ECE 661 - Computer Vision
Homework 5

Arnav Singh

Contents

1 Theory Questions

1.1 Question 1

Conceptually speaking, how do we differentiate between the inliers and the outliers when using the RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

In each trial of the RANSAC algorithm, after we have randomly chosen n correspondences, we calculate the Linear Least-Squares (LLS) estimate of the homography H. Based on this homography H we find what is called the re-projection error for both the domain and range points. A correspondence is an inlier (outlier) if the re-projection error was less than or equal to (greater than) a certain user-defined threshold δ.

Let the i^{th} correspondence be $(\boldsymbol{x}_i, \boldsymbol{x'}_i)$ and H be the LLS estimate of this particular trial.
The estimate of the range point would be

$$\hat{\boldsymbol{x'}}_i = H\boldsymbol{x}_i.$$

And the corresponding estimate of the domain point would be

$$\hat{\boldsymbol{x}}_i = H^{-1}\boldsymbol{x'}_i.$$

The overall re-projection or geometric error associated with this correspondence is (after converting them from HC to physical coordinates)

$$d_i = \frac{1}{2}\sqrt{||\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i||^2 + ||\boldsymbol{x'}_i - \hat{\boldsymbol{x'}}_i||^2}.$$

The factor of 1/2 is a matter of convention, since we are considering errors on both domain and range planes.

The threshold δ is calculated based on the expected deviation of the locations of the keypoints in the scene. If we consider the accuracy of finding the exact location of the keypoint at (x, y) has a Gaussian behaviour, i.e., maximum at (x, y) and falls away with standard deviation σ at the point $(x + \Delta x, y + \Delta y)$, we are interested in the distribution of $d^2 = (\Delta x)^2 + (\Delta y)^2$. We find that it has a χ^2 distribution with 2 d.o.f. The threshold δ is chosen so that 90% of the inliers are accepted. Based on c.d.f. tables of χ^2 distribution, we arrive at $\delta = 3\sigma$.

1.2 Question 2

The Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method but it can be numerically unstable. Explain in your words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and is numerically stable at the same time.

The Levenberg-Marquardt (LM) algorithm uses a damping coefficient μ which dictates if we take a Gradient Descent (GD) step or a Gauss-Newton (GN) step. At the k^{th} iteration, the "update" term $\delta_{\boldsymbol{p}}$ is the solution of the following linear equation:

$$(J_{\boldsymbol{f}}^\top J_{\boldsymbol{f}} + \mu \boldsymbol{I})\boldsymbol{\delta_p} = J_{\boldsymbol{f}}^\top \boldsymbol{\epsilon}(\boldsymbol{p_k})$$

where, $J_{\boldsymbol{f}}$ is the Jacobian of the vector predictor function $\boldsymbol{f}(\boldsymbol{p_k})$, $\boldsymbol{I}$ is the Identity matrix, and $\boldsymbol{\epsilon}(\boldsymbol{p_k})$ is the prediction residual at the k^{th} iteration.
If the value of μ is much larger compared to the diagonal of $J_{\boldsymbol{f}}^\top J_{\boldsymbol{f}}$, the solution of the "update" term is closer to GD. Similarly, if $\mu = 0$, the we get the GN solution.
We know that the GD algorithm is numerically stable but is slow to convergence because the gradient approaches 0 near the minimum of the cost function. The "update" term becomes smaller and smaller. On the the other hand, GN can reach the solution within minimal number of iterations, but is highly unstable because in most cases, the Jacobian matrices are not full rank and it gets difficult to find its pseudo-inverse.
In the LM algorithm, we start with a high value of μ to exploit the numerical stability of the GD algorithm. At each iteration, we perform a check to see if we are close enough to take the GN jump to the minima. The value of μ in each iteration is updated based on the ratio of the actual change in the cost function to the predicted value of the change in the cost function based on the current choice of μ.

2 Panoramic Stitches or Image Mosaics

2.1 Overall Pipeline

The overall pipeline to create a panorama of a given set of images is -

1. For every consecutive pairs of images, automatically estimate the homography by -
   (a) Using an interest point detector and matcher like SIFT or SuperPoint+SuperGlue to find correspondences between the two images.
   (b) Using an outlier ejection algorithm like RANSAC to differentiate between the noisy (inliers) and false (outliers) correspondences.
   (c) Using Linear Least-Squares minimization to find the homography based on the inliers that have the best inlier support.
   (d) Using Nonlinear Least-Squares minimization to refine the homography.
2. Calculate the homography w.r.t to the reference image (will be placed at the center of the canvas).
3. From left to right, apply the homographies to the image and place it on the canvas.

2.2 Mathematical Background and Python Implementation

2.2.1 Linear Least-Squares Minimization (using SVD)

Given a set of K point-to-point correspondences $(\boldsymbol{x}, \boldsymbol{x'})$, we can find the Linear Least-Squares (LLS) estimate of the homography H using two methods. One of those methods involves using a homogeneous set of equations followed by a Singular Value Decomposition (SVD).
For a correspondence, using homogeneous coordinates representation, we can say that the rows of H denoted by $(\boldsymbol{h^1}, \boldsymbol{h^2}, \boldsymbol{h^3})$ satisfy

$$\begin{aligned} \mathbf{0}^\top \boldsymbol{h^1} - w' \boldsymbol{x}^\top \boldsymbol{h^2} + y' \boldsymbol{x}^\top \boldsymbol{h^3} = 0 \\ w' \boldsymbol{x}^\top \boldsymbol{h^1} \quad + \mathbf{0}^\top \boldsymbol{h^2} - x' \boldsymbol{x}^\top \boldsymbol{h^3} = 0 \end{aligned}$$

Collecting all the equations using the K correspondences gives us

$$\boldsymbol{A}\boldsymbol{h} = \mathbf{0}$$

where,

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & -w'_1 x_1 & -w'_1 y_1 & -w'_1 w_1 & y'_1 x_1 & y'_1 y_1 & y'_1 w_1 \\ w'_1 x_1 & w'_1 y_1 & w'_1 w_1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 & -x'_1 w_1 \\ & & & & \vdots & & & & \\ 0 & 0 & 0 & -w'_K x_K & -w'_K y_K & -w'_K w_K & y'_K x_K & y'_K y_K & y'_K w_K \\ w'_K x_K & w'_K y_K & w'_K w_K & 0 & 0 & 0 & -x'_K x_K & -x'_K y_K & -x'_K w_K \end{bmatrix}$$

$$\boldsymbol{h} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{bmatrix}^\top$$

The matrix $\boldsymbol{A}$ is of dimension $2K \times 9$ and therefore, we have $2K$ equations. Here, $2K$ far exceeds the number of variables (9) and thus is an over-determined system of equations. Often, this system of equations does not have a solution.
To obtain a solution is this very scenario, we apply an constrained minimization of $||\boldsymbol{A}\boldsymbol{h}||$, subject to the constraint $||\boldsymbol{h}|| = 1$.
The way to solve this constrained minimization problem is by using SVD. The value of $\boldsymbol{h}$ that satisfies this constraint is the vector associated with smallest singular value.

```python
def H_p_to_p(pts1:np.ndarray, pts2:np.ndarray):

    """
    Function to estimate Linear Least Squares fit of the homography matrix H such that pts2 =
    H * pts1.

    Args:
        pts1 (np.ndarray): Set of points on domain plane. Must be in homogeneous coordinates.
        pts2 (np.ndarray): Set of matching points on range plane. Must be in homogeneous
    coordinates.

    Returns:
        np.ndarray: Homography matrix such that pts2 = H * pts1
    """

    assert len(pts1) == len(pts2)

    assert len(pts1) >= 4

    A = []

    for (x, y, w), (xp, yp, wp) in zip(pts1, pts2):
        A.append([0, 0, 0, -wp*x, -wp*y, -wp*w, yp*x, yp*y, yp*w])
        A.append([wp*x, wp*y, wp*w, 0, 0, 0, -xp*x, -xp*y, -xp*w])

    A = np.array(A)
    _, _, vt = np.linalg.svd(A)

    h = vt.T[:, -1]

    H = np.reshape(h, (3, 3))

    H /= H[2, 2] + eps

    return H
```

Listing 1: Python implementation to find the LLS estimate of the homography H.

2.2.2 RANdom SAmpling Consensus (RANSAC)

Once we obtain a set of correspondences, we must be able to remove any false correspondences or outliers to get a correct estimate of the homography H. A single outlier can give incorrect results of H. One method to remove the outliers is using the RANSAC algorithm, it returns a set of inliers that has the maximum inlier support.

In a trial, we consider a set of n (typically $4 < n < 10$) correspondences to calculate the LLS estimate of the homography. Then we calculate the re-projection error both in the domain points and range points and threshold it by $\delta = 3\sigma$. Correspondences that have re-projection error less than or equal to the threshold are considered as the inliers and once with re-projection error greater than the threshold are considered as outliers. This was further discussed in the answer of the theory question. We conduct N trials that is determined using an adaptive algorithm based on the outlier probability (ϵ) and based on the probability that at least one of those trials only contain inliers (p).

Algorithm 1 Adaptive algorithm to determine the number of RANSAC samples

$N \leftarrow \infty$
$sample\_count \leftarrow 0$
$e \leftarrow 1$
while $(N > sample\_count)$ **do**
  $sample\_count \leftarrow sample\_count + 1$
  Choose a sample of n correspondences and count the number of inliers
  $\epsilon \leftarrow 1 - \dfrac{n\_inliers}{n\_total}$
  $N \leftarrow \dfrac{\log(1-p)}{\log(1-(1-\epsilon)^n)}$
end while

Algorithm 2 RANSAC(domain points $\boldsymbol{x}$. range points $\boldsymbol{x'}$, σ, p, n)

$n\_total \leftarrow (len(pts1))$
$\delta \leftarrow 3\sigma$
$\epsilon \leftarrow 1$
$N \leftarrow \infty$
$sample\_count \leftarrow 0$
$inliers,\ outliers,\ len\_inliers \leftarrow [\ ]$
while $N > sample\_count$ **do**
  $sample\_count \leftarrow sample\_count + 1$
  randomly select n correspondences
  $H \leftarrow$ LLS estimate of the homography H ▷ Using SVD discussed previously
  $sample\_inliers,\ sample\_outliers \leftarrow [\ ]$
  $len\_sample\_inliers \leftarrow 0$
  for (i^{th} correspondence $(\boldsymbol{x_i}, \boldsymbol{x'_i})$) **do**
    $\hat{\boldsymbol{x}'}_i \leftarrow H\boldsymbol{x}_i$
    $\hat{\boldsymbol{x}}_i \leftarrow H^{-1}\boldsymbol{x}'_i$
    $d_i \leftarrow \frac{1}{2}\sqrt{||\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i||_2^2 + ||\boldsymbol{x}'_i - \hat{\boldsymbol{x}'}_i||_2^2}$ ▷ Re-projection error
    if $d_i \leq \delta$ **then**
      add $(\boldsymbol{x_i}, \boldsymbol{x'_i})$ to $sample\_inliers$
      $len\_sample\_inliers \leftarrow len\_sample\_inliers + 1$
    else
      add $(\boldsymbol{x_i}, \boldsymbol{x'_i})$ to $sample\_outliers$
    end if
  end for
  $\epsilon \leftarrow 1 - \frac{len\_sample\_inliers}{n\_total}$
  $N \leftarrow \frac{\log(1-p)}{\log(1-(1-\epsilon)^n)}$
  add $sample\_inliers$ to $inliers$
  add $sample\_outliers$ to $outliers$
  add $len\_sample\_inliers$ to $len\_inliers$
end while
$largest\_set \leftarrow \arg\max(len\_inliers)$
$best\_inliers \leftarrow inliers[largest\_set]$
$best\_ouliers \leftarrow outliers[largest\_set]$
return $best\_inliers,\ best\_outliers$

```python
def RANSAC(pts1:np.ndarray, pts2:np.ndarray, sigma:float, p:float, n:int, e:float=None):

    """
    Function to apply the RANSAC algorithm to find inliers and outliers of matching set of points pts1 and pts2.

    Args:
        pts1 (np.ndarray): Set of points on domain plane. Must be in regular coordinates.
        pts2 (np.ndarray): Set of matching points on range plane. Must be in regular coordinates.
        sigma (float): Estimate of noise induced on the noisy matches.
        p (float): Probability that atleast 1 trial will be free of outliers. ``0 <= p <= 1``
        n (int): Number of correspondences used to find the LLS estimate of homography H. ``4 < n < 10``
        e (float, optional): Probability that chosen correspondence is an outlier. Defaults to ``None`` as it is adaptively calculated. ``0 <= e <= 1``

    Returns:
        (np.ndarray, np.ndarray): Tuple consisting of the inliers and outliers with the maximum inlier support. Each row is a correspondence in the order (domain.x, domain.y, range.x, range.y)
    """

    assert len(pts1) == len(pts2)

    n_total = len(pts1)

```

```
    if (p < 0) or (p > 1):
        raise ValueError(f"Invalid value of p={p}. ‘p‘ must be within [0, 1].")

    if (n < 4) or (n > 10):
        raise ValueError(f"Invalid value of n={n}. ‘n‘ must be within (4, 10).")

    if e is None:
        e = 1
    else:
        if (e < 0) or (e > 1):
            raise ValueError(f"Invalid value of e={e}. ‘e‘ must be within [0, 1].")

    correspondences = np.hstack((pts1, pts2))

    delta = 3 * sigma

    N = np.inf
    sample_count = 0

    M = np.floor((1 - e) * n_total).astype(np.int32)

    pts1_hc = np.hstack((pts1, np.ones((n_total, 1))))
    pts2_hc = np.hstack((pts2, np.ones((n_total, 1))))

    outliers = []
    inliers = []
    len_inliers = []

    while (N > sample_count):
        sample_count += 1
        sample_idx = np.random.permutation(np.arange(n_total))[:n]

        pts1_selected = pts1_hc[sample_idx]
        pts2_selected = pts2_hc[sample_idx]

        H = H_p_to_p(pts1_selected, pts2_selected)

        pts1_estimate = np.linalg.inv(H) @ pts2_hc.T
        pts1_estimate /= pts1_estimate[2] + eps
        pts1_estimate = pts1_estimate.T
        dist1 = np.sum(np.power(pts1_estimate[:, :2] - pts1, 2), 1)

        pts2_estimate = H @ pts1_hc.T
        pts2_estimate /= pts2_estimate[2] + eps
        pts2_estimate = pts2_estimate.T
        dist2 = np.sum(np.power(pts2_estimate[:, :2] - pts2, 2), 1)

        dists = np.sqrt(dist1 + dist2) / 2

        inlier_idx = np.where(dists <= delta)[0]
        outlier_idx = np.where(dists > delta)[0]

        inlier = correspondences[inlier_idx, :]
        outlier = correspondences[outlier_idx, :]

        e = 1 - len(inlier)/n_total
        N = np.log(1-p) / (np.log(1-(1-e)**n))
        M = np.floor((1 - e) * n_total).astype(np.int32)

        inliers.append(inlier)
        outliers.append(outlier)
        len_inliers.append(len(inlier))

    largest_set_idx = np.argmax(len_inliers)
    if len_inliers[largest_set_idx] < M:
        warnings.warn(f"Largest inlier set size = {len_inliers[largest_set_idx]} < {M}. Relax
    constraints by changing params")

    best_inliers = np.array(inliers[largest_set_idx])
    best_outliers = np.array(outliers[largest_set_idx])

    return (best_inliers, best_outliers)
```

Listing 2: Python implementation of the RANSAC algorithm.

2.2.3 Nonlinear Least-Squares Minimization (using LM)

Once we obtain a LLS estimate of the homography H, we must refine it using Nonlinear Least-Squares (NLLS) to get a better estimate of H between two scenes. A common NLLS strategy used is Levenberg-Marquardt (LM) algorithm. It combines the numerical stability of Gradient Descent (GD) and the speed of Gauss-Newton (GN). The following pseudo-code is from here. The maximum number of iterations and other tolerance levels $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ are user inputs. The scaling factor τ for the damping coefficient μ is also an user input. The Jacobian is calculated using finite differences.

Algorithm 3 LM(Vector prediction function $\boldsymbol{f}$, measurement vector $\boldsymbol{x}$, Initial guess $\boldsymbol{p}_0$)

$n\_iter \leftarrow 0$
$\nu \leftarrow 2$
$\boldsymbol{p} \leftarrow \boldsymbol{p}_0$
$\boldsymbol{A} \leftarrow J_{\boldsymbol{f}}^{\top} J_{\boldsymbol{f}}$
$\epsilon_{\boldsymbol{p}} \leftarrow \boldsymbol{x} - \boldsymbol{f}(\boldsymbol{p})$
$\boldsymbol{g} \leftarrow J_{\boldsymbol{f}}^{\top} \boldsymbol{\epsilon}_{\boldsymbol{p}}$
$stop \leftarrow ||\boldsymbol{g}||_{\infty} \leq \varepsilon_1$
$\mu \leftarrow \tau \times \max_{i=1,\ldots,m}(\boldsymbol{A}_{ii})$
while (not $stop$) and $(n\_iter < max\_iter)$ **do**
  $n\_iter \leftarrow n\_iter + 1$
  Solve $(\boldsymbol{A} + \mu \boldsymbol{I})\boldsymbol{\delta}_{\boldsymbol{p}} = \boldsymbol{g}$
  if $||\boldsymbol{\delta}_{\boldsymbol{p}}||_2 \leq \varepsilon_2 ||\boldsymbol{p}||_2$ **then**
    $stop \leftarrow$ True
  else
    $\boldsymbol{p}_{new} \leftarrow \boldsymbol{p} + \boldsymbol{\delta}_{\boldsymbol{p}}$
    $\rho \leftarrow \dfrac{(||\boldsymbol{\epsilon}_{\boldsymbol{p}}||_2^2 - ||\boldsymbol{x} - \boldsymbol{f}(\boldsymbol{p})||_2^2)}{\boldsymbol{\delta}_{\boldsymbol{p}}^{\top}(\mu \boldsymbol{\delta}_{\boldsymbol{p}} + \boldsymbol{g})}$
    if $\rho > 0$ **then**
      $\boldsymbol{p} \leftarrow \boldsymbol{p}_{new}$
      $\boldsymbol{A} \leftarrow J_{\boldsymbol{f}}^{\top} J_{\boldsymbol{f}}$
      $\epsilon_{\boldsymbol{p}} \leftarrow \boldsymbol{x} - \boldsymbol{f}(\boldsymbol{p})$
      $\boldsymbol{g} \leftarrow J_{\boldsymbol{f}}^{\top} \boldsymbol{\epsilon}_{\boldsymbol{p}}$
      $stop \leftarrow (||g||_{\infty} \leq \varepsilon_1)$ or $(||\boldsymbol{\epsilon}_{\boldsymbol{p}}||_2^2 \leq \varepsilon_3)$
      $\mu \leftarrow \mu \times \max\left(1/3, 1 - (2\rho - 1)^3\right)$
      $\nu \leftarrow 2$
    else
      $\mu \leftarrow \mu\nu$
      $\nu \leftarrow 2\nu$
    end if
  end if
end while
return $\boldsymbol{p}$

```python
def LM_optim(fun:callable, x0:np.ndarray, tau:float=1e-3, eps1:float=1e-15, eps2:float=1e-15,
    eps3:float=1e-15, max_iter:int=100, args=(), kwargs={}):

    """
    Function to implement Levenberg-Marquardt optimization algorithm. Implementation follows
    pseudocode in https://users.ics.forth.gr/~lourakis/levmar/levmar.pdf.
    Jacobian is estimated using a 2-pt finite difference method.

    Args:
        fun (callable): Callable function which computes the residuals. Function signature and
     usage: ``fun(x, *args, **kwargs)``. Minimization proceeds w.r.t first argument.
        x0 (np.ndarray): Initial guess for the independent variables.
        tau (float): Scaling factor for the damping parameter mu. ``0 < tau <= 1``. Defaults
    to ``1e-3``.
        eps1 (float): Termination threshold for the inf-norm of the gradient. Termination
    condition: ``norm(grad, inf) <= eps1``. Defaults to ``1e-15``.
        eps2 (float): Relative termination threshold for delta. Termination condition: ``norm(
    delta) <= eps2 * norm(x)``. Defaults to ``1e-15``.
        eps3 (float): Termination threshold for the square of the norm of residuals. ``norm(
    fun(x))**2 <= eps3``. Defaults to ``1e-15``.
        max_iter (int): Maximum number of iterations.

```

```
    Returns:
        solution (OptimizeSolution): The solution to the optimization problem. ``
    OptimizeSolution`` has the following fields -
        1. x (np.ndarray): Solution to the optimization problem.
        2. errors (list): Values of the cost function (C) at every iteration. C = norm(fun(x,
    *args, **kwargs), 2)^2
        3. mu (float): Final value fo the damping parameter used in the algorithm.
        4. jac (np.ndarray): Value of the Jacobian at the end of optimization.
        5. hess (np.ndarray): Value of the Hessian at the end of optimization.
        6. n_iter (int): Number of iteration taken to reach termination conditions.
    """

    print("---------------------------- Starting LM optimization...
    ---------------------------- \n")

    m = len(x0)
    errors = []

    n_iter = 0
    nu = 2
    x = x0

    residual = fun(x, *args, **kwargs)

    jac = -approx_fprime(x, fun, 1e-8, *args) # '-' sign because taking jacobian of residual
    vector.

    hess = jac.T @ jac
    grad = jac.T @ residual

    stop = (norm(grad, np.inf) <= eps1)
    mu = tau * np.max(np.diag(hess))

    while (not stop) and (n_iter < max_iter):
        n_iter += 1
        errors.append(norm(residual, 2)**2)

        delta = solve(hess + mu*np.eye(m), grad)

        if (norm(delta, 2) <= eps2*norm(x, 2)):
            stop = True

        else:
            x_new = x + delta
            rho = (norm(residual, 2)**2 - norm(fun(x_new, *args, **kwargs), 2)**2) / (delta.T
    @ (mu*delta + grad))

            if rho > 0:
                x = x_new

                residual = fun(x, *args, **kwargs)
                jac = -approx_fprime(x, fun, 1e-8, *args)

                hess = jac.T @ jac
                grad = jac.T @ residual

                stop = (norm(grad, np.inf) <= eps1) or (norm(residual, 2)**2 <= eps3)
                mu *= max(1/3, 1-(2*rho - 1)**3)
                nu = 2

            else:
                mu *= nu
                nu *= 2
    else:
        if stop:
            print(f"Stopped on reaching termination conditions in {n_iter} iteration(s). \n")
            print("
    ------------------------------------------------------------------------------------------")
        else:
            print("Stopped on reaching maximum iteration(s). \n")
            print("
    ------------------------------------------------------------------------------------------")

    class OptimizeSolution:
        def __init__(self):
            self.x = None
```

```
            self.errors = None
            self.mu = None
            self.jac = None
            self.hess = None
            self.n_iter = None

        def set_solution(self, x, errors, mu, jac, hess, n_iter):
            self.x = x
            self.errors = errors
            self.mu = mu
            self.jac = jac
            self.hess = hess
            self.n_iter = n_iter


    solution = OptimizeSolution()
    solution.set_solution(x, errors, mu, jac, hess, n_iter)

    return solution
```

Listing 3: Python implementation of Levenberg-Marquardt optimization algorithm.

2.2.4 Panoramic Stitching

To be able to place all projections on a single canvas to create a panorama, we must first decide a reference image with respect to whom we will find the homography H. We will choose the middle image as the reference that is placed at the center of the canvas. In case of odd number of images, the middle is well defined. But if there were even number of images, we can just choose the middle be $(\frac{M}{2} - 1)$.
Now we must calculate the homography with respect to this reference image.
Let $H_{i\to j}$ represent the homography that takes domain plane i to range plane j. Therefore,

$$H_{j\to i} = H_{i\to j}^{-1}$$

In our case, i and j are the indices of the images. I will demonstrate this for the case of 5 images. The reference index would be 3. We can find $H_{1\to 2}, H_{2\to 3}, H_{3\to 4}, H_{4\to 5}$ using the automatic homography estimation pipeline discussed previously. We must find $H_{i\to 3}$ for all i.

$$\begin{aligned}
H_{1\to 3} &= H_{1\to 2} \times H_{2\to 3} \\
H_{2\to 3} &= H_{2\to 3} \\
H_{3\to 3} &= I \\
H_{4\to 3} &= H_{3\to 4}^{-1} \\
H_{5\to 3} &= H_{5\to 4} \times H_{4\to 3} = H_{4\to 5}^{-1} \times H_{3\to 4}^{-1}
\end{aligned}$$

Once we have this we must find out the size of canvas which will help us to place the reference image in the middle of the canvas. I found this by projecting the corners of the i^{th} image using $H_{i\to 3}$ and

$$\text{Height of canvas} = \max(\text{height of projected image using the corners})$$

$$\text{Width of canvas} = \sum(\text{width of image})$$

Of course, there will be some overlap between the projected images. This will be taken care by trimming away all the zeros along the borders after all images are placed on the canvas.
To place the reference image at the center of the canvas, we define

$$t_x = \frac{\text{Width of canvas - Width of reference image}}{2}$$

$$t_y = \frac{\text{Height of canvas - Height of reference image}}{2}$$

and

$$H_{\text{translation}} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}.$$

Now we must multiply every $H_{i\to 3}$ by this translation homography to obtain the final homography that needs to be applied to the i^{th} image.

The homography can be applied using two interpolation strategies - nearest neighbors and bi-linear interpolation. Bi-linear interpolation gives us a better quality of image because it interpolates for a pixel at (x, y) using all four of its nearest neighbors. The nearest neighbor interpolation just picks the pixel value at $(\lfloor x \rfloor, \lfloor y \rfloor)$ as the value of the pixel at (x, y).

```
def generate_pano(image_filenames:list, matcher:str="SP_SG", optimizer:str="scipy_lm", interp:str="bilinear"):

    """
    Function to create a panorama given a set of images. The middle image is placed at the center of the canvas and then homographies are found relative to the middle image.

    Args:
        image_filenames (list): List that contains the paths to all images to create a panorama.
        matcher (str): Select which feature detection and matching algorithm to use. If ``matcher`` is "sift", correspondence are found using Scale-Invariant Feature Transform (SIFT) algorithm. If ``matcher`` is "SP_SG", correspondence are found using SuperPoint+SuperGlue deep learning pre-trained network. Defaults to "SP_SG". Check ``superglue_wrapper.py`` for more information to run the SuperPoint+SuperGlue pre-trained network.
        optimizer (str): Select if homography should be refined using the LM algorithm. If ``optimizer`` is ``None``, the homography is not refined. If ``optimizer`` is "scipy_lm", the homography is refined using the in-built SciPy implementation of the LM algorithm. If ``optimizer`` is "lm", the homography is refined using my implementation of the LM algorithm.
        interp (str): Select which interpolation strategy to use when applying the homogrpahy. If ``interp`` is "n_neighbors", uses nearest neighbors interpolation. If ``interp`` is "bilinear", uses bi-linear interpolation. Defaults to "bilinear".

    Returns:
        canvas (np.ndarray): The canvas that contains the panorama.
    """

    H_all = []
    n_images = len(image_filenames)

    for i in range(0, n_images-1):

        img1 = image_filenames[i]
        img2 = image_filenames[i+1]
        H = auto_estimate_homography(img1, img2, matcher, optimizer).H
        H_all.append(H)

    ref_idx = ((n_images+1)//2) - 1

    H_wrt_ref_all = [np.eye(3, dtype=np.float64)]*(len(H_all)+1)

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(ref_idx+1, n_images):
        H_wrt_ref = H_wrt_ref @ np.linalg.inv(H_all[i-1])
        H_wrt_ref_all[i] = H_wrt_ref

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(ref_idx-1, -1, -1):
        H_wrt_ref = H_wrt_ref @ H_all[i]
        H_wrt_ref_all[i] = H_wrt_ref

    image_all = [cv.imread(image_filenames[i]) for i in range(n_images)]

    temp = [find_projected_corner_h(image_all[i], H_wrt_ref_all[i]) for i in range(n_images)]

    canvas_h = np.max(temp)

    canvas_w = np.sum([image_all[i].shape[1] for i in range(n_images)])

    canvas = np.zeros((canvas_h, canvas_w, 3), dtype=np.uint8)

    tx = (canvas_w-image_all[ref_idx].shape[1]) // 2

    ty = (canvas_h-image_all[ref_idx].shape[0]) // 2


    H_tr = np.array([1, 0, tx, 0, 1, ty, 0, 0, 1])
    H_tr = np.reshape(H_tr, (3, 3))

```

```
    for i in range(n_images):
        H_wrt_ref_all[i] = H_tr @ H_wrt_ref_all[i]

    for i in range(0, n_images):
        canvas = apply_homography(image_all[i], canvas, H_wrt_ref_all[i], interp)

        ## If want to view the panorama after every iteration

        # cv.namedWindow("canvas", cv.WINDOW_NORMAL)
        # cv.resizeWindow("canvas", 600, 600)
        # cv.imshow("canvas", canvas)
        # cv.waitKey(0)
        # cv.destroyAllWindows()

    canvas = trim_zeros(canvas)
    return canvas
```

Listing 4: Python implementation to create a panorama.

3 Programming Task - 1

3.1 Input Images

(a) `1.jpg`

(b) `2.jpg`

(c) `3.jpg`

(d) `4.jpg`

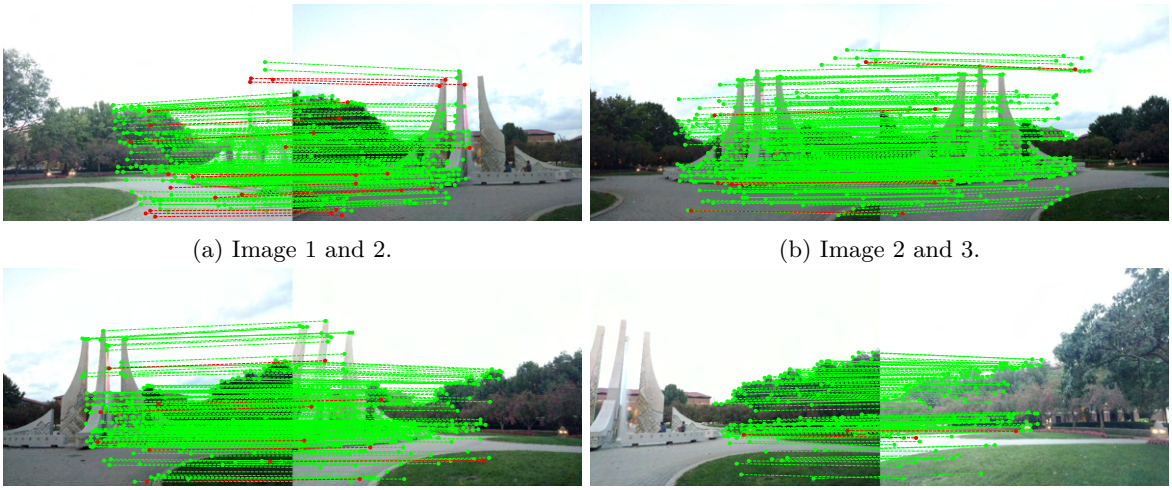
(e) `5.jpg`

Figure 1: Input images to create a panorama.

Paramaters for RANSAC	Value
δ for SIFT	30
δ for SP+SG	2
p	0.99
n	8

Parameters for LM	Value
τ	10^{-3}
ε_1	10^{-15}
ε_2	10^{-15}
ε_3	10^{-15}
max_iter	100
δ used in Jacobian	10^{-8}

3.2 Correspondences - Inliers and Outliers



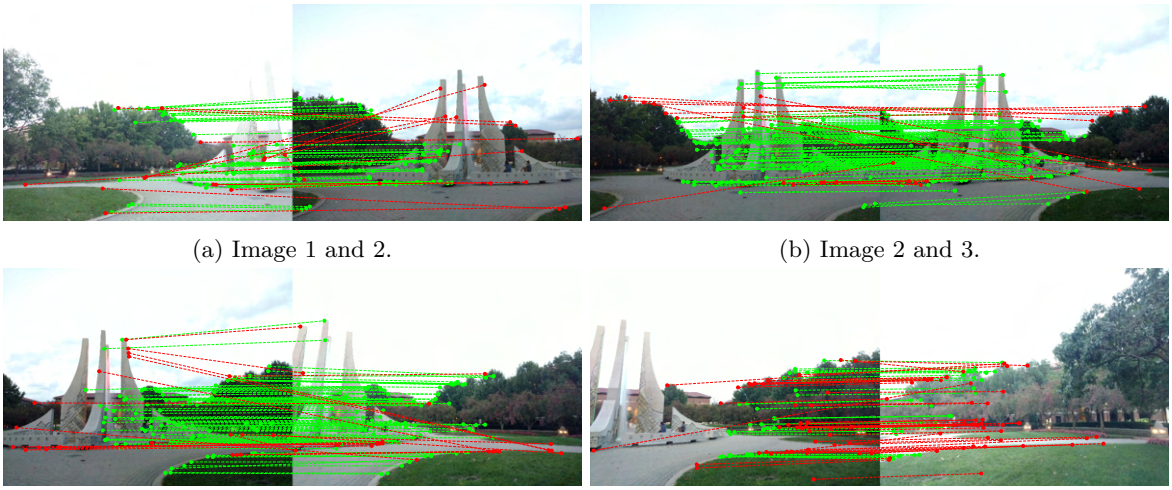
(a) Image 1 and 2.

(b) Image 2 and 3.

(c) Image 3 and 4.

(d) Image 4 and 5.

Figure 2: Using SuperPoint+SuperGlue (SP+SG) feature detector and matcher.



(a) Image 1 and 2.

(b) Image 2 and 3.

(c) Image 3 and 4.

(d) Image 4 and 5.

Figure 3: Using SIFT feature detector and matcher.

3.3 Panoramic stitching

I will show the results using the SP+SG feature matcher because it is more robust as it detects higher number of inliers used to find the homography between the two scenes.

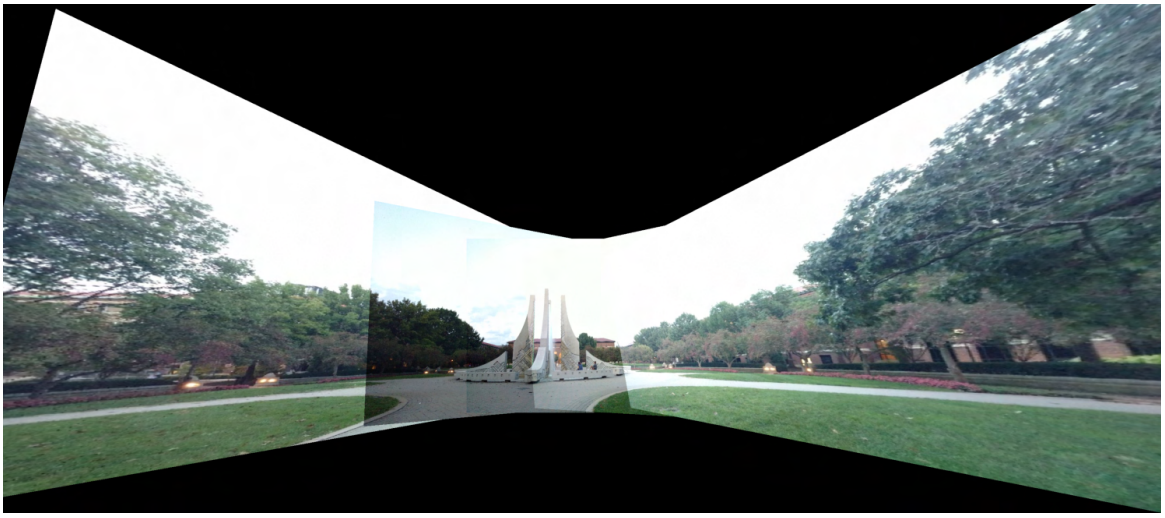


Figure 4: Stitching using no nonlinear least-squares optimization and bi-linear interpolation when applying H.

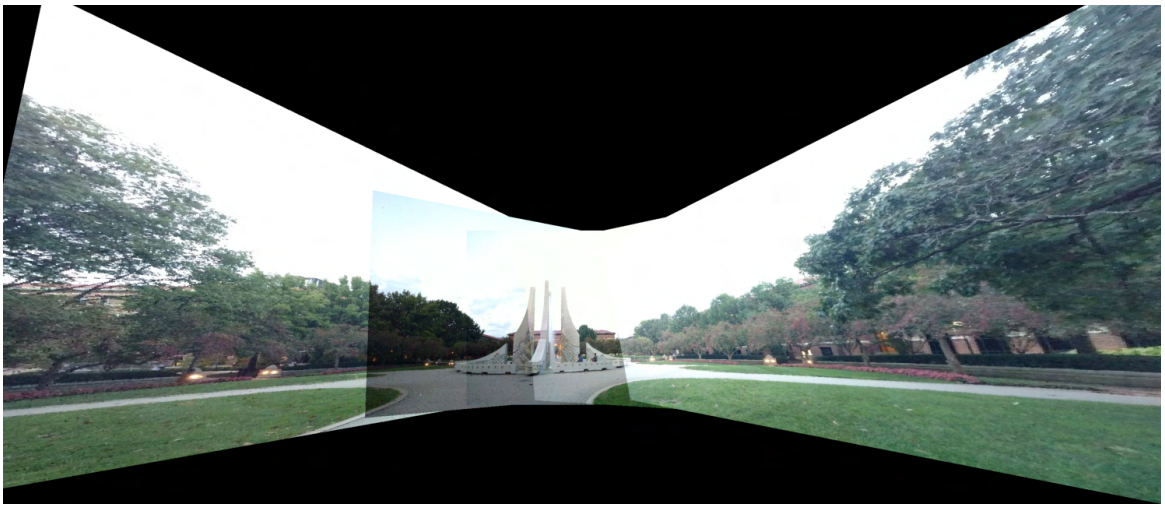


Figure 5: Stitching using no nonlinear least-squares optimization and nearest neighbors interpolation when applying H.

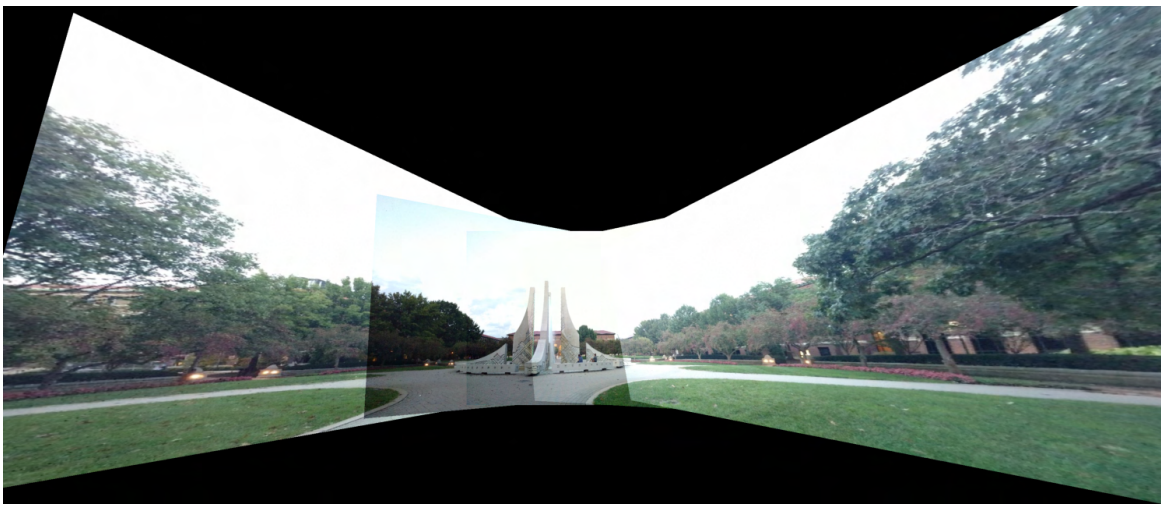


Figure 6: Stitching using SciPy's LM optimization and bi-linear interpolation when applying H.

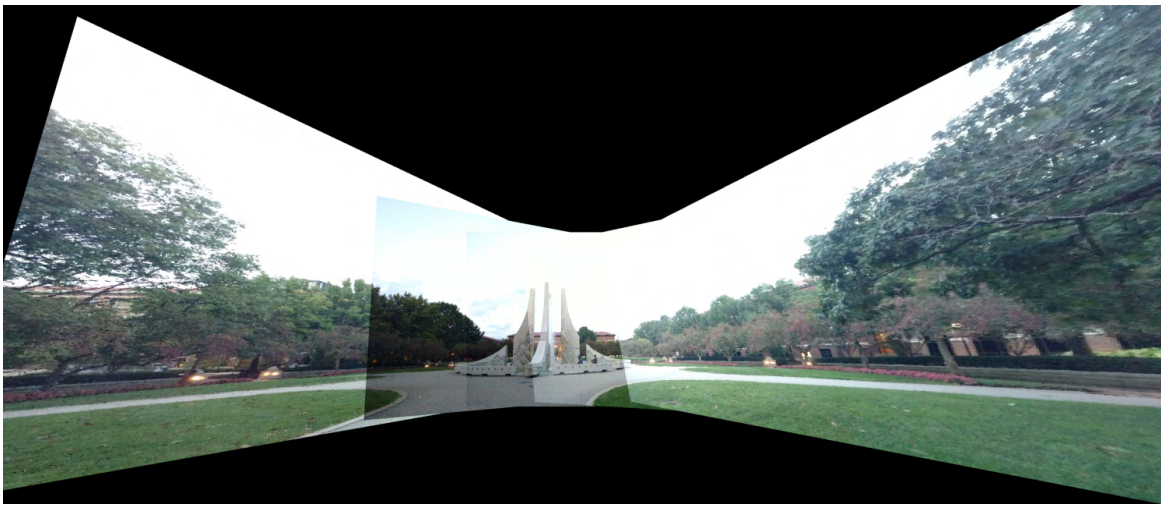


Figure 7: Stitching using SciPy's LM optimization and nearest neighbors interpolation when applying H.

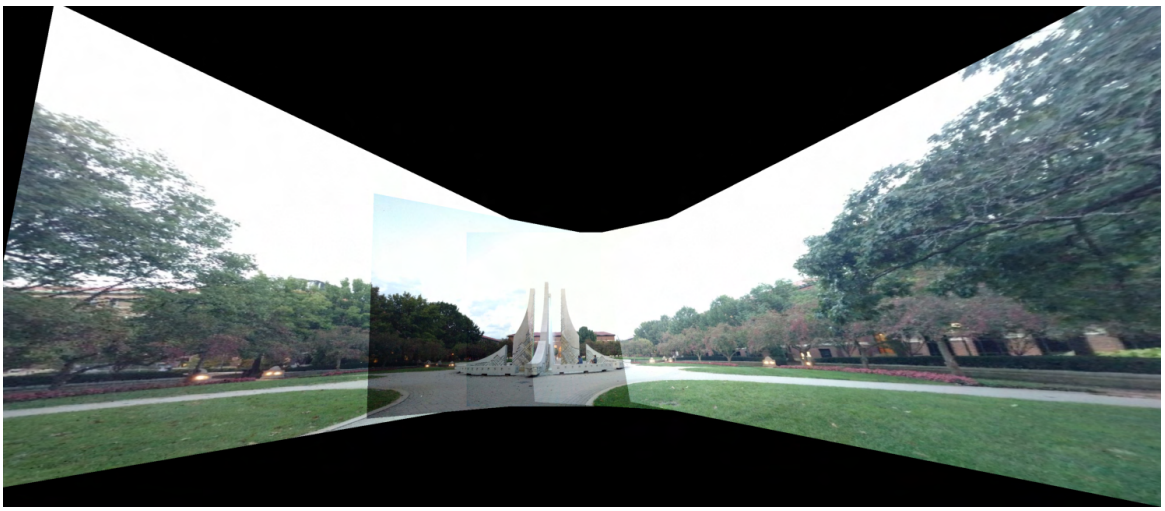


Figure 8: Stitching using my implementation of LM optimization and bi-linear interpolation when applying H.

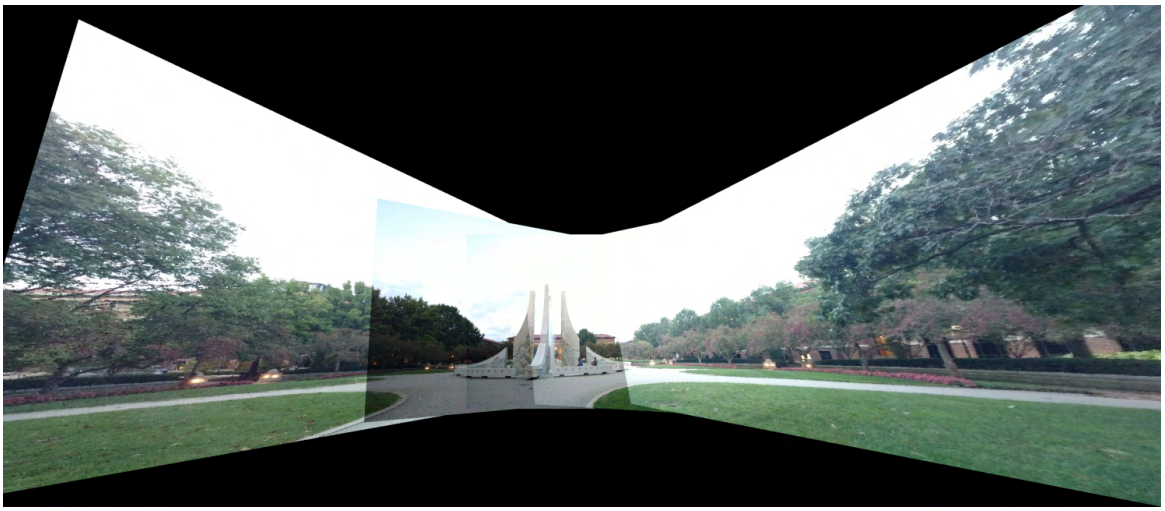


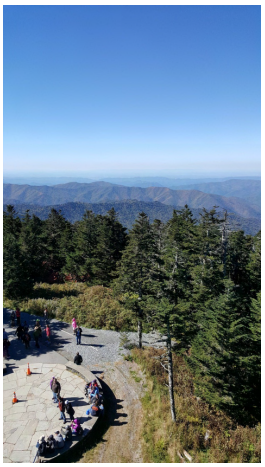
Figure 9: Stitching using my implementation of LM optimization and nearest neighbors interpolation when applying H.

4 Programming Task - 2

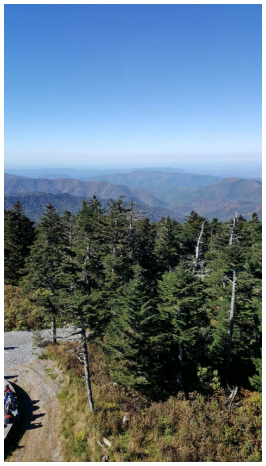
4.1 Input Images

(a) `1.jpg`

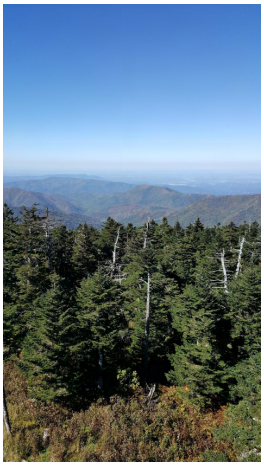
(b) `2.jpg`



(c) `3.jpg`



(d) `4.jpg`



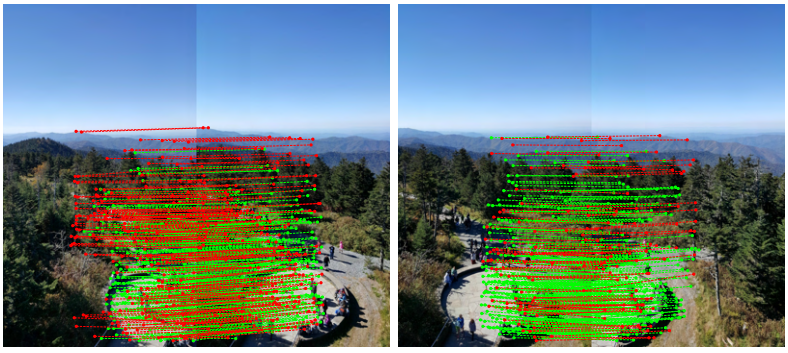
(e) `5.jpg`

Figure 10: Input images to create a panorama.

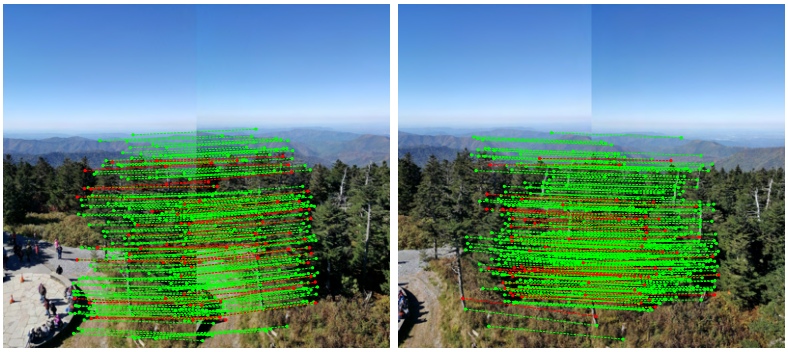
Paramaters for RANSAC	Value
δ for SIFT	1
δ for SP+SG	2
p	0.99
n	8

Parameters for LM	Value
τ	10^{-3}
ε_1	10^{-15}
ε_2	10^{-15}
ε_3	10^{-15}
max_iter	100
δ used in Jacobian	10^{-8}

4.2 Correspondences - Inliers and Outliers

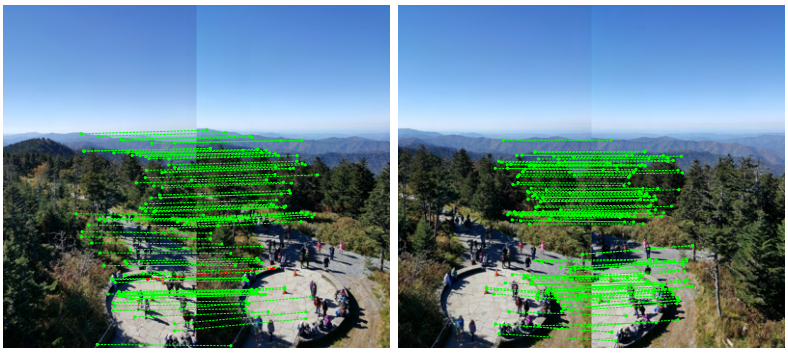


(a) Image 1 and 2. (b) Image 2 and 3.

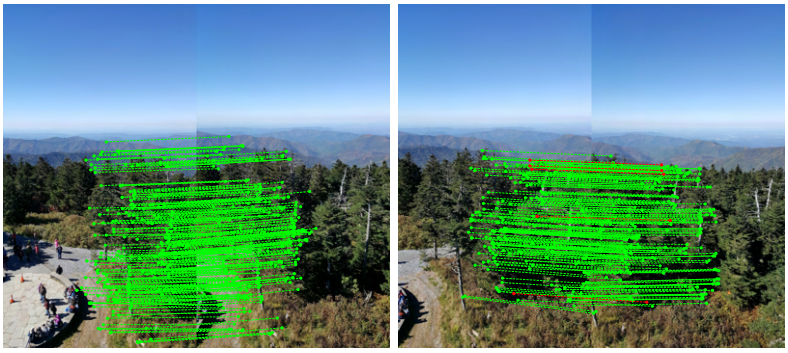


(c) Image 3 and 4. (d) Image 4 and 5.

Figure 11: Correspondences found using SuperPoint+SuperGlue (SP+SG) feature detector and matcher.



(a) Image 1 and 2. (b) Image 2 and 3.



(c) Image 3 and 4. (d) Image 4 and 5.

Figure 12: Correspondences found using SIFT feature detector and matcher.

4.3 Panoramic Stitching

We see that SIFT performs well in detecting and matching the keypoints, so I will only show results using SIFT as the matcher function.

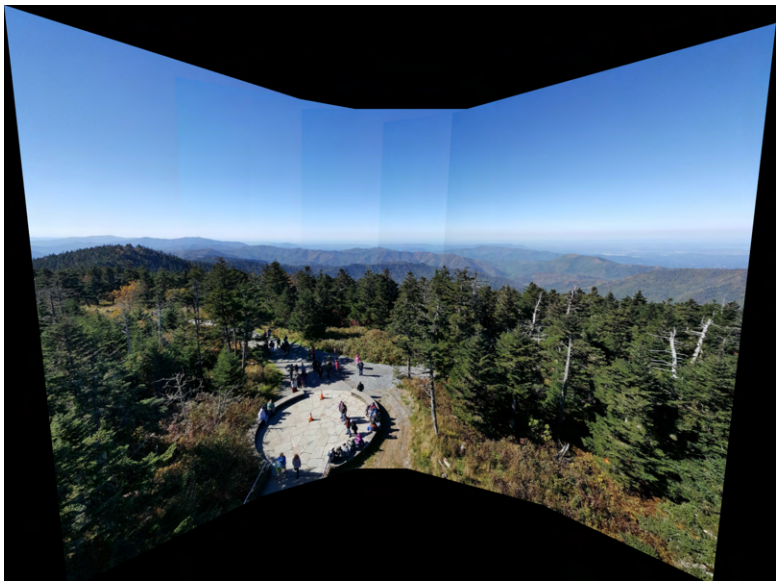


Figure 13: Stitching using no nonlinear least-squares optimization and bi-linear interpolation when applying H.

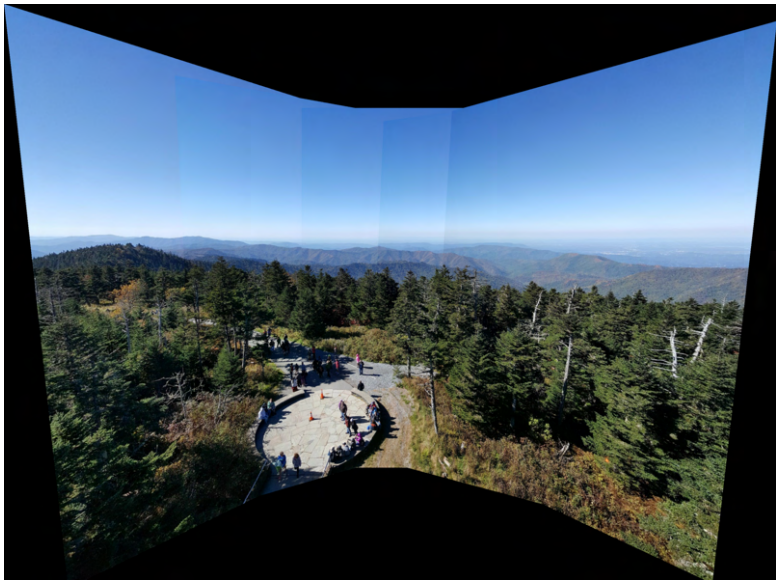


Figure 14: Stitching using no nonlinear least-squares optimization and nearest neighbors interpolation when applying H.

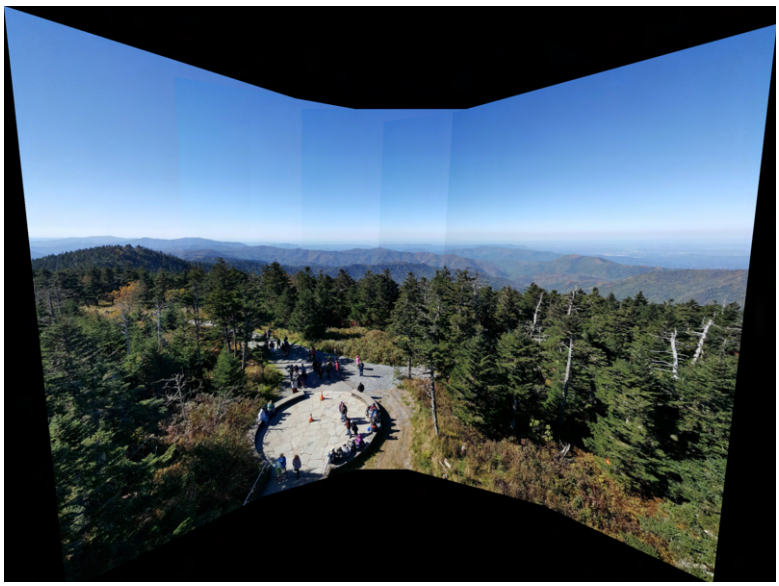


Figure 15: Stitching using SciPy's LM optimization and bi-linear interpolation when applying H.

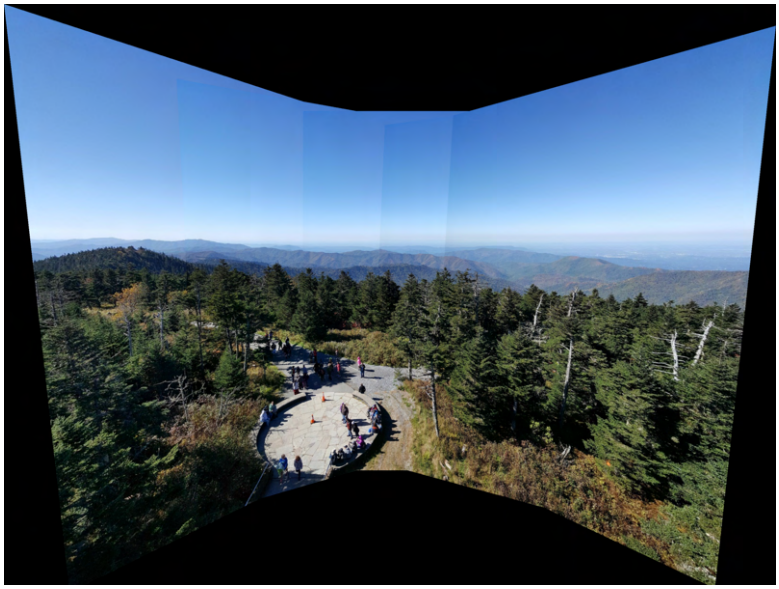


Figure 16: Stitching using SciPy's LM optimization and nearest neighbors interpolation when applying H.

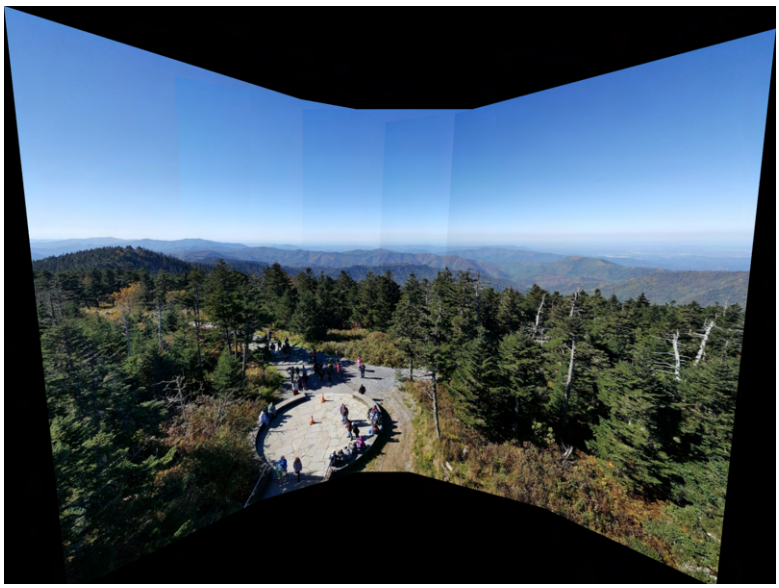


Figure 17: Stitching using my implementation of LM optimization and bi-linear interpolation when applying H.

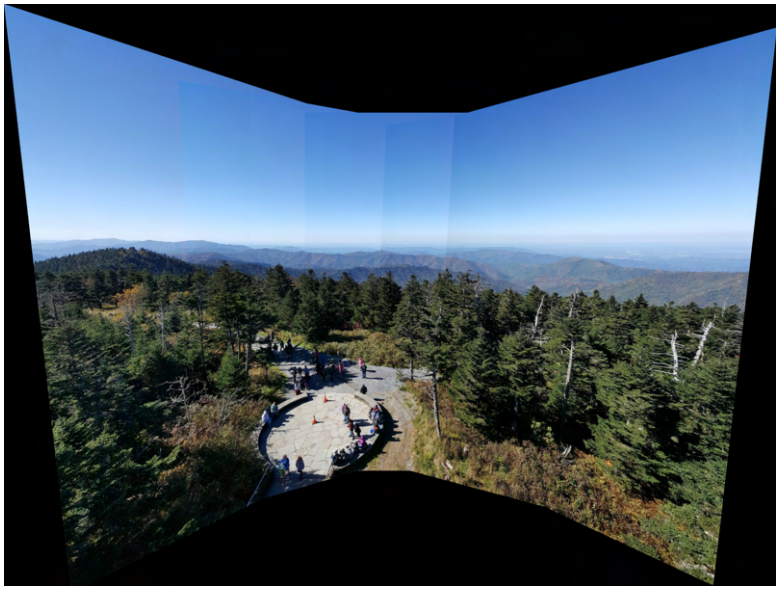
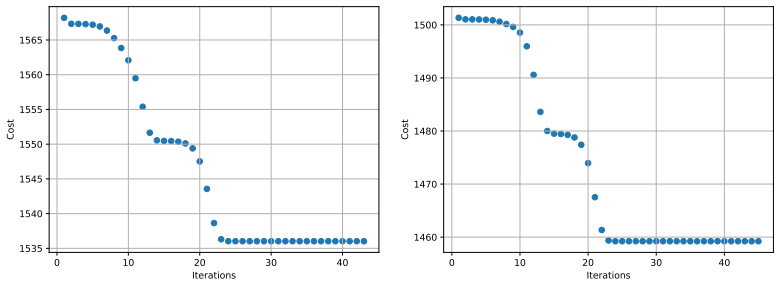


Figure 18: Stitching using my implementation of LM optimization and nearest neighbors interpolation when applying H.

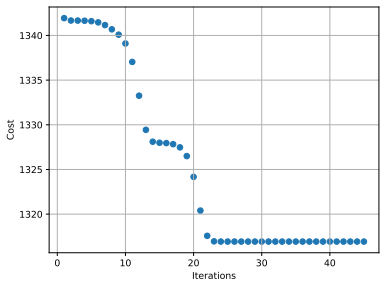
5 Extra Credit - LM algorithm implementation

I implemented the Levenberg-Marquardt algorithm and we can visualize the reduction in the cost function using the following graphs -

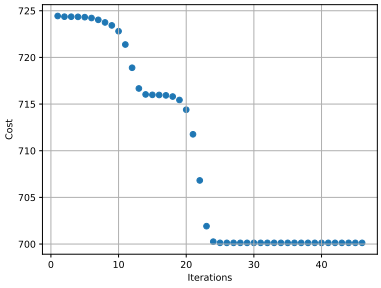


(a) Between images 1 and 2

(b) Between images 2 and 3

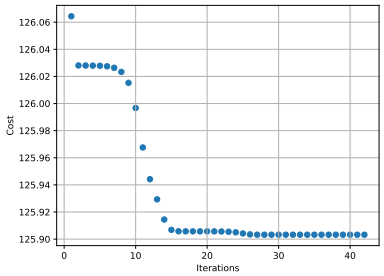


(c) Between images 3 and 4

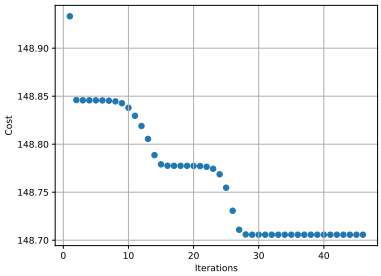


(d) Between images 4 and 5

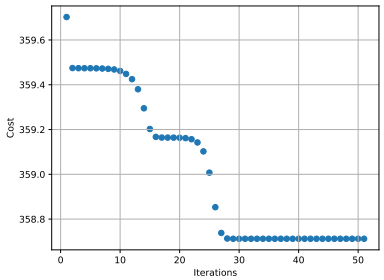
Figure 19: LM iterations between the fountain images using the SP+SG matcher



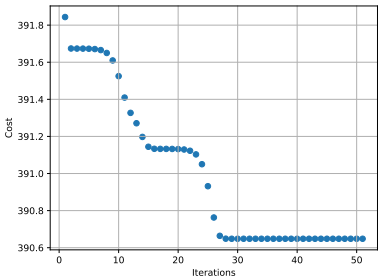
(a) Between images 1 and 2



(b) Between images 2 and 3



(c) Between images 3 and 4



(d) Between images 4 and 5

Figure 20: LM iterations between the hill images using the SIFT matcher

6 Source Code

6.1 main.py

```python
import cv2 as cv
import os
import matplotlib.pyplot as plt

from homography_estimation import auto_estimate_homography, draw_inliers_outliers
from panorama import generate_pano

if __name__ == '__main__':

    img_dir = "pics"
    names = ["fountain", "building", "hill"]
    out_dir = "outs"

    optimizers = [None, "scipy_lm", "lm"]

    interps = ["bilinear", "n_neighbors"]

    matchers = ["SP_SG", "sift"]

    for name in names[-1:]:
        n_images = 5
        images = [os.path.join(img_dir, name, f"{i}.jpg") for i in range(1, n_images+1)]

        for optimizer in optimizers[:]:
            for matcher in matchers:
                for i in range(1, n_images):

                    img1 = images[i-1]
                    img2 = images[i]
                    homography_sol = auto_estimate_homography(img1, img2, matcher, optimizer)

                    H = homography_sol.H
                    inliers = homography_sol.inliers
                    outliers = homography_sol.outliers

                    in_out_filename = os.path.join(out_dir, name, f"{name}_{optimizer}_{matcher}_in_out_{i}_{i+1}.pdf")
                    draw_inliers_outliers(img1, img2, inliers, outliers, in_out_filename)

                    if optimizer == "lm":
                        errors = homography_sol.optim_solution.errors
                        n_iter = homography_sol.optim_solution.n_iter
                        plt.scatter(range(1, n_iter+1), errors)
                        plt.grid()
                        plt.xlabel("Iterations")
                        plt.ylabel("Cost")
                        error_filename = os.path.join(out_dir, name, f"{name}_{matcher}_error_{i}_{i+1}.pdf")
                        plt.savefig(error_filename, bbox_inches="tight", dpi=300)
                        plt.close()


        for optimizer in optimizers:
            for matcher in matchers:
                for interp in interps:
                    pano_img = generate_pano(images, matcher=matcher, optimizer=optimizer, interp=interp)
                    pano_filename = os.path.join(out_dir, name, f"{name}_pano_{optimizer}_{matcher}_{interp}.pdf")
                    plt.imshow(cv.cvtColor(pano_img, cv.COLOR_BGR2RGB))
                    plt.axis('off')
                    plt.savefig(pano_filename, bbox_inches='tight', pad_inches=0, dpi=300)
                    plt.close()
```

6.2 homography_estimation.py

```python
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares
import warnings

from superglue_wrapper import SuperGlue

from LM_optim import LM_optim

eps = np.finfo(float).eps

def draw_inliers_outliers(img1:str, img2:str, inliers:np.ndarray, outliers:np.ndarray, filename:str):
    """
    Function to draw the inliers and outliers after RANSAC.

    Args:
        img1 (str): Path to image 1.
        img2 (str): Path to image 2.
        inliers (np.ndarray): Set of inliers.
        outliers (np.ndarray): Set of outliers.
        filename (str): Filename to save the figure.
    """

    img1 = cv.imread(img1)
    img2 = cv.imread(img2)

    ms = 3
    lw = 0.5

    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    result = np.zeros((np.max([h1, h2]), w1 + w2, 3), dtype=np.uint8)
    result[:h1, :w1] = img1
    result[:h2, w1:] = img2
    result = cv.cvtColor(result, cv.COLOR_BGR2RGB)

    in_pts1_x = inliers[:, 0]
    in_pts1_y = inliers[:, 1]
    in_pts2_x = inliers[:, 2] + w1
    in_pts2_y = inliers[:, 3]

    out_pts1_x = outliers[:, 0]
    out_pts1_y = outliers[:, 1]
    out_pts2_x = outliers[:, 2] + w1
    out_pts2_y = outliers[:, 3]

    plt.imshow(result)
    c_g = [0, 1, 0]
    c_r = [1, 0, 0]
    plt.plot([in_pts1_x, in_pts2_x], [in_pts1_y, in_pts2_y], color=c_g, linestyle='--', linewidth=lw, marker=".", ms=ms, markerfacecolor=c_g)
    plt.plot([out_pts1_x, out_pts2_x], [out_pts1_y, out_pts2_y], color=c_r, linestyle='--', linewidth=lw, marker=".", ms=ms, markerfacecolor=c_r)

    plt.axis('off')
    plt.savefig(filename, bbox_inches='tight', pad_inches=0, dpi=300)
    plt.close()

    return

def _SP_SG_feature_match(img1:str, img2:str):

    """
    Wrapper function to use SuperPoint+SuperGlue pre-trained network to detect and match features.

    Args:
        img1 (str): Path to image 1.
        img2 (str): Path to image 2.

    Returns:
```

```
        tuple: Returns in order
                1. Matched keypoints of image 1
                2. Matched keypoints of image 2
                3. All keypoints of image 1
                4. All keypoints of image 2
                5. Desciptor vector associated to all keypoints of image 1
                6. Desciptor vector associated to all keypoints of image 2
    """

    detector = SuperGlue.create()
    kp0, descriptor0 = detector.detect_and_compute(img1)
    kp1, descriptor1 = detector.detect_and_compute(img2)

    mkpts0, mkpts1 = detector.match(img1, img2)

    return (mkpts0, mkpts1, kp0, kp1, descriptor0, descriptor1)

def _SIFT_feature_match(img1:str, img2:str):

    """
    Wrapper function to use Scale-Invariant Feature Transform (SIFT) to detect and match features.

    Args:
        img1 (str): Path to image 1.
        img2 (str): Path to image 2.

    Returns:
        tuple: Returns in order
                1. Matched keypoints of image 1
                2. Matched keypoints of image 2
                3. All keypoints of image 1
                4. All keypoints of image 2
                5. Desciptor vector associated to all keypoints of image 1
                6. Desciptor vector associated to all keypoints of image 2
    """

    img1_gray = cv.imread(img1, cv.IMREAD_GRAYSCALE)
    img2_gray = cv.imread(img2, cv.IMREAD_GRAYSCALE)


    sift = cv.SIFT_create()
    kp0, descriptor0 = sift.detectAndCompute(img1_gray, None)
    kp1, descriptor1 = sift.detectAndCompute(img2_gray, None)

    bf = cv.BFMatcher(cv.NORM_L2, True)
    matches = bf.match(descriptor0, descriptor1)

    matches = sorted(matches, key=lambda x: x.distance)

    mkpts0 = np.array([kp0[match.queryIdx].pt for match in matches if match.distance <= 100])
    mkpts1 = np.array([kp1[match.trainIdx].pt for match in matches if match.distance <= 100])

    return (mkpts0, mkpts1, kp0, kp1, descriptor0, descriptor1)

def H_p_to_p(pts1:np.ndarray, pts2:np.ndarray):

    """
    Function to estimate Linear Least Squares fit of the homography matrix H such that pts2 = H * pts1.

    Args:
        pts1 (np.ndarray): Set of points on domain plane. Must be in homogeneous coordinates.
        pts2 (np.ndarray): Set of matching points on range plane. Must be in homogeneous coordinates.

    Returns:
        np.ndarray: Homography matrix such that pts2 = H * pts1
    """

    assert len(pts1) == len(pts2)

    assert len(pts1) >= 4

    A = []

```

```
    for (x, y, w), (xp, yp, wp) in zip(pts1, pts2):
        A.append([0, 0, 0, -wp*x, -wp*y, -wp*w, yp*x, yp*y, yp*w])
        A.append([wp*x, wp*y, wp*w, 0, 0, 0, -xp*x, -xp*y, -xp*w])

    A = np.array(A)
    _, _, vt = np.linalg.svd(A)

    h = vt.T[:, -1]

    H = np.reshape(h, (3, 3))

    H /= H[2, 2] + eps

    return H

def RANSAC(pts1:np.ndarray, pts2:np.ndarray, sigma:float, p:float, n:int, e:float=None):

    """
    Function to apply the RANSAC algorithm to find inliers and outliers of matching set of
    points pts1 and pts2.

    Args:
        pts1 (np.ndarray): Set of points on domain plane. Must be in regular coordinates.
        pts2 (np.ndarray): Set of matching points on range plane. Must be in regular
    coordinates.
        sigma (float): Estimate of noise induced on the noisy matches.
        p (float): Probability that atleast 1 trial will be free of outliers. ``0 <= p <= 1``
        n (int): Number of correspondences used to find the LLS estimate of homography H. ``4
    < n < 10``
        e (float, optional): Probability that chosen correspondence is an outlier. Defaults to
     ``None`` as it is adaptively calculated. ``0 <= e <= 1``

    Returns:
        (np.ndarray, np.ndarray): Tuple consisting of the inliers and outliers with the
    maximum inlier support. Each row is a correspondence in the order (domain.x, domain.y,
    range.x, range.y)
    """

    assert len(pts1) == len(pts2)

    n_total = len(pts1)

    if (p < 0) or (p > 1):
        raise ValueError(f"Invalid value of p={p}. `p` must be within [0, 1].")

    if (n < 4) or (n > 10):
        raise ValueError(f"Invalid value of n={n}. `n` must be within (4, 10).")

    if e is None:
        e = 1
    else:
        if (e < 0) or (e > 1):
            raise ValueError(f"Invalid value of e={e}. `e` must be within [0, 1].")


    correspondences = np.hstack((pts1, pts2))

    delta = 3 * sigma

    N = np.inf
    sample_count = 0


    M = np.floor((1 - e) * n_total).astype(np.int32)


    pts1_hc = np.hstack((pts1, np.ones((n_total, 1))))
    pts2_hc = np.hstack((pts2, np.ones((n_total, 1))))

    outliers = []
    inliers = []
    len_inliers = []

    while (N > sample_count):
        sample_count += 1
        sample_idx = np.random.permutation(np.arange(n_total))[:n]
```

```
        pts1_selected = pts1_hc[sample_idx]
        pts2_selected = pts2_hc[sample_idx]

        H = H_p_to_p(pts1_selected, pts2_selected)

        pts1_estimate = np.linalg.inv(H) @ pts2_hc.T
        pts1_estimate /= pts1_estimate[2] + eps
        pts1_estimate = pts1_estimate.T
        dist1 = np.sum(np.power(pts1_estimate[:, :2] - pts1, 2), 1)

        pts2_estimate = H @ pts1_hc.T
        pts2_estimate /= pts2_estimate[2] + eps
        pts2_estimate = pts2_estimate.T
        dist2 = np.sum(np.power(pts2_estimate[:, :2] - pts2, 2), 1)

        dists = np.sqrt(dist1 + dist2) / 2

        inlier_idx = np.where(dists <= delta)[0]
        outlier_idx = np.where(dists > delta)[0]

        inlier = correspondences[inlier_idx, :]
        outlier = correspondences[outlier_idx, :]

        e = 1 - len(inlier)/n_total
        N = np.log(1-p) / (np.log(1-(1-e)**n))
        M = np.floor((1 - e) * n_total).astype(np.int32)


        inliers.append(inlier)
        outliers.append(outlier)
        len_inliers.append(len(inlier))

    largest_set_idx = np.argmax(len_inliers)
    if len_inliers[largest_set_idx] < M:
        warnings.warn(f"Largest inlier set size = {len_inliers[largest_set_idx]} < {M}. Relax
    constraints by changing params")

    best_inliers = np.array(inliers[largest_set_idx])
    best_outliers = np.array(outliers[largest_set_idx])

    return (best_inliers, best_outliers)

def cost_fun(p, inliers):
    """
    Cost function to return the residuals used in the LM algorithm.
    """

    X = []
    F = []

    for (x, y, xp, yp) in inliers:
        X.append(xp)
        X.append(yp)

        F.append((p[0]*x + p[1]*y + p[2]) / (p[6]*x + p[7]*y + p[8]))
        F.append((p[3]*x + p[4]*y + p[5]) / (p[6]*x + p[7]*y + p[8]))

    X = np.array(X)
    F = np.array(F)

    return X - F

def _scipy_lm_wrapper(cost_fun, H, inliers):
    """
    Wrapper function to execute SciPy's LM algorithm.
    """
    return least_squares(cost_fun, H, method='lm', args=[inliers])

def _LM_optim_wrapper(cost_fun, H, inliers):
    """
    Wrapper function to execute my implementation of LM algorithm.
    """
    return LM_optim(cost_fun, H, args=[inliers])
```

```
289 def auto_estimate_homography(img1:str, img2:str, matcher:str="SP_SG", optimizer:str="scipy_lm"
    ):
290
291     """
292     Function to automatically estimate the homography between two images using
293     feature detection and matching algorithms followed by the application of the
294     RAndom SAmpling Consensus (RANSAC) algorithm to remove false correspondences. If ``
        optimizer`` is not ``None``, the homography is refined using Levenberg-Marquardt(LM) non-
        linear least squares optimization algorithm.
295
296     Args:
297         img1 (str): Path to image 1.
298         img2 (str): Path to image 2.
299         matcher (str): Select which feature detection and matching algorithm to use. If ``
        matcher`` is "sift", correspondence are found using Scale-Invariant Feature Transform (
        SIFT) algorithm. If ``matcher`` is "SP_SG", correspondence are found using SuperPoint+
        SuperGlue deep learning pre-trained network. Defaults to "SP_SG". Check ``
        superglue_wrapper.py`` for more information to run the SuperPoint+SuperGlue pre-trained
        network.
300         optimizer (str): Select if homography should be refined using the LM algorithm. If ``
        optimizer`` is None, the homography is not refined. If ``optimizer`` is "scipy_lm", the
        homography is refined using the in-built SciPy implementation of the LM algorithm. If ``
        optimizer`` is "lm", the homography is refined using my implementation of the LM algorithm
        .
301
302     Returns:
303         (Homography_sol): The solution of the automatic homography estimation. ``
        Homography_sol`` has the following fields -
304         1. H (np.ndarray): Estimated homography.
305         2. solution: If ``optimizer`` is not ``None``, this is set as the solution field
        retured from either SciPy LM optimization or my implementation of the LM optimization
        algorithm. Else is ``None``.
306         3. inliers (np.ndarray): Set of inliers with the best inlier support.
307         4. outlier (np.ndarray): Corresponding set of outliers.
308         5. kp1 (np.ndarray): Keypoints of image 1.
309         6. kp2 (np.ndarray): Keypoints of image 2.
310         7. desc1 (np.ndarray): Descriptor vector of the keypoints of image 1.
311         8. desc2 (np.ndarray): Descriptor vector of the keypoints of image 2.
312
313     """
314
315     val_matcher = {"sift": _SIFT_feature_match,
316                    "SP_SG": _SP_SG_feature_match}
317
318     try:
319         matcher_fun = val_matcher[matcher]
320
321     except KeyError:
322         raise ValueError(f"Invalid matcher function. Choose from {list(val_matcher.keys())}")
323
324     default_matcher_params = {"sift": [1, 0.99, 8, None],
325                               "SP_SG": [2, 0.99, 8, None]}
326
327     val_optimzer = [None, "scipy_lm", "lm"]
328
329     if optimizer not in val_optimzer:
330         raise ValueError(f"Invalid optimizer method. Chose from {val_optimzer}")
331
332
333     (mkpts1, mkpts2, kp1, kp2, desc1, desc2) = matcher_fun(img1, img2)
334
335     r_params = default_matcher_params[matcher]
336
337     inliers, outliers = RANSAC(mkpts1, mkpts2, r_params[0], r_params[1], r_params[2], r_params
    [3])
338
339     inliers_1_hc = np.hstack((inliers[:, :2], np.ones((inliers.shape[0], 1))))
340     inliers_2_hc = np.hstack((inliers[:, 2:], np.ones((inliers.shape[0], 1))))
341
342     H = H_p_to_p(inliers_1_hc, inliers_2_hc)
343
344     if optimizer is None:
345         print("No optimization applied. \n")
346
347     else:
348
```

```
        if optimizer == "scipy_lm":
            print("Using scipy LM optimization. \n")
            optim_fun = _scipy_lm_wrapper
        else:
            print("Using implemented LM optimization. \n")
            optim_fun = _LM_optim_wrapper

        H = np.reshape(H, (1, 9))[0]

        optimize_solution = optim_fun(cost_fun, H, inliers)

        H = np.reshape(optimize_solution.x, (3, 3))

        H /= H[2, 2] + eps

    class Homography_sol():
        def __init__(self):
            self.H = None
            self.optim_solution = None
            self.inliers = None
            self.outliers = None
            self.kp1 = None
            self.kp2 = None
            self.desc1 = None
            self.desc2 = None

        def _set_solution(self, H, optim_solution, inliers, outliers, kp1, kp2, desc1, desc2):
            self.H = H
            self.optim_solution = optim_solution
            self.inliers = inliers
            self.outliers = outliers
            self.kp1 = kp1
            self.kp2 = kp2
            self.desc1 = desc1
            self.desc2 = desc2

    solution = Homography_sol()
    if optimizer is None:
        solution._set_solution(H, None, inliers, outliers, kp1, kp2, desc1, desc2)
    else:
        solution._set_solution(H, optimize_solution, inliers, outliers, kp1, kp2, desc1, desc2
    )

    return solution
```

6.3 LM_optim.py

```
import numpy as np
from scipy.optimize import approx_fprime

from numpy.linalg import norm, solve

def LM_optim(fun:callable, x0:np.ndarray, tau:float=1e-3, eps1:float=1e-15, eps2:float=1e-15, eps3:float=1e-15, max_iter:int=100, args=(), kwargs={}):

    """
    Function to implement Levenberg-Marquardt optimization algorithm. Implementation follows pseudocode in https://users.ics.forth.gr/~lourakis/levmar/levmar.pdf.
    Jacobian is estimated using a 2-pt finite difference method.

    Args:
        fun (callable): Callable function which computes the residuals. Function signature and usage: ``fun(x, *args, **kwargs)``. Minimization proceeds w.r.t first argument.
        x0 (np.ndarray): Initial guess for the independent variables.
        tau (float): Scaling factor for the damping parameter mu. ``0 < tau <= 1``. Defaults to ``1e-3``.
        eps1 (float): Termination threshold for the inf-norm of the gradient. Termination condition: ``norm(grad, inf) <= eps1``. Defaults to ``1e-15``.
        eps2 (float): Relative termination threshold for delta. Termination condition: ``norm(delta) <= eps2 * norm(x)``. Defaults to ``1e-15``.
        eps3 (float): Termination threshold for the square of the norm of residuals. ``norm(fun(x))**2 <= eps3``. Defaults to ``1e-15``.
        max_iter (int): Maximum number of iterations.

    Returns:
        solution (OptimizeSolution): The solution to the optimization problem. ``OptimizeSolution`` has the following fields -
        1. x (np.ndarray): Solution to the optimization problem.
        2. errors (list): Values of the cost function (C) at every iteration. C = norm(fun(x, *args, **kwargs), 2)^2
        3. mu (float): Final value fo the damping parameter used in the algorithm.
        4. jac (np.ndarray): Value of the Jacobian at the end of optimization.
        5. hess (np.ndarray): Value of the Hessian at the end of optimization.
        6. n_iter (int): Number of iteration taken to reach termination conditions.
    """

    print("---------------------------- Starting LM optimization... ---------------------------- \n")

    m = len(x0)
    errors = []

    n_iter = 0
    nu = 2
    x = x0

    residual = fun(x, *args, **kwargs)

    jac = -approx_fprime(x, fun, 1e-8, *args) # '-' sign because taking jacobian of residual vector.

    hess = jac.T @ jac
    grad = jac.T @ residual

    stop = (norm(grad, np.inf) <= eps1)
    mu = tau * np.max(np.diag(hess))

    while (not stop) and (n_iter < max_iter):
        n_iter += 1
        errors.append(norm(residual, 2)**2)

        delta = solve(hess + mu*np.eye(m), grad)

        if (norm(delta, 2) <= eps2*norm(x, 2)):
            stop = True

        else:
            x_new = x + delta
            rho = (norm(residual, 2)**2 - norm(fun(x_new, *args, **kwargs), 2)**2) / (delta.T @ (mu*delta + grad))

```

```
            if rho > 0:
                x = x_new

                residual = fun(x, *args, **kwargs)
                jac = -approx_fprime(x, fun, 1e-8, *args)

                hess = jac.T @ jac
                grad = jac.T @ residual

                stop = (norm(grad, np.inf) <= eps1) or (norm(residual, 2)**2 <= eps3)
                mu *= max(1/3, 1-(2*rho - 1)**3)
                nu = 2

            else:
                mu *= nu
                nu *= 2
    else:
        if stop:
            print(f"Stopped on reaching termination conditions in {n_iter} iteration(s). \n")
            print("
    ----------------------------------------------------------------------------------------")
        else:
            print("Stopped on reaching maximum iteration(s). \n")
            print("
    ----------------------------------------------------------------------------------------")

    class OptimizeSolution:
        def __init__(self):
            self.x = None
            self.errors = None
            self.mu = None
            self.jac = None
            self.hess = None
            self.n_iter = None

        def set_solution(self, x, errors, mu, jac, hess, n_iter):
            self.x = x
            self.errors = errors
            self.mu = mu
            self.jac = jac
            self.hess = hess
            self.n_iter = n_iter


    solution = OptimizeSolution()
    solution.set_solution(x, errors, mu, jac, hess, n_iter)

    return solution
```

6.4 panorama.py

```python
import cv2 as cv
import numpy as np

from homography_estimation import auto_estimate_homography

eps = np.finfo(float).eps

def generate_pano(image_filenames:list, matcher:str="SP_SG", optimizer:str="scipy_lm", interp:str="bilinear"):

    """
    Function to create a panorama given a set of images. The middle image is placed at the center of the canvas and then homographies are found relative to the middle image.

    Args:
        image_filenames (list): List that contains the paths to all images to create a panorama.
        matcher (str): Select which feature detection and matching algorithm to use. If ``matcher`` is "sift", correspondence are found using Scale-Invariant Feature Transform (SIFT) algorithm. If ``matcher`` is "SP_SG", correspondence are found using SuperPoint+SuperGlue deep learning pre-trained network. Defaults to "SP_SG". Check ``superglue_wrapper.py`` for more information to run the SuperPoint+SuperGlue pre-trained network.
        optimizer (str): Select if homography should be refined using the LM algorithm. If ``optimizer`` is ``None``, the homography is not refined. If ``optimizer`` is "scipy_lm", the homography is refined using the in-built SciPy implementation of the LM algorithm. If ``optimizer`` is "lm", the homography is refined using my implementation of the LM algorithm.
        interp (str): Select which interpolation strategy to use when applying the homogrpahy. If ``interp`` is "n_neighbors", uses nearest neighbors interpolation. If ``interp`` is "bilinear", uses bi-linear interpolation. Defaults to "bilinear".

    Returns:
        canvas (np.ndarray): The canvas that contains the panorama.
    """

    H_all = []
    n_images = len(image_filenames)

    for i in range(0, n_images-1):

        img1 = image_filenames[i]
        img2 = image_filenames[i+1]
        H = auto_estimate_homography(img1, img2, matcher, optimizer).H
        H_all.append(H)

    ref_idx = ((n_images+1)//2) - 1

    H_wrt_ref_all = [np.eye(3, dtype=np.float64)]*(len(H_all)+1)

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(ref_idx+1, n_images):
        H_wrt_ref = H_wrt_ref @ np.linalg.inv(H_all[i-1])
        H_wrt_ref_all[i] = H_wrt_ref

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(ref_idx-1, -1, -1):
        H_wrt_ref = H_wrt_ref @ H_all[i]
        H_wrt_ref_all[i] = H_wrt_ref

    image_all = [cv.imread(image_filenames[i]) for i in range(n_images)]

    temp = [find_projected_corner_h(image_all[i], H_wrt_ref_all[i]) for i in range(n_images)]

    canvas_h = np.max(temp)

    canvas_w = np.sum([image_all[i].shape[1] for i in range(n_images)])

    canvas = np.zeros((canvas_h, canvas_w, 3), dtype=np.uint8)

    tx = (canvas_w-image_all[ref_idx].shape[1]) // 2

    ty = (canvas_h-image_all[ref_idx].shape[0]) // 2

```

```
61
62      H_tr = np.array([1, 0, tx, 0, 1, ty, 0, 0, 1])
63      H_tr = np.reshape(H_tr, (3, 3))
64
65      for i in range(n_images):
66          H_wrt_ref_all[i] = H_tr @ H_wrt_ref_all[i]
67
68      for i in range(0, n_images):
69          canvas = apply_homography(image_all[i], canvas, H_wrt_ref_all[i], interp)
70
71          ## If want to view the panorama after every iteration
72
73          # cv.namedWindow("canvas", cv.WINDOW_NORMAL)
74          # cv.resizeWindow("canvas", 600, 600)
75          # cv.imshow("canvas", canvas)
76          # cv.waitKey(0)
77          # cv.destroyAllWindows()
78
79      canvas = trim_zeros(canvas)
80      return canvas
81
82  def _bounds_per_dimesion(arr:np.ndarray):
83      return map(lambda e: range(e.min(), e.max()+1), np.where(arr != 0))
84
85  def trim_zeros(arr:np.ndarray):
86      """
87      Function to trim zeros from multi-dimensional array.
88
89      Args:
90          arr (np.ndarray): array to trim zeros along edges.
91
92      Returns:
93          (nd.array): array with trimmed zeros along edges.
94      """
95      return arr[np.ix_(*_bounds_per_dimesion(arr))]
96
97  def find_projected_corner_h(img, H):
98          h, w = img.shape[:2]
99
100         source_corners = np.array([(0, 0, 1), (w - 1, 0, 1), (w - 1, h - 1, 1), (0, h - 1, 1)
    ]).T
101         dest_corners = H @ source_corners
102
103         dest_corners /= (dest_corners[2] + eps)
104
105         min_y = np.min(dest_corners[1, :])
106         max_y = np.max(dest_corners[1, :])
107
108         world_h = np.ceil(max_y - min_y).astype(np.int32)
109
110         return (world_h)
111
112 def _n_neighbors(f:np.ndarray, source_coords:np.ndarray, dest_coords:np.ndarray, canvas:np.
    ndarray):
113
114     """
115     Function to project using nearest neighbors interpolation.
116
117     Args:
118         f (np.ndarray): Image to be projected or source.
119         source_coords (np.ndarray): Source coordinates in HC.
120         dest_coord (np.ndarray): Destination coordinates in HC.
121         canvas (np.ndarray): Canvas where the image will be projected or destination.
122
123     Returns:
124         (np.ndarray): canvas with the projected image.
125     """
126
127     source_x = source_coords[0] / (source_coords[2] + eps)
128     source_y = source_coords[1] / (source_coords[2] + eps)
129
130     dest_x = dest_coords[0]
131     dest_y = dest_coords[1]
132
133     h, w = f.shape[:2]
134     valid_mask = (source_x >= 0) & (source_x < w) & (source_y >= 0) & (source_y < h)
```

```
    source_x_valid = np.floor(source_x[valid_mask]).astype(np.int32)
    source_y_valid = np.floor(source_y[valid_mask]).astype(np.int32)

    dest_x_valid = dest_x[valid_mask].astype(np.int32)
    dest_y_valid = dest_y[valid_mask].astype(np.int32)

    canvas[dest_y_valid, dest_x_valid] = f[source_y_valid, source_x_valid]

    return canvas

def _bilinear(f:np.ndarray, source_coords:np.ndarray, dest_coords:np.ndarray, canvas:np.ndarray):

    """
    Function to project using bi-linear interpolation.

    Args:
        f (np.ndarray): Image to be projected or source.
        source_coords (np.ndarray): Source coordinates in HC.
        dest_coord (np.ndarray): Destination coordinates in HC.
        canvas (np.ndarray): Canvas where the image will be projected or destination.

    Returns:
        (np.ndarray): canvas with the projected image.
    """

    source_x = source_coords[0] / (source_coords[2] + eps)
    source_y = source_coords[1] / (source_coords[2] + eps)

    dest_x = dest_coords[0]
    dest_y = dest_coords[1]

    h, w = f.shape[:2]
    if f.ndim == 3:
        c = f.shape[2]

    valid_mask = (source_x >= 0) & (source_x < w-1) & (source_y >= 0) & (source_y < h-1)

    source_x_valid = source_x[valid_mask]
    source_y_valid = source_y[valid_mask]

    x1, y1 = np.floor(source_x_valid).astype(np.int32), np.floor(source_y_valid).astype(np.int32)
    x_diff, y_diff = source_x_valid - x1, source_y_valid - y1
    x2, y2 = x1 + 1, y1 + 1

    dest_x_valid = dest_x[valid_mask].astype(np.int32)
    dest_y_valid = dest_y[valid_mask].astype(np.int32)

    if f.ndim == 2:
        f11 = f[y1, x1]
        f12 = f[y1, x2]
        f21 = f[y2, x1]
        f22 = f[y2, x2]

        canvas[dest_y_valid, dest_x_valid] = (f11 * (1 - x_diff) * (1 - y_diff) +
                                              f12 * (x_diff) * (1 - y_diff) +
                                              f21 * (1 - x_diff) * (y_diff) +
                                              f22 * (x_diff) * (y_diff))

    else:
        for i in range(c):
            f11 = f[y1, x1, i]
            f12 = f[y1, x2, i]
            f21 = f[y2, x1, i]
            f22 = f[y2, x2, i]

            canvas[dest_y_valid, dest_x_valid, i] = (f11 * (1 - x_diff) * (1 - y_diff) +
                                                     f12 * (x_diff) * (1 - y_diff) +
                                                     f21 * (1 - x_diff) * (y_diff) +
                                                     f22 * (x_diff) * (y_diff))

    return canvas

def apply_homography(img:np.ndarray, canvas:np.ndarray, H:np.ndarray, interp:str="bilinear"):
```

```
    """
    Function to apply the homogrpahy H to img and place it on the canvas.

    Args:
        img (np.ndarray): Image to which the homography will be applied to.
        canvas (np.ndarray): Canvas to place the projected image on.
        H (np.ndarray): The homography to apply.
        interp (str): Select which interpolation strategy to use when applying the homogrpahy.
     If ``interp`` is "n_neighbors", uses nearest neighbors interpolation. If ``interp`` is "
    bilinear", uses bi-linear interpolation. Defaults to "bilinear".

    Returns:
        (np.ndarray): Canvas after projecting the image.
    """

    val_interp = {"n_neighbors": _n_neighbors, "bilinear": _bilinear}

    try:
        interp_fun = val_interp[interp]
    except KeyError:
        raise ValueError(f"Invalid interpolation method. Choose from {list(val_interp.keys())}
    ")


    world_h, world_w = canvas.shape[:2]

    x = np.arange(0, world_w, 1)
    y = np.arange(0, world_h, 1)
    x, y = np.meshgrid(x, y)
    dest_coords = np.vstack((x.ravel(), y.ravel()))
    dest_coords = np.vstack((dest_coords, np.ones(dest_coords.shape[1])))

    source_coords = np.linalg.inv(H) @ dest_coords

    res = interp_fun(img, source_coords, dest_coords, canvas)

    return res
```

6.5 superglue_wrapper.py

```python
import torch
import numpy as np


from SuperGluePretrainedNetwork.models.matching import Matching
from SuperGluePretrainedNetwork.models.utils import read_image

class SuperGlue(object):
    def __init__(self):
        super(SuperGlue, self).__init__()
        self.matcher = None
        self.device = None
        self.config = None
        self.resize = None

    @classmethod
    def create(cls,
               force_gpu=False,
               nms_radius=4,
               keypoint_threshold=0.005,
               max_keypoints=-1,
               superglue_wts='indoor',
               sinkhorn_iterations=20,
               match_threshold=0.2,
               resize=[640, 480]):

        det = cls()
        det.set_device_as_gpu(force_gpu=force_gpu)
        det.set_config(nms_radius,
                       keypoint_threshold,
                       max_keypoints,
                       superglue_wts,
                       sinkhorn_iterations,
                       match_threshold)

        det.matcher = Matching(det.config).eval().to(det.device)
        det.resize = resize
        return det

    def set_device_as_gpu(self, force_gpu=True):
        if torch.cuda.is_available() and force_gpu:
            device = 'cuda'

        elif torch.backends.mps.is_available() and torch.backends.mps.is_built():
            device = 'mps'

        else:
            device = 'cpu'

        self.device = device

    def set_config(self, nms_radius,
                   keypoint_threshold,
                   max_keypoints,
                   superglue_wts,
                   sinkhorn_iterations,
                   match_threshold):
        self.config = {
            'superpoint': {
                'nms_radius': nms_radius,
                'keypoint_threshold': keypoint_threshold,
                'max_keypoints': max_keypoints
            },
            'superglue': {
                'weights': superglue_wts,
                'sinkhorn_iterations': sinkhorn_iterations,
                'match_threshold': match_threshold,
            }
        }

    @torch.no_grad()
    def detect_and_compute(self, img):
        inp, scales = self.read_img(img)

```

```
        data = self.matcher.superpoint({'image': inp})
        kp = data['keypoints'][0].to('cpu').numpy() * np.array(scales)
        desc = data['descriptors'][0].to('cpu').numpy()

        return kp, desc

    @torch.no_grad()
    def match(self, img1, img2):

        inp0, scales0 = self.read_img(img1)
        inp1, scales1 = self.read_img(img2)

        pred = self.matcher({'image0': inp0, 'image1': inp1})
        pred = {k: v[0].to('cpu').numpy() for k, v in pred.items()}
        kpts0, kpts1 = pred['keypoints0'], pred['keypoints1']
        matches, conf = pred['matches0'], pred['matching_scores0']

        valid = matches > -1
        mkpts0 = kpts0[valid]
        mkpts1 = kpts1[matches[valid]]

        mkpts0 = np.array(mkpts0) * np.array(scales0)
        mkpts1 = np.array(mkpts1) * np.array(scales1)

        return mkpts0, mkpts1


    def read_img(self, img_path):
        _, inp, scale = read_image(img_path, self.device, self.resize, 0, False)
        return inp, scale
```