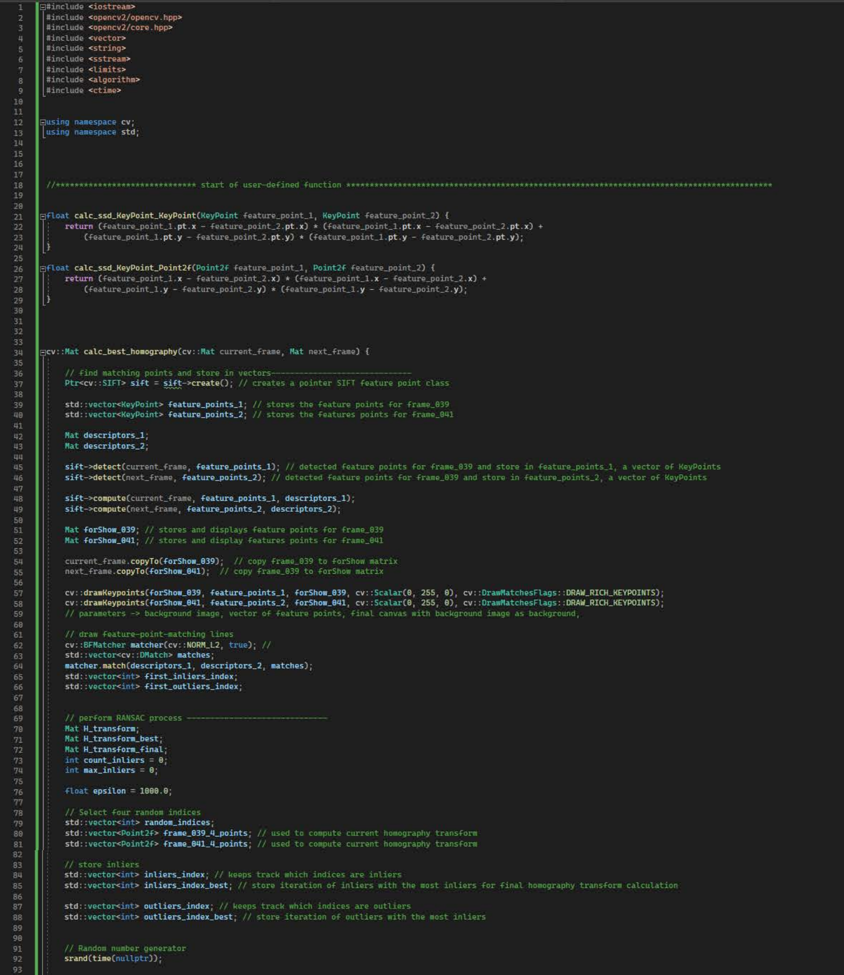
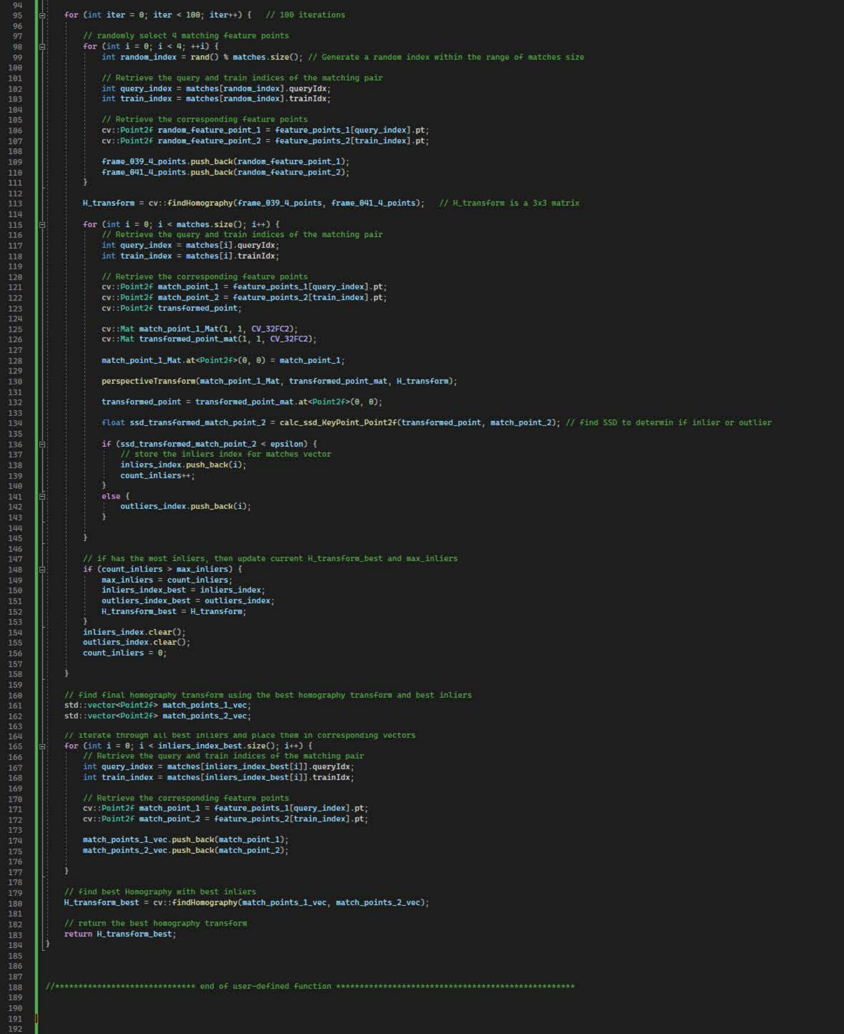
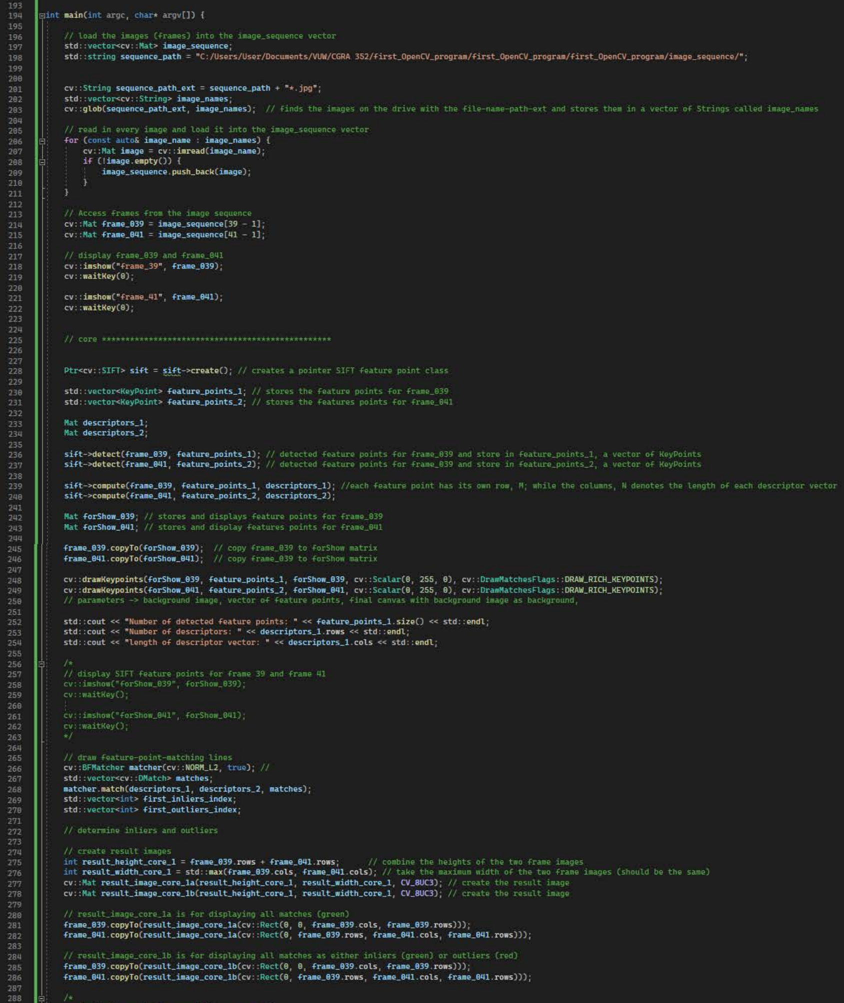


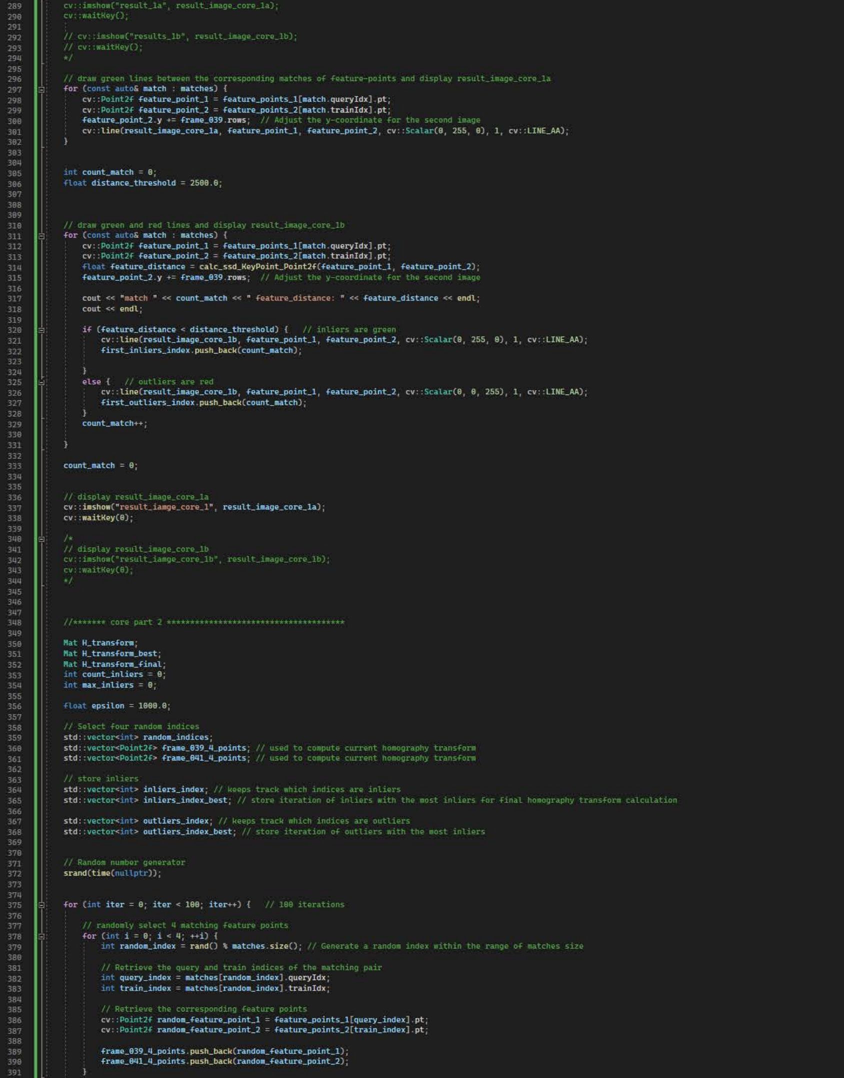
```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/core.hpp>
#include <vector>
#include <string>
#include <sstream>
#include <limits>
#include <algorithm>
#include <ctime>


using namespace cv;
using namespace std;




//****************************** start of user-defined function *******************************************************************************************


float calc_ssd_KeyPoint_KeyPoint(KeyPoint feature_point_1, KeyPoint feature_point_2) {
    return (feature_point_1.pt.x - feature_point_2.pt.x) * (feature_point_1.pt.x - feature_point_2.pt.x) +
        (feature_point_1.pt.y - feature_point_2.pt.y) * (feature_point_1.pt.y - feature_point_2.pt.y);
}

float calc_ssd_KeyPoint_Point2f(Point2f feature_point_1, Point2f feature_point_2) {
    return (feature_point_1.x - feature_point_2.x) * (feature_point_1.x - feature_point_2.x) +
        (feature_point_1.y - feature_point_2.y) * (feature_point_1.y - feature_point_2.y);
}




cv::Mat calc_best_homography(cv::Mat current_frame, Mat next_frame) {

    // find matching points and store in vectors-------------------------------
    Ptr<cv::SIFT> sift = sift->create(); // creates a pointer SIFT feature point class

    std::vector<KeyPoint> feature_points_1; // stores the feature points for frame_039
    std::vector<KeyPoint> feature_points_2; // stores the features points for frame_041

    Mat descriptors_1;
    Mat descriptors_2;

    sift->detect(current_frame, feature_points_1); // detected feature points for frame_039 and store in feature_points_1, a vector of KeyPoints
    sift->detect(next_frame, feature_points_2); // detected feature points for frame_039 and store in feature_points_2, a vector of KeyPoints

    sift->compute(current_frame, feature_points_1, descriptors_1);
    sift->compute(next_frame, feature_points_2, descriptors_2);

    Mat forShow_039; // stores and displays feature points for frame_039
    Mat forShow_041; // stores and display features points for frame_041

    current_frame.copyTo(forShow_039);  // copy frame_039 to forShow matrix
    next_frame.copyTo(forShow_041);  // copy frame_039 to forShow matrix

    cv::drawKeypoints(forShow_039, feature_points_1, forShow_039, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::drawKeypoints(forShow_041, feature_points_2, forShow_041, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    // parameters -> background image, vector of feature points, final canvas with background image as background,

    // draw feature-point-matching lines
    cv::BFMatcher matcher(cv::NORM_L2, true); //
    std::vector<cv::DMatch> matches;
    matcher.match(descriptors_1, descriptors_2, matches);
    std::vector<int> first_inliers_index;
    std::vector<int> first_outliers_index;


    // perform RANSAC process -------------------------------
    Mat H_transform;
    Mat H_transform_best;
    Mat H_transform_final;
    int count_inliers = 0;
    int max_inliers = 0;

    float epsilon = 1000.0;

    // Select four random indices
    std::vector<int> random_indices;
    std::vector<Point2f> frame_039_4_points; // used to compute current homography transform
    std::vector<Point2f> frame_041_4_points; // used to compute current homography transform

    // store inliers
    std::vector<int> inliers_index; // keeps track which indices are inliers
    std::vector<int> inliers_index_best; // store iteration of inliers with the most inliers for final homography transform calculation

    std::vector<int> outliers_index; // keeps track which indices are outliers
    std::vector<int> outliers_index_best; // store iteration of outliers with the most inliers


    // Random number generator
    srand(time(nullptr));

```



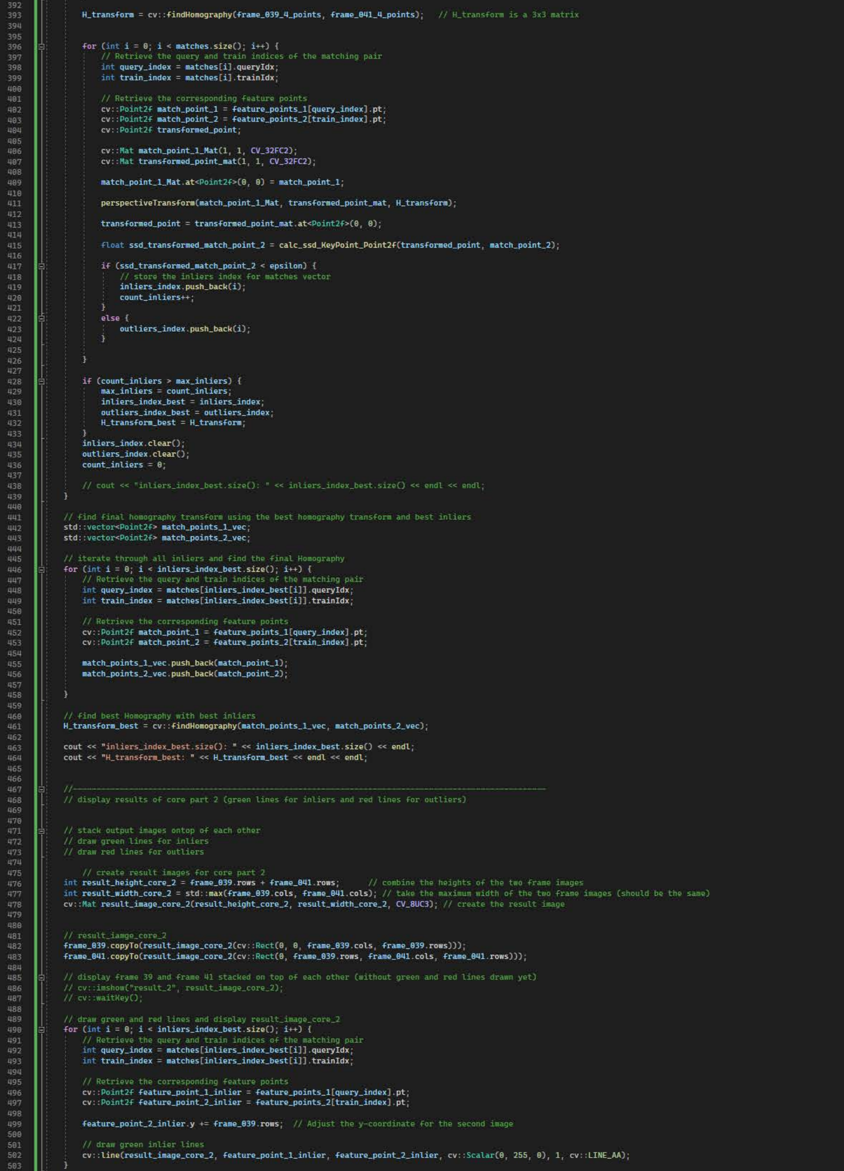
```
    for (int iter = 0; iter < 100; iter++) {   // 100 iterations

        // randomly select 4 matching feature points
        for (int i = 0; i < 4; ++i) {
            int random_index = rand() % matches.size(); // Generate a random index within the range of matches size

            // Retrieve the query and train indices of the matching pair
            int query_index = matches[random_index].queryIdx;
            int train_index = matches[random_index].trainIdx;

            // Retrieve the corresponding feature points
            cv::Point2f random_feature_point_1 = feature_points_1[query_index].pt;
            cv::Point2f random_feature_point_2 = feature_points_2[train_index].pt;

            frame_039_4_points.push_back(random_feature_point_1);
            frame_041_4_points.push_back(random_feature_point_2);
        }

        H_transform = cv::findHomography(frame_039_4_points, frame_041_4_points);   // H_transform is a 3x3 matrix

        for (int i = 0; i < matches.size(); i++) {
            // Retrieve the query and train indices of the matching pair
            int query_index = matches[i].queryIdx;
            int train_index = matches[i].trainIdx;

            // Retrieve the corresponding feature points
            cv::Point2f match_point_1 = feature_points_1[query_index].pt;
            cv::Point2f match_point_2 = feature_points_2[train_index].pt;
            cv::Point2f transformed_point;

            cv::Mat match_point_1_Mat(1, 1, CV_32FC2);
            cv::Mat transformed_point_mat(1, 1, CV_32FC2);

            match_point_1_Mat.at<Point2f>(0, 0) = match_point_1;

            perspectiveTransform(match_point_1_Mat, transformed_point_mat, H_transform);

            transformed_point = transformed_point_mat.at<Point2f>(0, 0);

            float ssd_transformed_match_point_2 = calc_ssd_KeyPoint_Point2f(transformed_point, match_point_2); // find SSD to determin if inlier or outlier

            if (ssd_transformed_match_point_2 < epsilon) {
                // store the inliers index for matches vector
                inliers_index.push_back(i);
                count_inliers++;
            }
            else {
                outliers_index.push_back(i);
            }

        }

        // if has the most inliers, then update current H_transform_best and max_inliers
        if (count_inliers > max_inliers) {
            max_inliers = count_inliers;
            inliers_index_best = inliers_index;
            outliers_index_best = outliers_index;
            H_transform_best = H_transform;
        }
        inliers_index.clear();
        outliers_index.clear();
        count_inliers = 0;

    }

    // find final homography transform using the best homography transform and best inliers
    std::vector<Point2f> match_points_1_vec;
    std::vector<Point2f> match_points_2_vec;

    // iterate through all best inliers and place them in corresponding vectors
    for (int i = 0; i < inliers_index_best.size(); i++) {
        // Retrieve the query and train indices of the matching pair
        int query_index = matches[inliers_index_best[i]].queryIdx;
        int train_index = matches[inliers_index_best[i]].trainIdx;

        // Retrieve the corresponding feature points
        cv::Point2f match_point_1 = feature_points_1[query_index].pt;
        cv::Point2f match_point_2 = feature_points_2[train_index].pt;

        match_points_1_vec.push_back(match_point_1);
        match_points_2_vec.push_back(match_point_2);

    }

    // find best Homography with best inliers
    H_transform_best = cv::findHomography(match_points_1_vec, match_points_2_vec);

    // return the best homography transform
    return H_transform_best;
}


//****************************** end of user-defined function ***************************************************
```



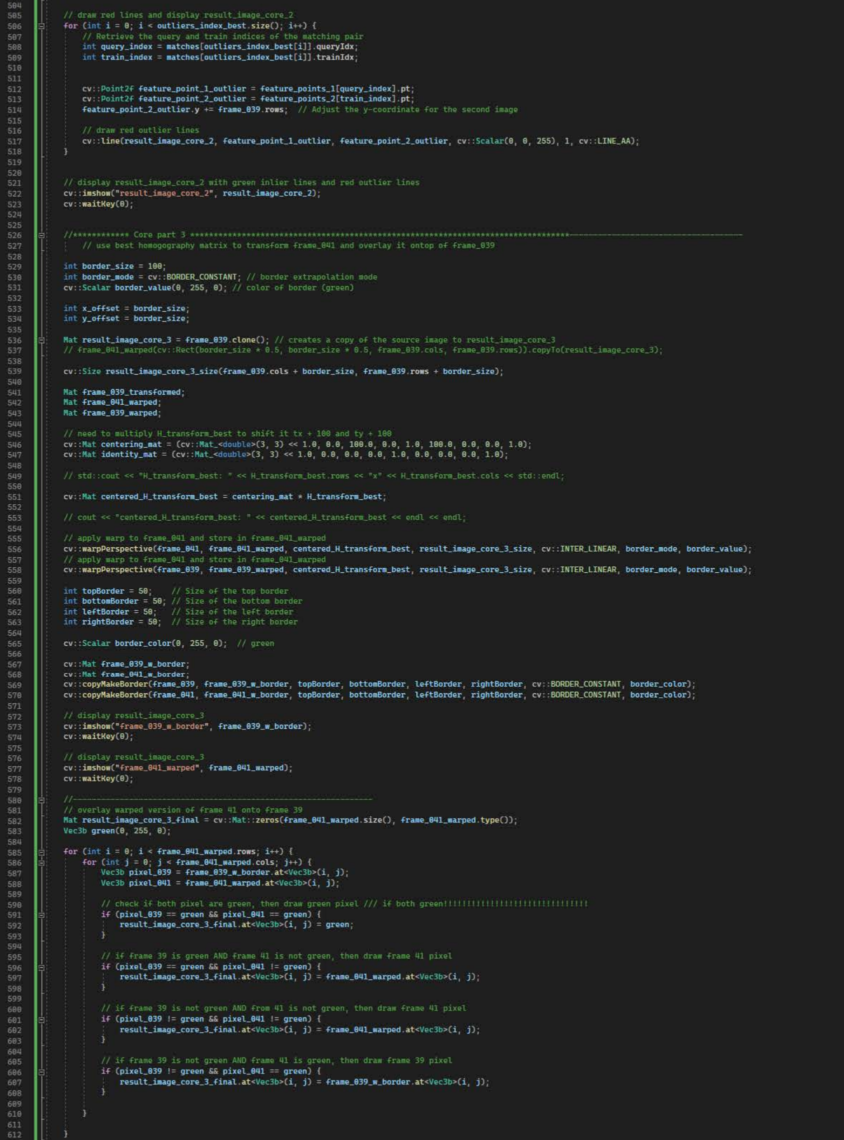
```cpp
int main(int argc, char* argv[]) {

    // load the images (frames) into the image_sequence vector
    std::vector<cv::Mat> image_sequence;
    std::string sequence_path = "C:/Users/User/Documents/VUW/CGRA 352/first_OpenCV_program/first_OpenCV_program/first_OpenCV_program/image_sequence/";


    cv::String sequence_path_ext = sequence_path + "*.jpg";
    std::vector<cv::String> image_names;
    cv::glob(sequence_path_ext, image_names);  // finds the images on the drive with the file-name-path-ext and stores them in a vector of Strings called image_names

    // read in every image and load it into the image_sequence vector
    for (const auto& image_name : image_names) {
        cv::Mat image = cv::imread(image_name);
        if (!image.empty()) {
            image_sequence.push_back(image);
        }
    }

    // Access frames from the image sequence
    cv::Mat frame_039 = image_sequence[39 - 1];
    cv::Mat frame_041 = image_sequence[41 - 1];

    // display frame_039 and frame_041
    cv::imshow("frame_39", frame_039);
    cv::waitKey(0);

    cv::imshow("frame_41", frame_041);
    cv::waitKey(0);


    // core *************************************************


    Ptr<cv::SIFT> sift = sift->create(); // creates a pointer SIFT feature point class

    std::vector<KeyPoint> feature_points_1; // stores the feature points for frame_039
    std::vector<KeyPoint> feature_points_2; // stores the features points for frame_041

    Mat descriptors_1;
    Mat descriptors_2;

    sift->detect(frame_039, feature_points_1); // detected feature points for frame_039 and store in feature_points_1, a vector of KeyPoints
    sift->detect(frame_041, feature_points_2); // detected feature points for frame_039 and store in feature_points_2, a vector of KeyPoints

    sift->compute(frame_039, feature_points_1, descriptors_1); //each feature point has its own row, M; while the columns, N denotes the length of each descriptor vector
    sift->compute(frame_041, feature_points_2, descriptors_2);

    Mat forShow_039; // stores and displays feature points for frame_039
    Mat forShow_041; // stores and display features points for frame_041

    frame_039.copyTo(forShow_039);  // copy frame_039 to forShow matrix
    frame_041.copyTo(forShow_041);  // copy frame_039 to forShow matrix

    cv::drawKeypoints(forShow_039, feature_points_1, forShow_039, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::drawKeypoints(forShow_041, feature_points_2, forShow_041, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    // parameters -> background image, vector of feature points, final canvas with background image as background,

    std::cout << "Number of detected feature points: " << feature_points_1.size() << std::endl;
    std::cout << "Number of descriptors: " << descriptors_1.rows << std::endl;
    std::cout << "length of descriptor vector: " << descriptors_1.cols << std::endl;

    /*
    // display SIFT feature points for frame 39 and frame 41
    cv::imshow("forShow_039", forShow_039);
    cv::waitKey();

    cv::imshow("forShow_041", forShow_041);
    cv::waitKey();
    */

    // draw feature-point-matching lines
    cv::BFMatcher matcher(cv::NORM_L2, true); //
    std::vector<cv::DMatch> matches;
    matcher.match(descriptors_1, descriptors_2, matches);
    std::vector<int> first_inliers_index;
    std::vector<int> first_outliers_index;

    // determine inliers and outliers

    // create result images
    int result_height_core_1 = frame_039.rows + frame_041.rows;      // combine the heights of the two frame images
    int result_width_core_1 = std::max(frame_039.cols, frame_041.cols); // take the maximum width of the two frame images (should be the same)
    cv::Mat result_image_core_1a(result_height_core_1, result_width_core_1, CV_8UC3); // create the result image
    cv::Mat result_image_core_1b(result_height_core_1, result_width_core_1, CV_8UC3); // create the result image

    // result_image_core_1a is for displaying all matches (green)
    frame_039.copyTo(result_image_core_1a(cv::Rect(0, 0, frame_039.cols, frame_039.rows)));
    frame_041.copyTo(result_image_core_1a(cv::Rect(0, frame_039.rows, frame_041.cols, frame_041.rows)));

    // result_image_core_1b is for displaying all matches as either inliers (green) or outliers (red)
    frame_039.copyTo(result_image_core_1b(cv::Rect(0, 0, frame_039.cols, frame_039.rows)));
    frame_041.copyTo(result_image_core_1b(cv::Rect(0, frame_039.rows, frame_041.cols, frame_041.rows)));

    /*
```



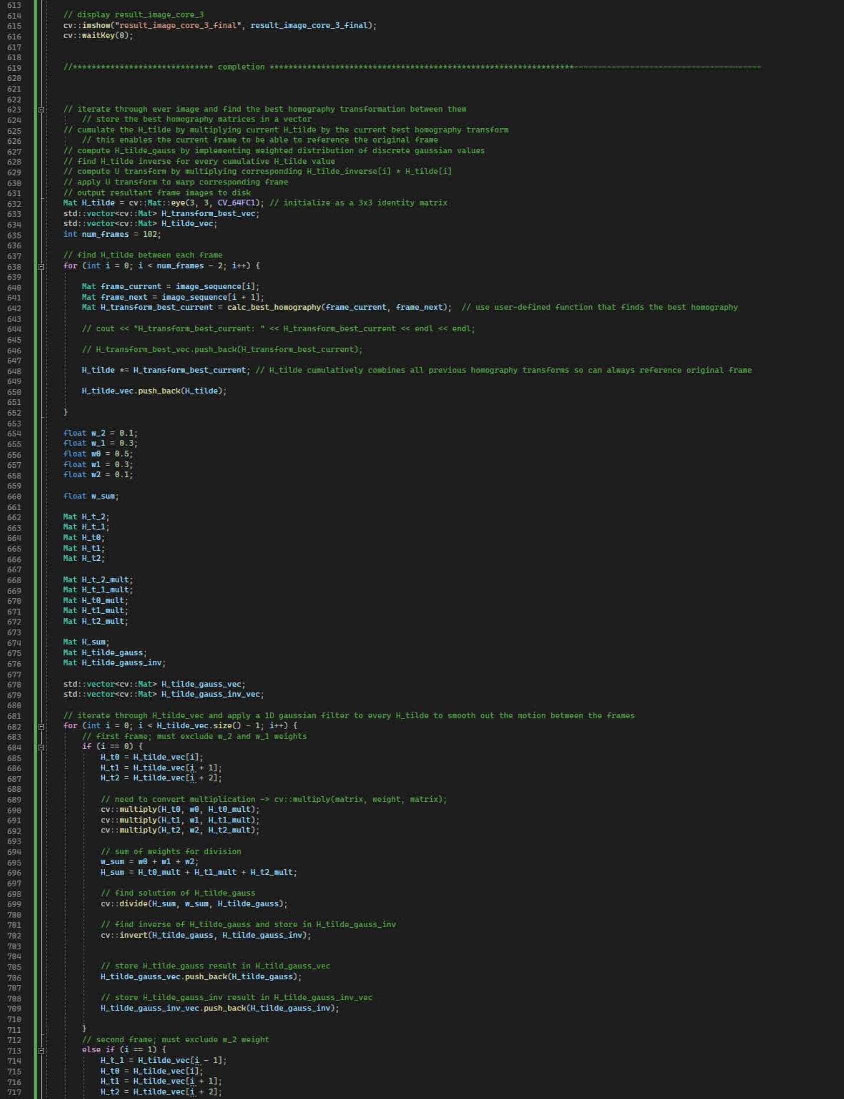
```cpp
    cv::imshow("result_1a", result_image_core_1a);
    cv::waitKey();

    // cv::imshow("results_1b", result_image_core_1b);
    // cv::waitKey();
    */

    // draw green lines between the corresponding matches of feature-points and display result_image_core_1a
    for (const auto& match : matches) {
        cv::Point2f feature_point_1 = feature_points_1[match.queryIdx].pt;
        cv::Point2f feature_point_2 = feature_points_2[match.trainIdx].pt;
        feature_point_2.y += frame_039.rows;  // Adjust the y-coordinate for the second image
        cv::line(result_image_core_1a, feature_point_1, feature_point_2, cv::Scalar(0, 255, 0), 1, cv::LINE_AA);
    }


    int count_match = 0;
    float distance_threshold = 2500.0;



    // draw green and red lines and display result_image_core_1b
    for (const auto& match : matches) {
        cv::Point2f feature_point_1 = feature_points_1[match.queryIdx].pt;
        cv::Point2f feature_point_2 = feature_points_2[match.trainIdx].pt;
        float feature_distance = calc_ssd_KeyPoint_Point2f(feature_point_1, feature_point_2);
        feature_point_2.y += frame_039.rows;  // Adjust the y-coordinate for the second image

        cout << "match " << count_match << " feature_distance: " << feature_distance << endl;
        cout << endl;

        if (feature_distance < distance_threshold) {   // inliers are green
            cv::line(result_image_core_1b, feature_point_1, feature_point_2, cv::Scalar(0, 255, 0), 1, cv::LINE_AA);
            first_inliers_index.push_back(count_match);

        }
        else {   // outliers are red
            cv::line(result_image_core_1b, feature_point_1, feature_point_2, cv::Scalar(0, 0, 255), 1, cv::LINE_AA);
            first_outliers_index.push_back(count_match);
        }
        count_match++;

    }

    count_match = 0;


    // display result_image_core_1a
    cv::imshow("result_iamge_core_1", result_image_core_1a);
    cv::waitKey(0);

    /*
    // display result_image_core_1b
    cv::imshow("result_iamge_core_1b", result_image_core_1b);
    cv::waitKey(0);
    */



    //******* core part 2 ****************************************

    Mat H_transform;
    Mat H_transform_best;
    Mat H_transform_final;
    int count_inliers = 0;
    int max_inliers = 0;

    float epsilon = 1000.0;

    // Select four random indices
    std::vector<int> random_indices;
    std::vector<Point2f> frame_039_4_points; // used to compute current homography transform
    std::vector<Point2f> frame_041_4_points; // used to compute current homography transform

    // store inliers
    std::vector<int> inliers_index; // keeps track which indices are inliers
    std::vector<int> inliers_index_best; // store iteration of inliers with the most inliers for final homography transform calculation

    std::vector<int> outliers_index; // keeps track which indices are outliers
    std::vector<int> outliers_index_best; // store iteration of outliers with the most inliers


    // Random number generator
    srand(time(nullptr));


    for (int iter = 0; iter < 100; iter++) {   // 100 iterations

        // randomly select 4 matching feature points
        for (int i = 0; i < 4; ++i) {
            int random_index = rand() % matches.size(); // Generate a random index within the range of matches size

            // Retrieve the query and train indices of the matching pair
            int query_index = matches[random_index].queryIdx;
            int train_index = matches[random_index].trainIdx;

            // Retrieve the corresponding feature points
            cv::Point2f random_feature_point_1 = feature_points_1[query_index].pt;
            cv::Point2f random_feature_point_2 = feature_points_2[train_index].pt;

            frame_039_4_points.push_back(random_feature_point_1);
            frame_041_4_points.push_back(random_feature_point_2);
        }
```



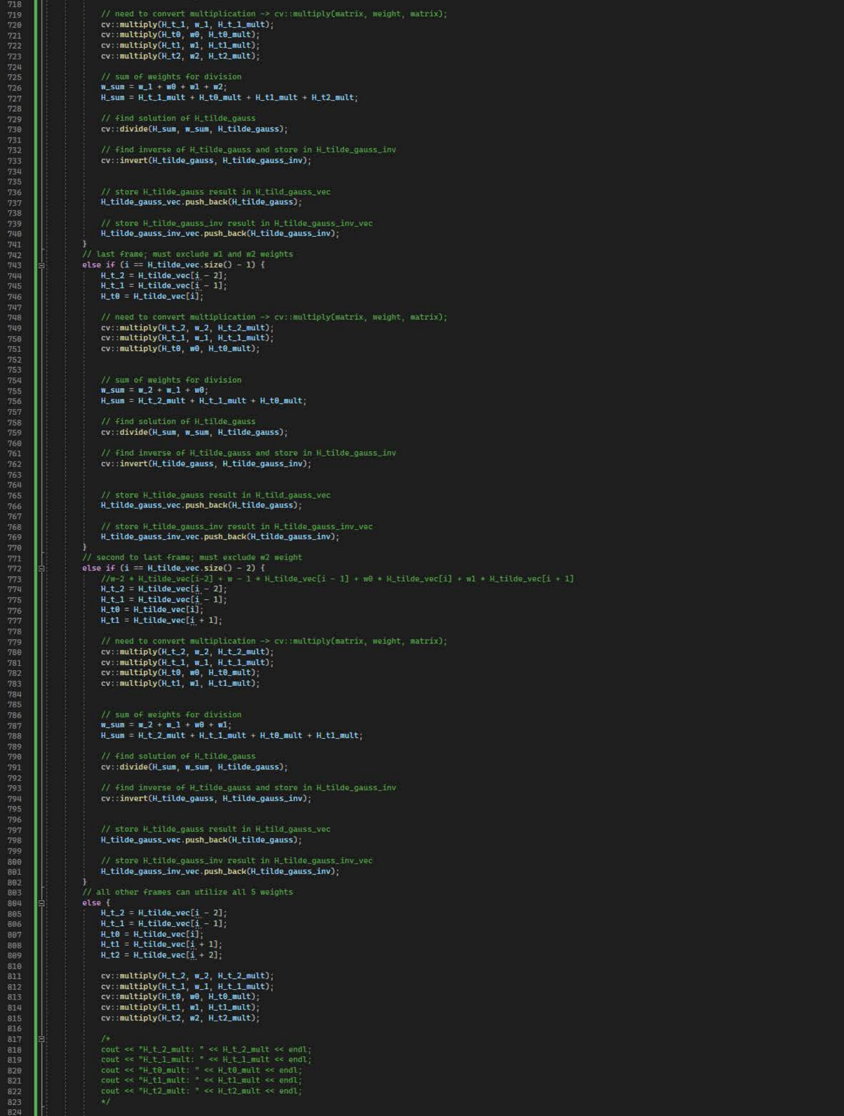
```

        H_transform = cv::findHomography(frame_039_4_points, frame_041_4_points);   // H_transform is a 3x3 matrix


        for (int i = 0; i < matches.size(); i++) {
            // Retrieve the query and train indices of the matching pair
            int query_index = matches[i].queryIdx;
            int train_index = matches[i].trainIdx;

            // Retrieve the corresponding feature points
            cv::Point2f match_point_1 = feature_points_1[query_index].pt;
            cv::Point2f match_point_2 = feature_points_2[train_index].pt;
            cv::Point2f transformed_point;

            cv::Mat match_point_1_Mat(1, 1, CV_32FC2);
            cv::Mat transformed_point_mat(1, 1, CV_32FC2);

            match_point_1_Mat.at<Point2f>(0, 0) = match_point_1;

            perspectiveTransform(match_point_1_Mat, transformed_point_mat, H_transform);

            transformed_point = transformed_point_mat.at<Point2f>(0, 0);

            float ssd_transformed_match_point_2 = calc_ssd_KeyPoint_Point2f(transformed_point, match_point_2);

            if (ssd_transformed_match_point_2 < epsilon) {
                // store the inliers index for matches vector
                inliers_index.push_back(i);
                count_inliers++;
            }
            else {
                outliers_index.push_back(i);
            }

        }

        if (count_inliers > max_inliers) {
            max_inliers = count_inliers;
            inliers_index_best = inliers_index;
            outliers_index_best = outliers_index;
            H_transform_best = H_transform;
        }
        inliers_index.clear();
        outliers_index.clear();
        count_inliers = 0;

        // cout << "inliers_index_best.size(): " << inliers_index_best.size() << endl << endl;
    }

    // find final homography transform using the best homography transform and best inliers
    std::vector<Point2f> match_points_1_vec;
    std::vector<Point2f> match_points_2_vec;

    // iterate through all inliers and find the final Homography
    for (int i = 0; i < inliers_index_best.size(); i++) {
        // Retrieve the query and train indices of the matching pair
        int query_index = matches[inliers_index_best[i]].queryIdx;
        int train_index = matches[inliers_index_best[i]].trainIdx;

        // Retrieve the corresponding feature points
        cv::Point2f match_point_1 = feature_points_1[query_index].pt;
        cv::Point2f match_point_2 = feature_points_2[train_index].pt;

        match_points_1_vec.push_back(match_point_1);
        match_points_2_vec.push_back(match_point_2);

    }

    // find best Homography with best inliers
    H_transform_best = cv::findHomography(match_points_1_vec, match_points_2_vec);

    cout << "inliers_index_best.size(): " << inliers_index_best.size() << endl;
    cout << "H_transform_best: " << H_transform_best << endl << endl;


    //-----------------------------------------------------------------------------------------------------
    // display results of core part 2 (green lines for inliers and red lines for outliers)


    // stack output images ontop of each other
    // draw green lines for inliers
    // draw red lines for outliers

        // create result images for core part 2
    int result_height_core_2 = frame_039.rows + frame_041.rows;      // combine the heights of the two frame images
    int result_width_core_2 = std::max(frame_039.cols, frame_041.cols); // take the maximum width of the two frame images (should be the same)
    cv::Mat result_image_core_2(result_height_core_2, result_width_core_2, CV_8UC3); // create the result image


    // result_iamge_core_2
    frame_039.copyTo(result_image_core_2(cv::Rect(0, 0, frame_039.cols, frame_039.rows)));
    frame_041.copyTo(result_image_core_2(cv::Rect(0, frame_039.rows, frame_041.cols, frame_041.rows)));

    // display frame 39 and frame 41 stacked on top of each other (without green and red lines drawn yet)
    // cv::imshow("result_2", result_image_core_2);
    // cv::waitKey();

    // draw green and red lines and display result_image_core_2
    for (int i = 0; i < inliers_index_best.size(); i++) {
        // Retrieve the query and train indices of the matching pair
        int query_index = matches[inliers_index_best[i]].queryIdx;
        int train_index = matches[inliers_index_best[i]].trainIdx;

        // Retrieve the corresponding feature points
        cv::Point2f feature_point_1_inlier = feature_points_1[query_index].pt;
        cv::Point2f feature_point_2_inlier = feature_points_2[train_index].pt;

        feature_point_2_inlier.y += frame_039.rows;  // Adjust the y-coordinate for the second image

        // draw green inlier lines
        cv::line(result_image_core_2, feature_point_1_inlier, feature_point_2_inlier, cv::Scalar(0, 255, 0), 1, cv::LINE_AA);
    }
```



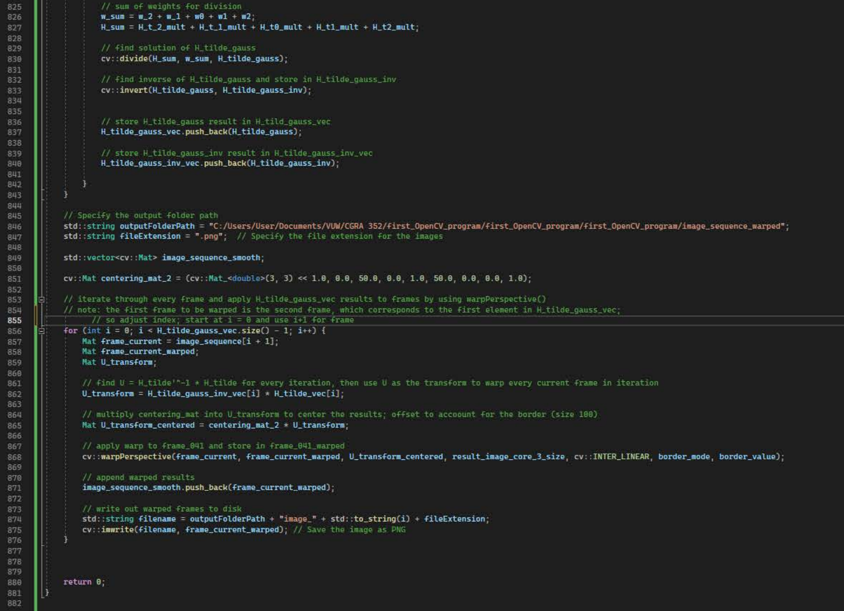
```cpp

    // draw red lines and display result_image_core_2
    for (int i = 0; i < outliers_index_best.size(); i++) {
        // Retrieve the query and train indices of the matching pair
        int query_index = matches[outliers_index_best[i]].queryIdx;
        int train_index = matches[outliers_index_best[i]].trainIdx;


        cv::Point2f feature_point_1_outlier = feature_points_1[query_index].pt;
        cv::Point2f feature_point_2_outlier = feature_points_2[train_index].pt;
        feature_point_2_outlier.y += frame_039.rows;  // Adjust the y-coordinate for the second image

        // draw red outlier lines
        cv::line(result_image_core_2, feature_point_1_outlier, feature_point_2_outlier, cv::Scalar(0, 0, 255), 1, cv::LINE_AA);
    }


    // display result_image_core_2 with green inlier lines and red outlier lines
    cv::imshow("result_image_core_2", result_image_core_2);
    cv::waitKey(0);


    //************ Core part 3 **********************************************************************************-------------------------------------
        // use best homogography matrix to transform frame_041 and overlay it ontop of frame_039

    int border_size = 100;
    int border_mode = cv::BORDER_CONSTANT; // border extrapolation mode
    cv::Scalar border_value(0, 255, 0); // color of border (green)

    int x_offset = border_size;
    int y_offset = border_size;

    Mat result_image_core_3 = frame_039.clone(); // creates a copy of the source image to result_image_core_3
    // frame_041_warped(cv::Rect(border_size * 0.5, border_size * 0.5, frame_039.cols, frame_039.rows)).copyTo(result_image_core_3);

    cv::Size result_image_core_3_size(frame_039.cols + border_size, frame_039.rows + border_size);

    Mat frame_039_transformed;
    Mat frame_041_warped;
    Mat frame_039_warped;

    // need to multiply H_transform_best to shift it tx + 100 and ty + 100
    cv::Mat centering_mat = (cv::Mat_<double>(3, 3) << 1.0, 0.0, 100.0, 0.0, 1.0, 100.0, 0.0, 0.0, 1.0);
    cv::Mat identity_mat = (cv::Mat_<double>(3, 3) << 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0);

    // std::cout << "H_transform_best: " << H_transform_best.rows << "x" << H_transform_best.cols << std::endl;

    cv::Mat centered_H_transform_best = centering_mat * H_transform_best;

    // cout << "centered_H_transform_best: " << centered_H_transform_best << endl << endl;

    // apply warp to frame_041 and store in frame_041_warped
    cv::warpPerspective(frame_041, frame_041_warped, centered_H_transform_best, result_image_core_3_size, cv::INTER_LINEAR, border_mode, border_value);
    // apply warp to frame_041 and store in frame_041_warped
    cv::warpPerspective(frame_039, frame_039_warped, centered_H_transform_best, result_image_core_3_size, cv::INTER_LINEAR, border_mode, border_value);

    int topBorder = 50;    // Size of the top border
    int bottomBorder = 50; // Size of the bottom border
    int leftBorder = 50;   // Size of the left border
    int rightBorder = 50;  // Size of the right border

    cv::Scalar border_color(0, 255, 0);  // green

    cv::Mat frame_039_w_border;
    cv::Mat frame_041_w_border;
    cv::copyMakeBorder(frame_039, frame_039_w_border, topBorder, bottomBorder, leftBorder, rightBorder, cv::BORDER_CONSTANT, border_color);
    cv::copyMakeBorder(frame_041, frame_041_w_border, topBorder, bottomBorder, leftBorder, rightBorder, cv::BORDER_CONSTANT, border_color);

    // display result_image_core_3
    cv::imshow("frame_039_w_border", frame_039_w_border);
    cv::waitKey(0);

    // display result_image_core_3
    cv::imshow("frame_041_warped", frame_041_warped);
    cv::waitKey(0);

    //----------------------------------------------------------------
    // overlay warped version of frame 41 onto frame 39
    Mat result_image_core_3_final = cv::Mat::zeros(frame_041_warped.size(), frame_041_warped.type());
    Vec3b green(0, 255, 0);

    for (int i = 0; i < frame_041_warped.rows; i++) {
        for (int j = 0; j < frame_041_warped.cols; j++) {
            Vec3b pixel_039 = frame_039_w_border.at<Vec3b>(i, j);
            Vec3b pixel_041 = frame_041_warped.at<Vec3b>(i, j);

            // check if both pixel are green, then draw green pixel /// if both green!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            if (pixel_039 == green && pixel_041 == green) {
                result_image_core_3_final.at<Vec3b>(i, j) = green;
            }

            // if frame 39 is green AND frame 41 is not green, then draw frame 41 pixel
            if (pixel_039 == green && pixel_041 != green) {
                result_image_core_3_final.at<Vec3b>(i, j) = frame_041_warped.at<Vec3b>(i, j);
            }

            // if frame 39 is not green AND from 41 is not green, then draw frame 41 pixel
            if (pixel_039 != green && pixel_041 != green) {
                result_image_core_3_final.at<Vec3b>(i, j) = frame_041_warped.at<Vec3b>(i, j);
            }

            // if frame 39 is not green AND frame 41 is green, then draw frame 39 pixel
            if (pixel_039 != green && pixel_041 == green) {
                result_image_core_3_final.at<Vec3b>(i, j) = frame_039_w_border.at<Vec3b>(i, j);
            }

        }

    }
```



```
    // display result_image_core_3
    cv::imshow("result_image_core_3_final", result_image_core_3_final);
    cv::waitKey(0);


    //****************************** completion *****************************************************************----------------------------------------



    // iterate through ever image and find the best homography transformation between them
        // store the best homography matrices in a vector
    // cumulate the H_tilde by multiplying current H_tilde by the current best homography transform
        // this enables the current frame to be able to reference the original frame
    // compute H_tilde_gauss by implementing weighted distribution of discrete gaussian values
    // find H_tilde inverse for every cumulative H_tilde value
    // compute U transform by multiplying corresponding H_tilde_inverse[i] * H_tilde[i]
    // apply U transform to warp corresponding frame
    // output resultant frame images to disk
    Mat H_tilde = cv::Mat::eye(3, 3, CV_64FC1); // initialize as a 3x3 identity matrix
    std::vector<cv::Mat> H_transform_best_vec;
    std::vector<cv::Mat> H_tilde_vec;
    int num_frames = 102;

    // find H_tilde between each frame
    for (int i = 0; i < num_frames - 2; i++) {

        Mat frame_current = image_sequence[i];
        Mat frame_next = image_sequence[i + 1];
        Mat H_transform_best_current = calc_best_homography(frame_current, frame_next);  // use user-defined function that finds the best homography

        // cout << "H_transform_best_current: " << H_transform_best_current << endl << endl;

        // H_transform_best_vec.push_back(H_transform_best_current);

        H_tilde *= H_transform_best_current; // H_tilde cumulatively combines all previous homography transforms so can always reference original frame

        H_tilde_vec.push_back(H_tilde);

    }

    float w_2 = 0.1;
    float w_1 = 0.3;
    float w0 = 0.5;
    float w1 = 0.3;
    float w2 = 0.1;

    float w_sum;

    Mat H_t_2;
    Mat H_t_1;
    Mat H_t0;
    Mat H_t1;
    Mat H_t2;

    Mat H_t_2_mult;
    Mat H_t_1_mult;
    Mat H_t0_mult;
    Mat H_t1_mult;
    Mat H_t2_mult;

    Mat H_sum;
    Mat H_tilde_gauss;
    Mat H_tilde_gauss_inv;

    std::vector<cv::Mat> H_tilde_gauss_vec;
    std::vector<cv::Mat> H_tilde_gauss_inv_vec;

    // iterate through H_tilde_vec and apply a 1D gaussian filter to every H_tilde to smooth out the motion between the frames
    for (int i = 0; i < H_tilde_vec.size() - 1; i++) {
        // first frame; must exclude w_2 and w_1 weights
        if (i == 0) {
            H_t0 = H_tilde_vec[i];
            H_t1 = H_tilde_vec[i + 1];
            H_t2 = H_tilde_vec[i + 2];

            // need to convert multiplication -> cv::multiply(matrix, weight, matrix);
            cv::multiply(H_t0, w0, H_t0_mult);
            cv::multiply(H_t1, w1, H_t1_mult);
            cv::multiply(H_t2, w2, H_t2_mult);

            // sum of weights for division
            w_sum = w0 + w1 + w2;
            H_sum = H_t0_mult + H_t1_mult + H_t2_mult;

            // find solution of H_tilde_gauss
            cv::divide(H_sum, w_sum, H_tilde_gauss);

            // find inverse of H_tilde_gauss and store in H_tilde_gauss_inv
            cv::invert(H_tilde_gauss, H_tilde_gauss_inv);


            // store H_tilde_gauss result in H_tild_gauss_vec
            H_tilde_gauss_vec.push_back(H_tilde_gauss);

            // store H_tilde_gauss_inv result in H_tilde_gauss_inv_vec
            H_tilde_gauss_inv_vec.push_back(H_tilde_gauss_inv);

        }
        // second frame; must exclude w_2 weight
        else if (i == 1) {
            H_t_1 = H_tilde_vec[i - 1];
            H_t0 = H_tilde_vec[i];
            H_t1 = H_tilde_vec[i + 1];
            H_t2 = H_tilde_vec[i + 2];
```



```cpp

                // need to convert multiplication -> cv::multiply(matrix, weight, matrix);
                cv::multiply(H_t_1, w_1, H_t_1_mult);
                cv::multiply(H_t0, w0, H_t0_mult);
                cv::multiply(H_t1, w1, H_t1_mult);
                cv::multiply(H_t2, w2, H_t2_mult);

                // sum of weights for division
                w_sum = w_1 + w0 + w1 + w2;
                H_sum = H_t_1_mult + H_t0_mult + H_t1_mult + H_t2_mult;

                // find solution of H_tilde_gauss
                cv::divide(H_sum, w_sum, H_tilde_gauss);

                // find inverse of H_tilde_gauss and store in H_tilde_gauss_inv
                cv::invert(H_tilde_gauss, H_tilde_gauss_inv);


                // store H_tilde_gauss result in H_tild_gauss_vec
                H_tilde_gauss_vec.push_back(H_tilde_gauss);

                // store H_tilde_gauss_inv result in H_tilde_gauss_inv_vec
                H_tilde_gauss_inv_vec.push_back(H_tilde_gauss_inv);
            }
            // last frame; must exclude w1 and w2 weights
            else if (i == H_tilde_vec.size() - 1) {
                H_t_2 = H_tilde_vec[i - 2];
                H_t_1 = H_tilde_vec[i - 1];
                H_t0 = H_tilde_vec[i];

                // need to convert multiplication -> cv::multiply(matrix, weight, matrix);
                cv::multiply(H_t_2, w_2, H_t_2_mult);
                cv::multiply(H_t_1, w_1, H_t_1_mult);
                cv::multiply(H_t0, w0, H_t0_mult);


                // sum of weights for division
                w_sum = w_2 + w_1 + w0;
                H_sum = H_t_2_mult + H_t_1_mult + H_t0_mult;

                // find solution of H_tilde_gauss
                cv::divide(H_sum, w_sum, H_tilde_gauss);

                // find inverse of H_tilde_gauss and store in H_tilde_gauss_inv
                cv::invert(H_tilde_gauss, H_tilde_gauss_inv);


                // store H_tilde_gauss result in H_tild_gauss_vec
                H_tilde_gauss_vec.push_back(H_tilde_gauss);

                // store H_tilde_gauss_inv result in H_tilde_gauss_inv_vec
                H_tilde_gauss_inv_vec.push_back(H_tilde_gauss_inv);
            }
            // second to last frame; must exclude w2 weight
            else if (i == H_tilde_vec.size() - 2) {
                //w-2 * H_tilde_vec[i-2] + w - 1 * H_tilde_vec[i - 1] + w0 * H_tilde_vec[i] + w1 * H_tilde_vec[i + 1]
                H_t_2 = H_tilde_vec[i - 2];
                H_t_1 = H_tilde_vec[i - 1];
                H_t0 = H_tilde_vec[i];
                H_t1 = H_tilde_vec[i + 1];

                // need to convert multiplication -> cv::multiply(matrix, weight, matrix);
                cv::multiply(H_t_2, w_2, H_t_2_mult);
                cv::multiply(H_t_1, w_1, H_t_1_mult);
                cv::multiply(H_t0, w0, H_t0_mult);
                cv::multiply(H_t1, w1, H_t1_mult);


                // sum of weights for division
                w_sum = w_2 + w_1 + w0 + w1;
                H_sum = H_t_2_mult + H_t_1_mult + H_t0_mult + H_t1_mult;

                // find solution of H_tilde_gauss
                cv::divide(H_sum, w_sum, H_tilde_gauss);

                // find inverse of H_tilde_gauss and store in H_tilde_gauss_inv
                cv::invert(H_tilde_gauss, H_tilde_gauss_inv);


                // store H_tilde_gauss result in H_tild_gauss_vec
                H_tilde_gauss_vec.push_back(H_tilde_gauss);

                // store H_tilde_gauss_inv result in H_tilde_gauss_inv_vec
                H_tilde_gauss_inv_vec.push_back(H_tilde_gauss_inv);
            }
            // all other frames can utilize all 5 weights
            else {
                H_t_2 = H_tilde_vec[i - 2];
                H_t_1 = H_tilde_vec[i - 1];
                H_t0 = H_tilde_vec[i];
                H_t1 = H_tilde_vec[i + 1];
                H_t2 = H_tilde_vec[i + 2];

                cv::multiply(H_t_2, w_2, H_t_2_mult);
                cv::multiply(H_t_1, w_1, H_t_1_mult);
                cv::multiply(H_t0, w0, H_t0_mult);
                cv::multiply(H_t1, w1, H_t1_mult);
                cv::multiply(H_t2, w2, H_t2_mult);

                /*
                cout << "H_t_2_mult: " << H_t_2_mult << endl;
                cout << "H_t_1_mult: " << H_t_1_mult << endl;
                cout << "H_t0_mult: " << H_t0_mult << endl;
                cout << "H_t1_mult: " << H_t1_mult << endl;
                cout << "H_t2_mult: " << H_t2_mult << endl;
                */

```



```
            // sum of weights for division
            w_sum = w_2 + w_1 + w0 + w1 + w2;
            H_sum = H_t_2_mult + H_t_1_mult + H_t0_mult + H_t1_mult + H_t2_mult;

            // find solution of H_tilde_gauss
            cv::divide(H_sum, w_sum, H_tilde_gauss);

            // find inverse of H_tilde_gauss and store in H_tilde_gauss_inv
            cv::invert(H_tilde_gauss, H_tilde_gauss_inv);


            // store H_tilde_gauss result in H_tild_gauss_vec
            H_tilde_gauss_vec.push_back(H_tilde_gauss);

            // store H_tilde_gauss_inv result in H_tilde_gauss_inv_vec
            H_tilde_gauss_inv_vec.push_back(H_tilde_gauss_inv);

        }
    }

    // Specify the output folder path
    std::string outputFolderPath = "C:/Users/User/Documents/VUW/CGRA 352/first_OpenCV_program/first_OpenCV_program/first_OpenCV_program/image_sequence_warped";
    std::string fileExtension = ".png";  // Specify the file extension for the images

    std::vector<cv::Mat> image_sequence_smooth;

    cv::Mat centering_mat_2 = (cv::Mat_<double>(3, 3) << 1.0, 0.0, 50.0, 0.0, 1.0, 50.0, 0.0, 0.0, 1.0);

    // iterate through every frame and apply H_tilde_gauss_vec results to frames by using warpPerspective()
    // note: the first frame to be warped is the second frame, which corresponds to the first element in H_tilde_gauss_vec;
          // so adjust index; start at i = 0 and use i+1 for frame
    for (int i = 0; i < H_tilde_gauss_vec.size() - 1; i++) {
        Mat frame_current = image_sequence[i + 1];
        Mat frame_current_warped;
        Mat U_transform;

        // find U = H_tilde'^-1 * H_tilde for every iteration, then use U as the transform to warp every current frame in iteration
        U_transform = H_tilde_gauss_inv_vec[i] * H_tilde_vec[i];

        // multiply centering_mat into U_transform to center the results; offset to accoount for the border (size 100)
        Mat U_transform_centered = centering_mat_2 * U_transform;

        // apply warp to frame_041 and store in frame_041_warped
        cv::warpPerspective(frame_current, frame_current_warped, U_transform_centered, result_image_core_3_size, cv::INTER_LINEAR, border_mode, border_value);

        // append warped results
        image_sequence_smooth.push_back(frame_current_warped);

        // write out warped frames to disk
        std::string filename = outputFolderPath + "image_" + std::to_string(i) + fileExtension;
        cv::imwrite(filename, frame_current_warped); // Save the image as PNG
    }



    return 0;
}
```