Iteration 12 is 39.12664958814495
Iteration 13
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_5/FA/Raw_FA_1_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
350
```

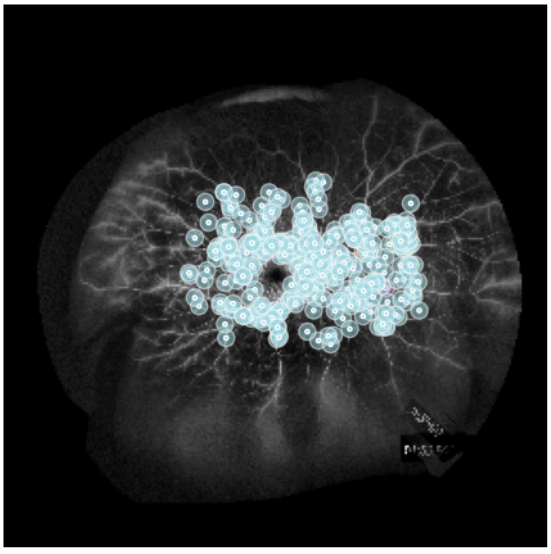
Chosen Keypoints



Source Image



Target Image



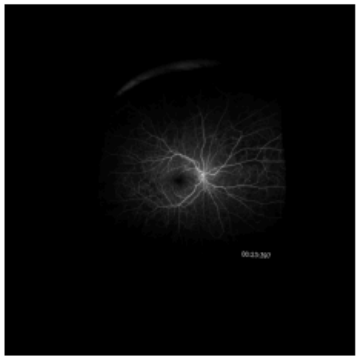
Homography Matrix:

```
[[ 4.92007827e-01  3.92596722e-02  2.23240880e+02]
 [-9.30849915e-02  6.02137103e-01  1.69675224e+02]
 [-1.84995925e-04  7.71913386e-05  1.00000000e+00]]
```

Fixed Image

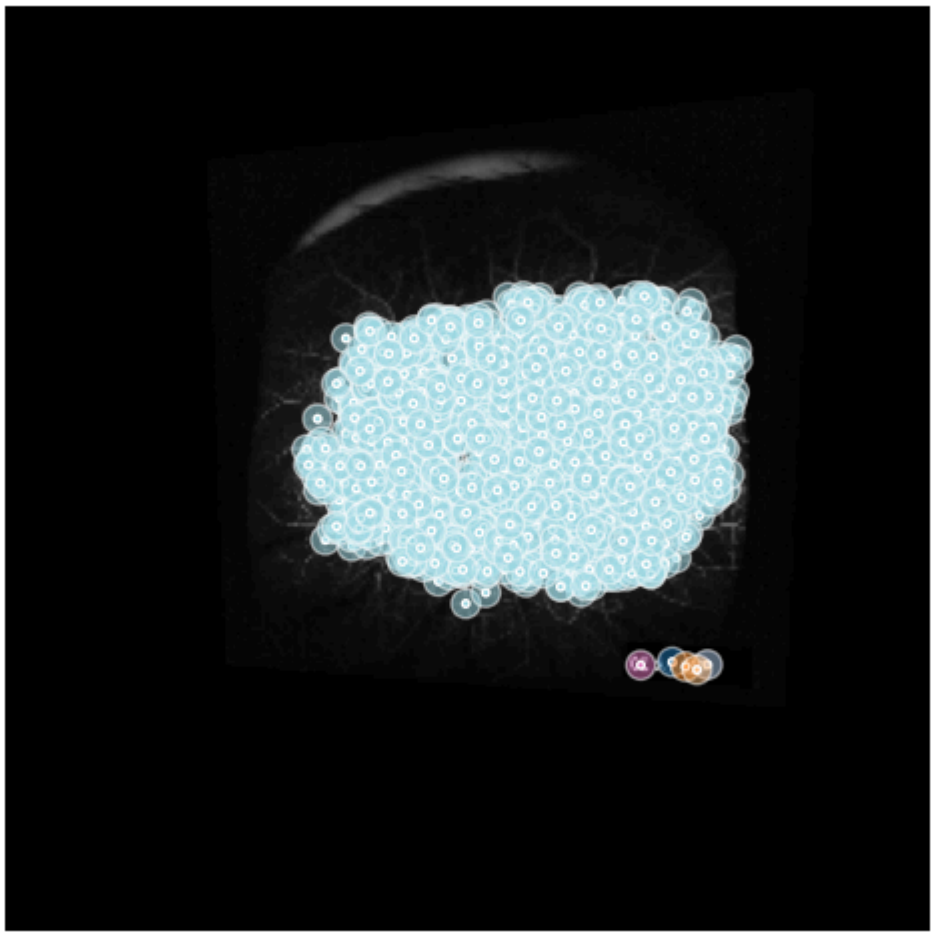
Moving Image

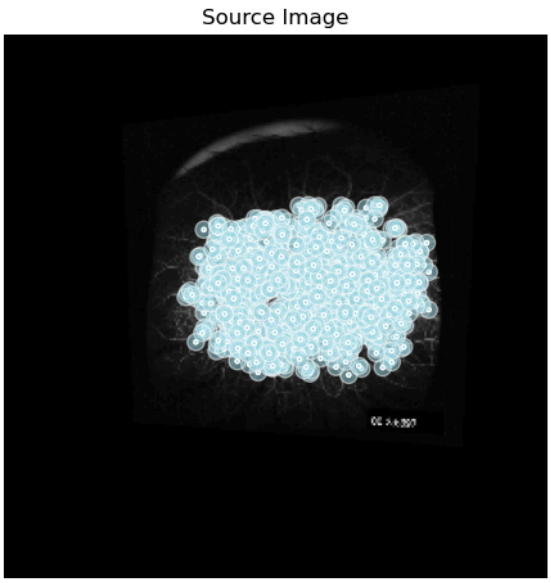
Deformed Image

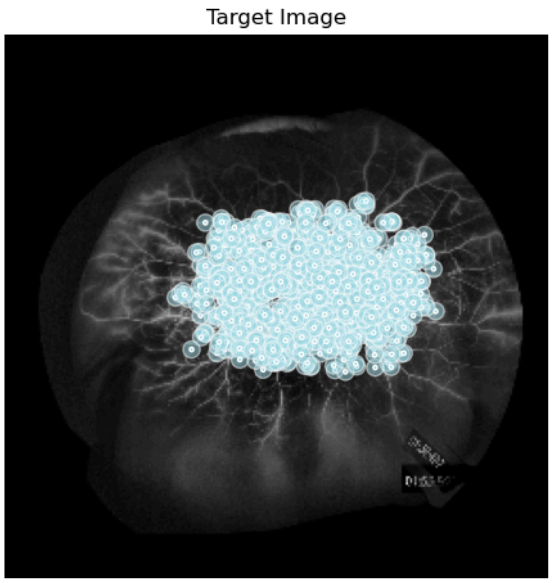


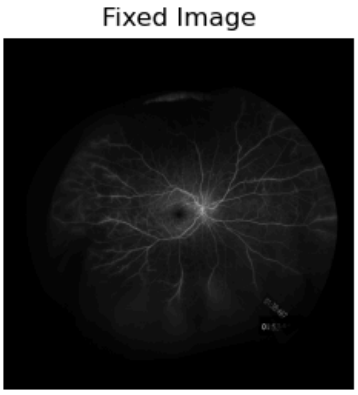
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
1046
```

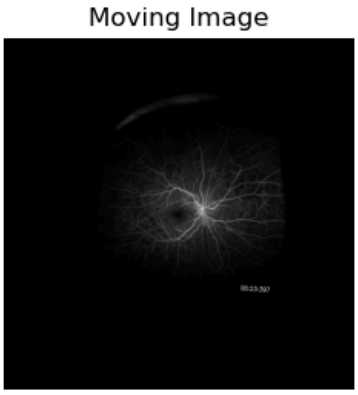
Chosen Keypoints

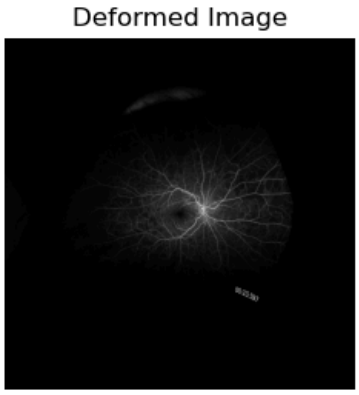


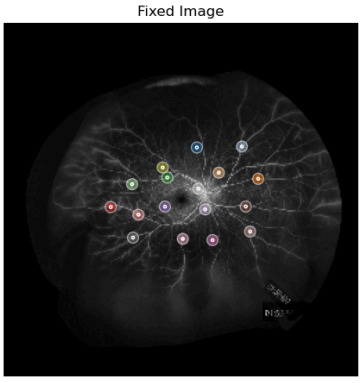


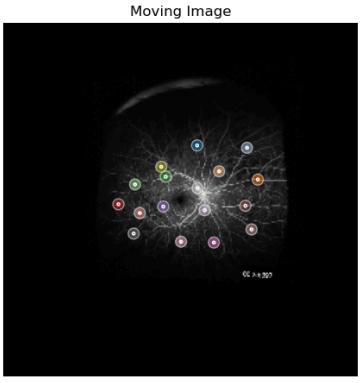


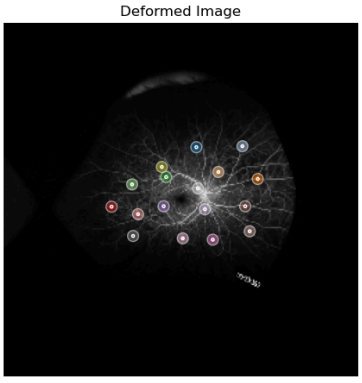






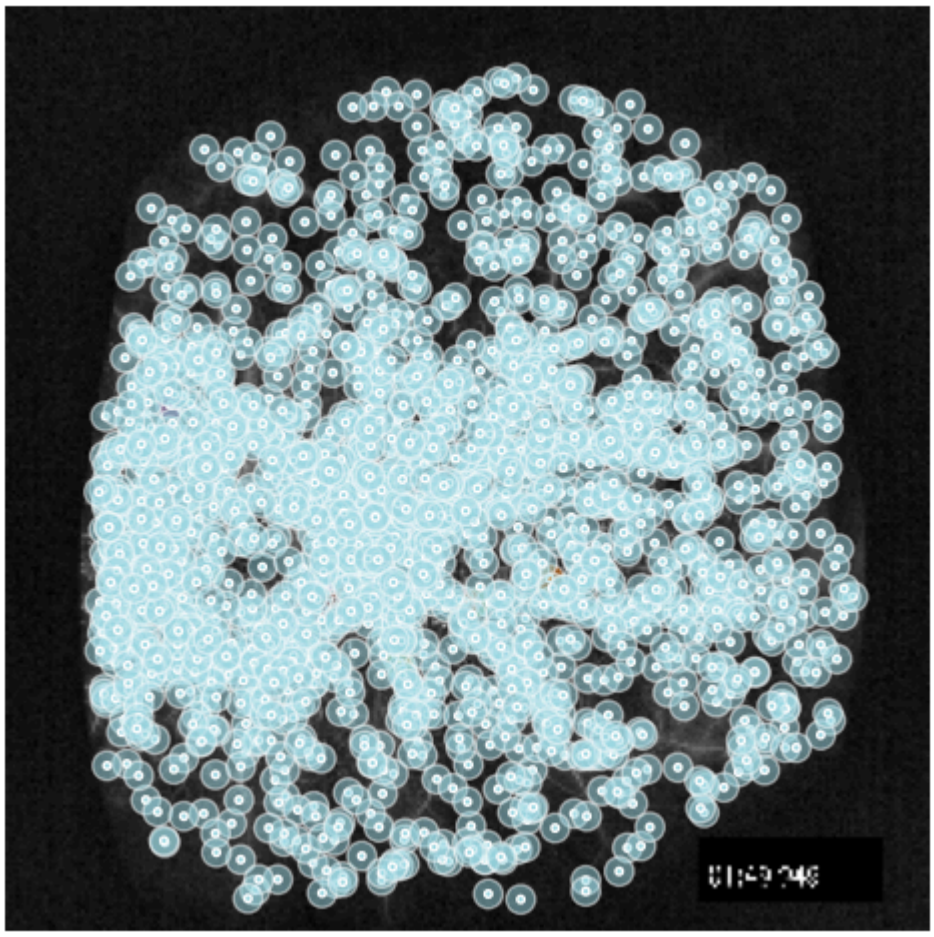




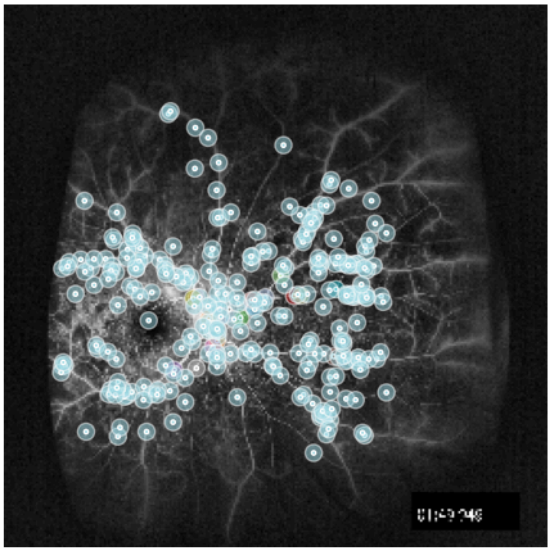



```
Recorded Landmark Error for Iteration 13 is 9.776384015721703
Iteration 14
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_5/FA/Raw_FA_2_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
269
```

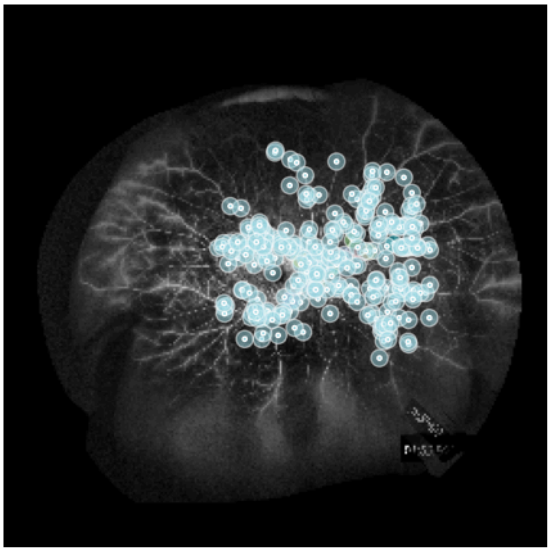
Chosen Keypoints



Source Image

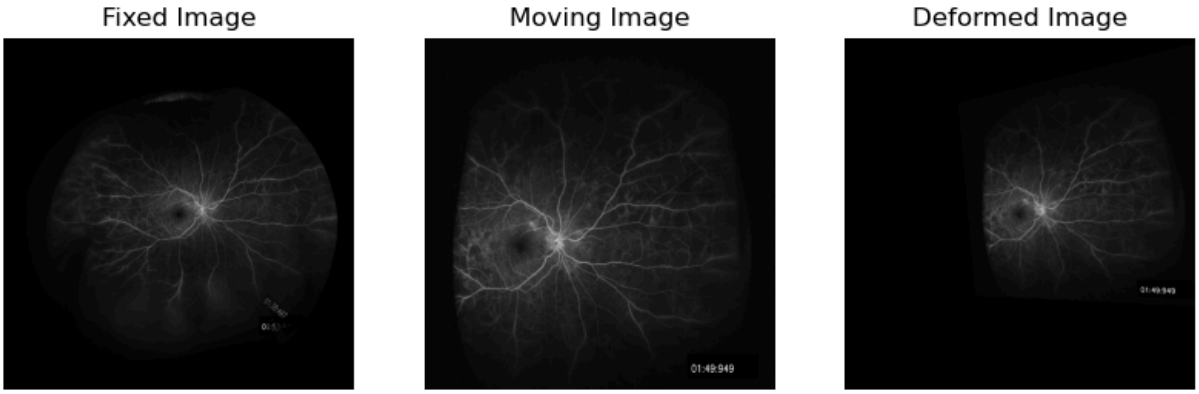


Target Image

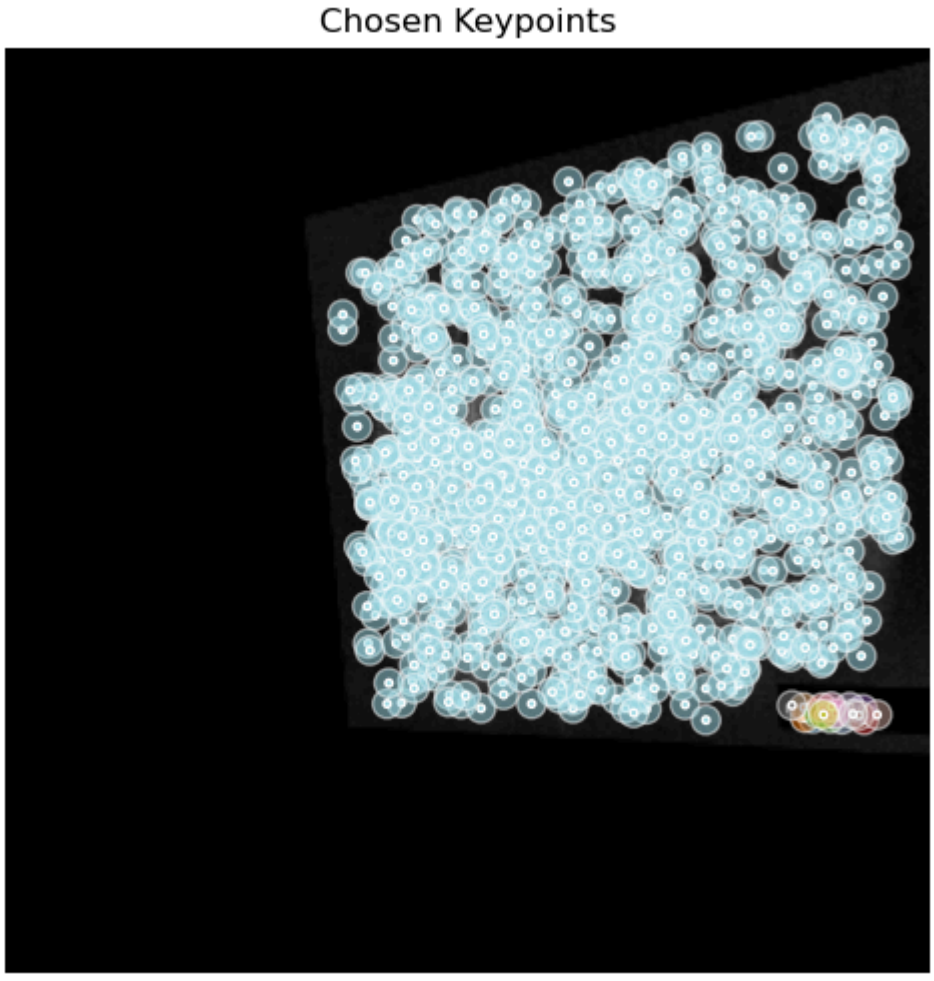


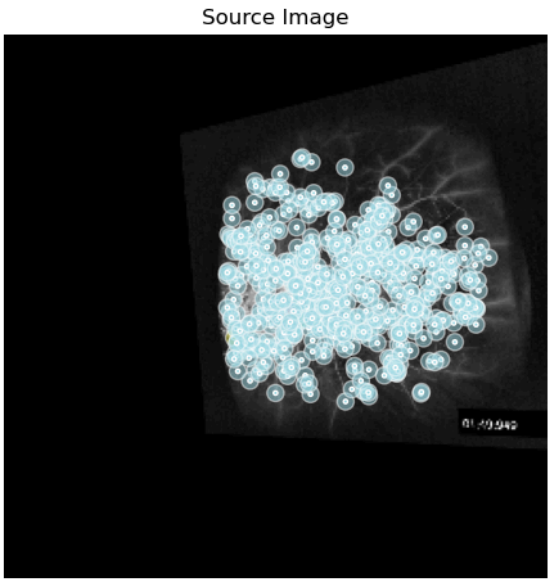
Homography Matrix:

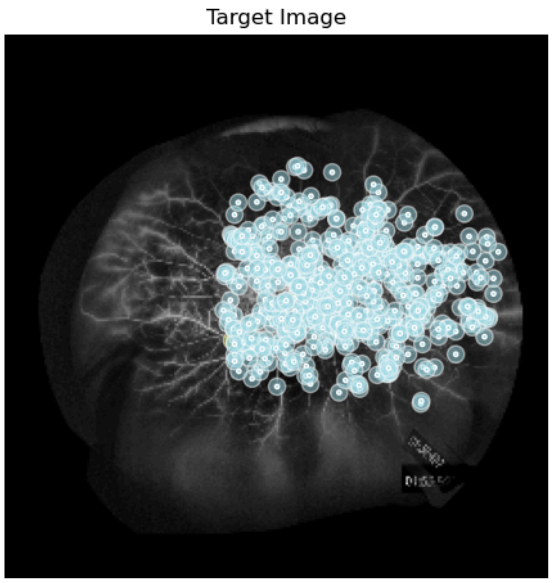
```
[[ 4.07267287e-01  5.50218055e-02  3.31102236e+02]
 [-1.73899897e-01  5.63392649e-01  1.87356513e+02]
 [-2.64131124e-04  1.71498764e-05  1.00000000e+00]]
```

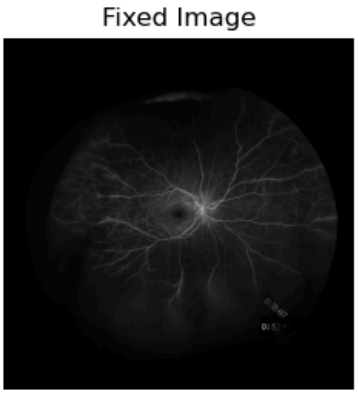


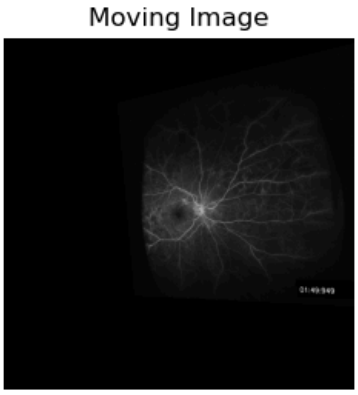

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
584
```

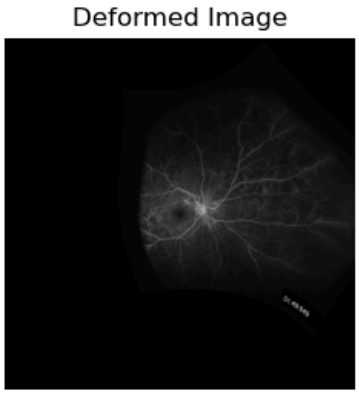


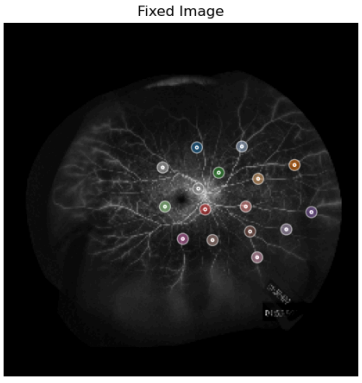


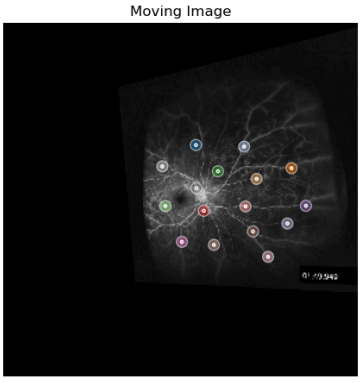


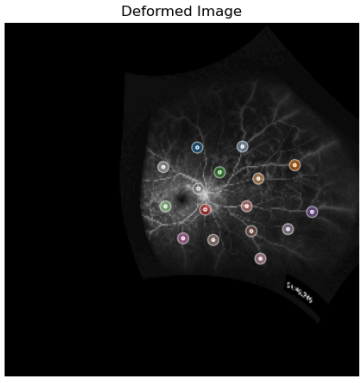






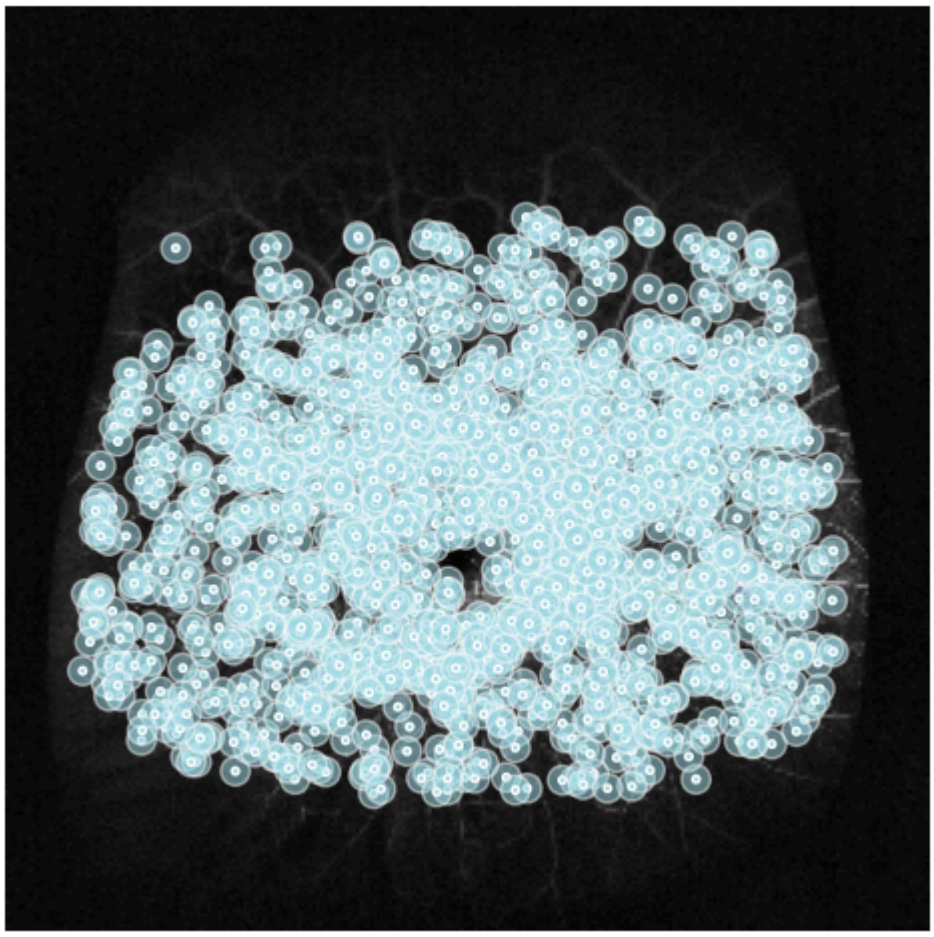




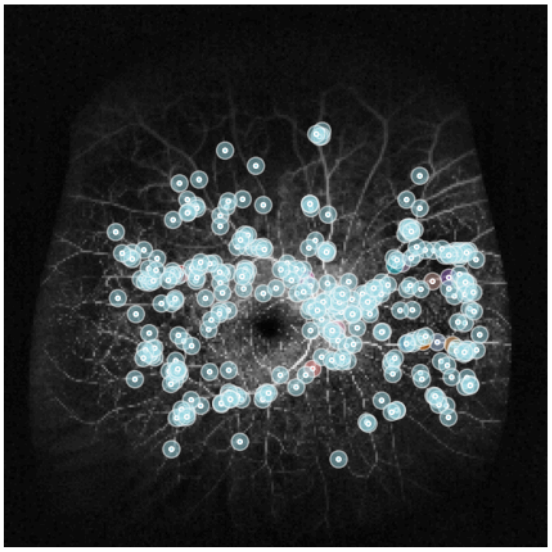



```
Recorded Landmark Error for Iteration 14 is 10.88296466691726
Iteration 15
Loading Source Images /blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_R
egistration/FLoRI21_DataPort/data/Subject_5/Montage/Montage_Subject_5.tif ,Target Im
age/blue/weishao/vi.sivaraman/Tasks/T17/2D Registration/FLoRI21_Registration/FLoRI21
_DataPort/data/Subject_5/FA/Raw_FA_3_Subject_5.tif to the framework
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
322
```

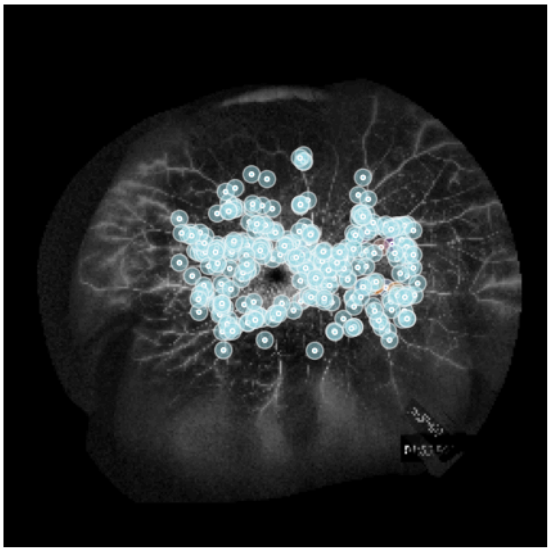
Chosen Keypoints



Source Image



Target Image



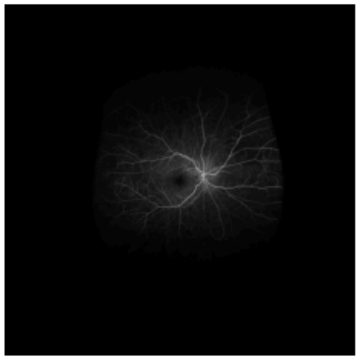
Homography Matrix:

```
[[ 4.99469425e-01  4.36042145e-02  2.12038293e+02]
 [-9.10940554e-02  6.02261021e-01  1.70667994e+02]
 [-1.64585875e-04  5.98574605e-05  1.00000000e+00]]
```

Fixed Image

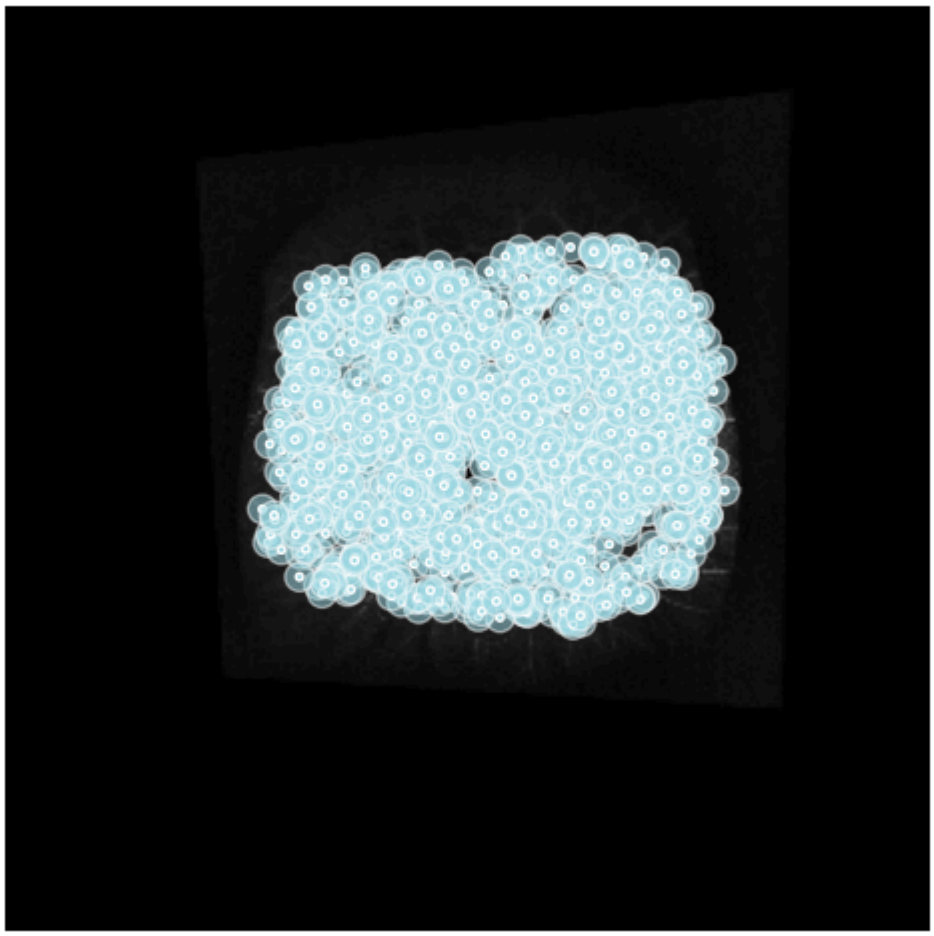
Moving Image

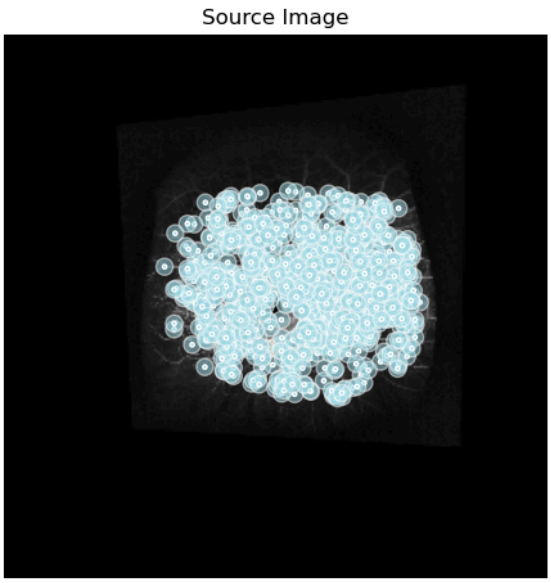
Deformed Image

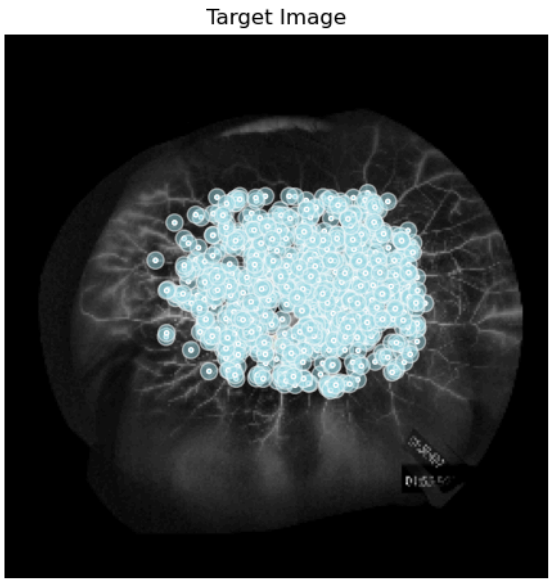


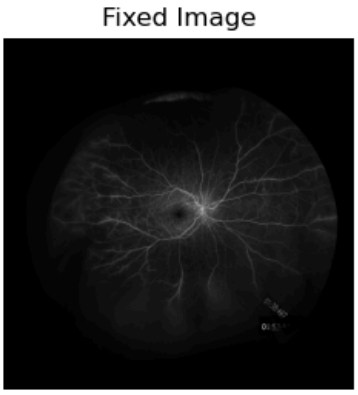
```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
835
```

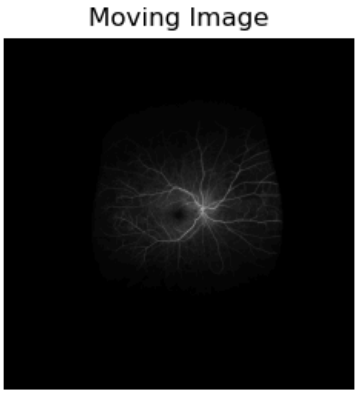
Chosen Keypoints

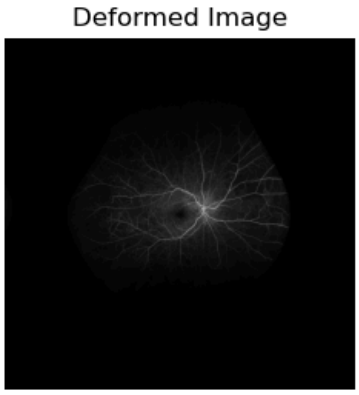


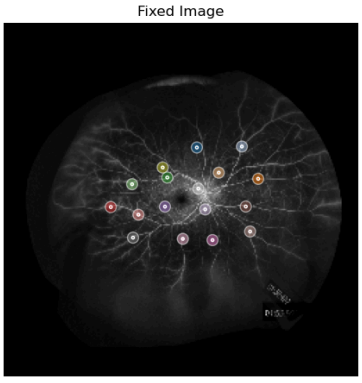


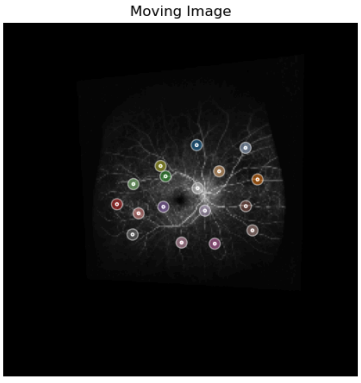


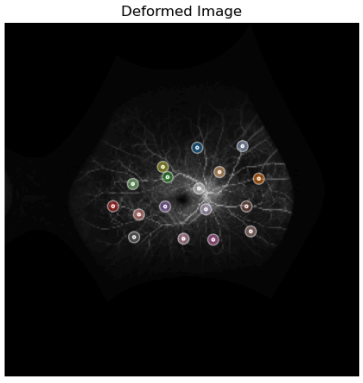






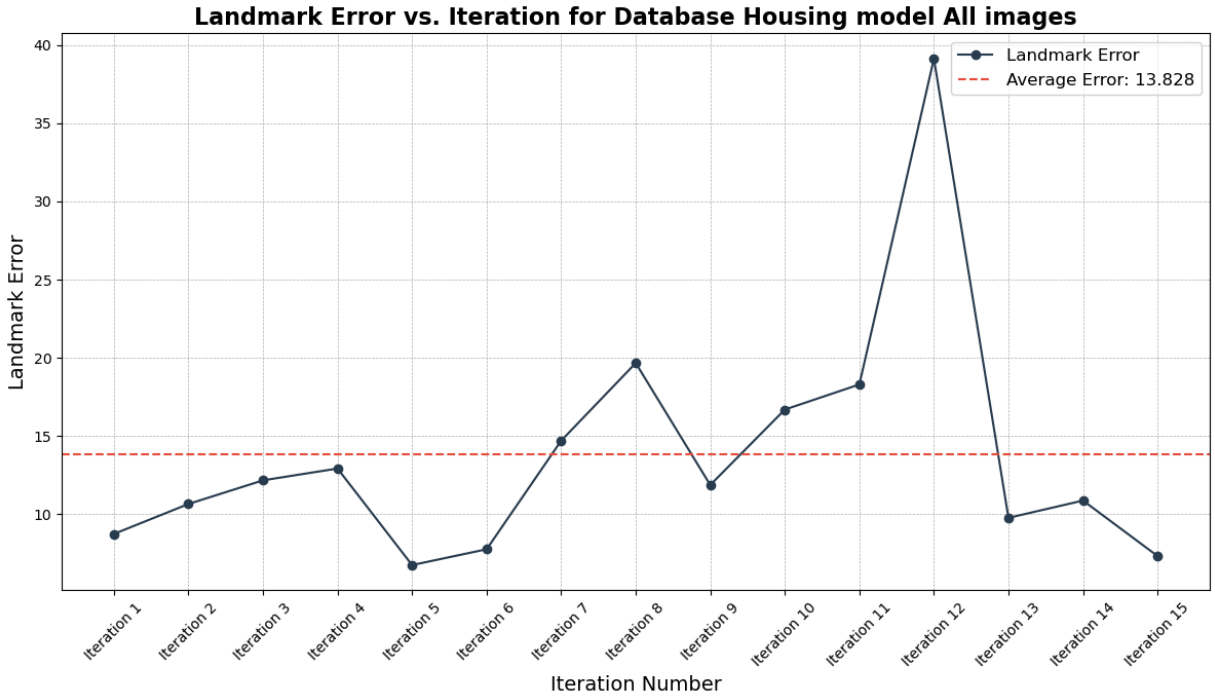




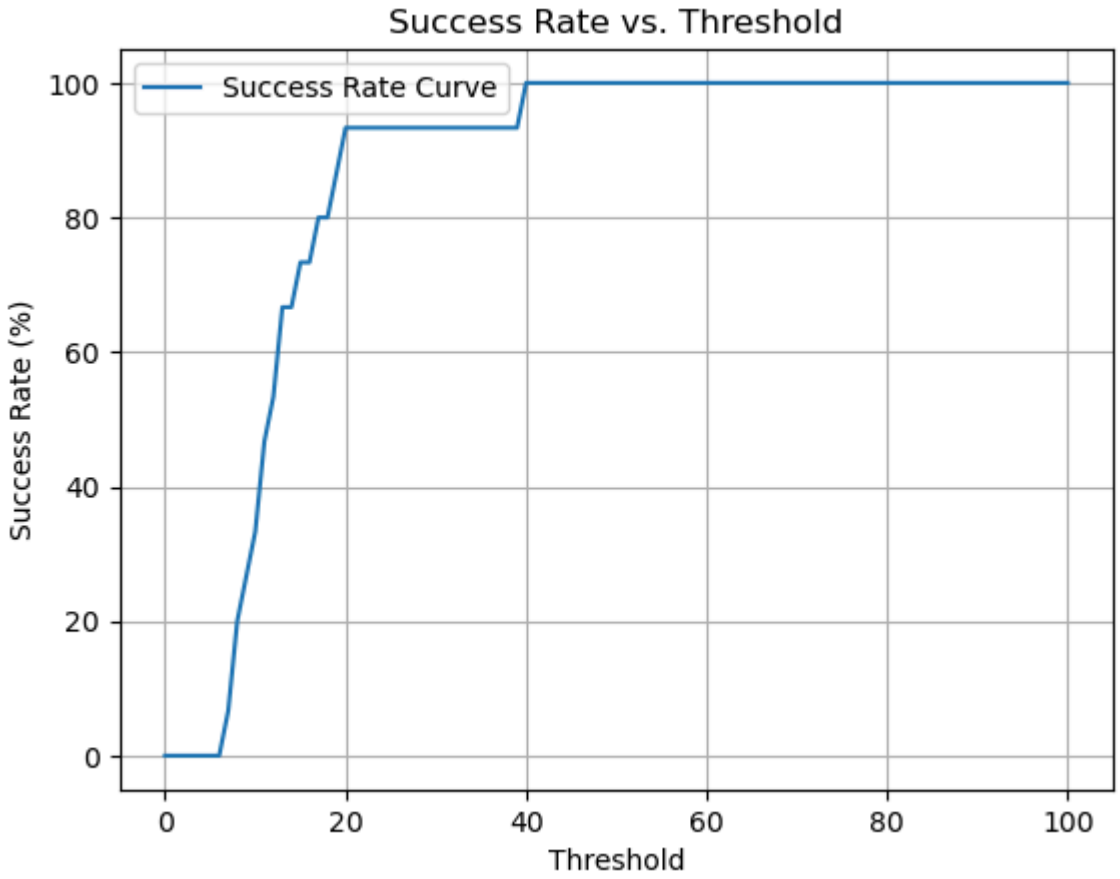



```
Recorded Landmark Error for Iteration 15 is 7.335814838372301
```

In [19]:
```
plot_landmark_errors(landmark_errors,os.path.join(os.getcwd(),'FLoRI21_DataPort_Ima
```



In [20]:
```
compute_plot_Flori21_AUC(landmark_errors)
```




```
AUC: 0.868
```