CSE578 Computer Vision

Assignment 3 : Stereo Matching

Karnik Ram
2018701007

Different techniques for stereo image matching, along with their code is explained in this report. The generated results are provided at the end. The code files are provided in the `src` directory along with this submission.

The code is written in C++ and is written as a class (`StereoMatcher`) for modularity. To run the program,

```
mkdir build && cd build
cmake ..
make
./run <stereo-pair-path> <output-result-path>
# eg: ./run ../data/img1_1.png ../data/results
```

1 Techniques with Code

The different techniques are explained briefly in the following subsections, along with their corresponding code.

1.1 Dense-SIFT Matching

SIFT descriptors for every tenth pixel in the left and right images are extracted and then matched using a nearest neighbor search. OpenCV's SIFT and KNN modules are used for this.
The following is the corresponding code:

```
void StereoMatcher::denseSift(std::vector<cv::Point2f> &l_points, std::vector<cv::Point2f> &r_points)
{
    std::vector<cv::KeyPoint> keypoints1, keypoints2;
    cv::Mat descriptors1, descriptors2;

    cv::Mat img1 = stereo_pair[0];
    cv::Mat img2 = stereo_pair[1];

    //Define grid of keypoints
    size_t step = 10;
    for(size_t i = step; i < img1.rows - step; i+=step)
        for(size_t j = step; j < img1.cols - step; j+=step)
            keypoints1.push_back(cv::KeyPoint(float(j),float(i),float(step)));

    for(size_t i = step; i < img2.rows - step; i+=step)
        for(size_t j = step; j < img2.cols - step; j+=step)
            keypoints2.push_back(cv::KeyPoint(float(j),float(i),float(step)));

    //Description
    cv::Ptr<cv::xfeatures2d::SIFT> sift = cv::xfeatures2d::SIFT::create();
    sift->compute(img1,keypoints1,descriptors1);
    sift->compute(img2,keypoints2,descriptors2);

    //Matching
    cv::Ptr<cv::DescriptorMatcher> matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
    std::vector<std::vector<cv::DMatch>> knn_matches;
    matcher->knnMatch(descriptors1,descriptors2,knn_matches,2);

    //Refine matches using Lowe's ratio threshold
    const float ratio_thresh = 0.7f;
    std::vector<cv::DMatch> good_matches;
    for(size_t i = 0; i < knn_matches.size(); i++)
    {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
        {
```

```
            good_matches.push_back(knn_matches[i][0]);
        }
    }

    cv::Mat img_matches;
    cv::drawMatches(img1,keypoints1,img2,keypoints2,good_matches,img_matches,cv::Scalar
    ::all(-1),
        cv::Scalar::all(-1), std::vector<char>(), cv::DrawMatchesFlags::
    NOT_DRAW_SINGLE_POINTS);
    //cv::namedWindow("Good Matches",CV_WINDOW_NORMAL);
    //cv::imshow("Good Matches",img_matches);
    std::cout << "Number of matches found: " << good_matches.size() << std::endl;
    cv::imwrite(output_path + "/dense_sift.png",img_matches);
    //cv::waitKey(0);

    for(cv::DMatch match : good_matches)
    {
        cv::Point2f pt = keypoints1[match.queryIdx].pt;
        l_points.push_back(pt);

        pt = keypoints2[match.trainIdx].pt;
        r_points.push_back(pt);
    }
}


```

1.2 Intensity window correlation

A 5×5 window of pixels centered around one pixel is extracted for every tenth pixel in the left image and is compared with every such window in the right image using different metrics like SAD,SSD,Correlation,Normalized Correlation. The pixel corresponding to the window giving the best match is taken to be its corresponding point in the right image.
The code as follows:

```
void StereoMatcher::intensityWindowCorrelation(const size_t &window_size, std::vector<cv
    ::Point2f> &l_points, std::vector<cv::Point2f> &r_points)
{
    cv::Mat l_img = stereo_pair[0];
    cv::Mat r_img = stereo_pair[1];

    std::vector<float> window1,window2;
    cv::Point2f best_match;
    float score, best_score;

    for(size_t i = window_size/2; i < l_img.rows - window_size/2; i+=10)
        for(size_t j = window_size/2; j < l_img.cols - window_size/2; j+=10)
        {
            l_points.push_back(cv::Point2f(float(j),float(i)));
            window1 = getWindow(l_img,i,j,window_size);

            best_score = std::numeric_limits<float>::max();
            for(size_t ii = window_size/2; ii < r_img.rows - window_size/2; ii+=10)
                for(size_t jj = window_size/2; jj < r_img.cols - window_size/2; jj+=10)
                {
                    window2 = getWindow(r_img,ii,jj,window_size);
                    //score = sumSquaredDifference(window1,window2);
                    score = normalizedCorrelation(window1,window2);
                    if(score < best_score)
                    {
                        best_match = cv::Point2f(float(jj),float(ii));
                        best_score = score;
                    }
                }

            r_points.push_back(best_match);
        }

    // Display matches
    std::vector<cv::KeyPoint> keypoints1,keypoints2;
```

```
    std::vector<cv::DMatch> matches;
    for(size_t i = 0; i < l_points.size(); i++)
    {
        cv::DMatch match(i,i,0);
        cv::KeyPoint keypoint1(l_points[i],5);
        cv::KeyPoint keypoint2(r_points[i],5);
        keypoints1.push_back(keypoint1);
        keypoints2.push_back(keypoint2);
        matches.push_back(match);
    }

    //cv::namedWindow("Intensity Window Matches",CV_WINDOW_NORMAL);

    cv::Mat img_matches;
    cv::drawMatches(l_img,keypoints1,r_img,keypoints2,matches,img_matches,cv::Scalar::
    all(-1),
        cv::Scalar::all(-1), std::vector<char>(), cv::DrawMatchesFlags::
    NOT_DRAW_SINGLE_POINTS);

    //cv::imshow("Intensity Window Matches",img_matches);
    cv::imwrite(output_path + "/window.png",img_matches);
    //cv::waitKey(0);
```

SSD, Correlation, and Normalized Correlation were tried out but none gave good results.

```
float StereoMatcher::sumSquaredDifference(const std::vector<float> &window1, const std::
    vector<float> &window2)
{
    float sum = 0;
    for(size_t i = 0; i < window1.size(); i++)
        sum += (window1[i]-window2[i])*(window1[i]-window2[i]);

    return sum;
}

float StereoMatcher::correlation(const std::vector<float> &window1, const std::vector<
    float> &window2)
{
    float sum1=0,sum2=0,sum3=0,correlation;
    for(size_t i = 0; i < window1.size(); i++)
    {
        sum1 += window1[i]*window2[i];
        sum2 += window1[i]*window1[i];
        sum3 += window2[i]*window2[i];
    }

    correlation = sum1/(sqrt(sum2)*sqrt(sum3));
    return correlation;
}

float StereoMatcher::normalizedCorrelation(const std::vector<float> &window1, const std
    ::vector<float> &window2)
{
    float sum1=0,sum2=0,sum3=0,nor_correlation;
    std::vector<float> mwindow1,mwindow2;
    mwindow1 = window1;
    mwindow2 = window2;

    // Mean subtraction
    float average = accumulate(mwindow1.begin(),mwindow1.end(),0.0)/mwindow1.size();
    for(float &d : mwindow1)
        d-=average;

    average = accumulate(mwindow1.begin(),mwindow1.end(),0.0)/mwindow1.size();
    for(float &d : mwindow2)
        d-=average;

    for(size_t i = 0; i < mwindow1.size(); i++)
    {
        sum1 += mwindow1[i]*mwindow2[i];
        sum2 += mwindow1[i]*mwindow1[i];
        sum3 += mwindow2[i]*mwindow2[i];
```

```
    }

    nor_correlation = sum1/(sqrt(sum2)*sqrt(sum3));
    return nor_correlation;
}

```

1.3 Rectification

Rectifying the image pair i.e. making the images parallel to one another can reduce the matching problem to a 1D search problem that's based on epipolar matching. This can signficantly speed up the process. For this the fundamental matrix between the two images has to be first estimated. This is done using the standard 8-point algorithm.
The code is as follows:

```
    void StereoMatcher::estimateFundamentalMatrix(const Eigen::MatrixXf &X1, const Eigen
    ::MatrixXf &X2,
        Eigen::MatrixXf &F)
{
    Eigen::MatrixXf A(2*X1.rows(), 9);
    Eigen::ArrayXf f(9);
    for(int i = 0, j = 0; i < A.rows(); i++, j++)
        A.row(i) << X1(i,1)*X2(i,1), X2(i,1)*X1(i,2), X2(i,1), X2(i,2)*X1(i,1), X2(i,2)*
    X1(i,2), X2(i,2), X1(i,1), X1(i,2),1;

    // Gave bad results for some unknown reason
    //Eigen::JacobiSVD<Eigen::MatrixXf> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV)
    ;

    cv::Mat cv_A,U,S,Vt;
    eigen2cv(A,cv_A);
    cv::SVD::compute(cv_A,U,S,Vt);

    Eigen::MatrixXf V,tempV;
    cv2eigen(Vt,tempV);
    V = tempV.transpose();

    f = V.col(V.cols()-1);

    F.resize(3,3);
    F.row(0) = f.segment(0,3);
    F.row(1) = f.segment(3,3);
    F.row(2) = f.segment(6,3);

    std::cout << "F:\n" << F << std::endl;
}

```

This is estimated within a RANSAC scheme for robustness.

```
    void StereoMatcher::estimateRansacFundamentalMatrix(const Eigen::MatrixXf &X1,
    const Eigen::MatrixXf &X2,
        const float &dist_threshold, const float &ratio_threshold, Eigen::MatrixXf &F,
        std::vector<int> &inlier_indices)
{
    std::cout << "Estimating Fundamental matrix within RANSAC..." << std::endl;
    Eigen::MatrixXf sample_X1, sample_X2;
    Eigen::Vector3f x1,x2;
    std::vector<int> largest_support;
    float inlier_avg = 0;

    for(size_t i = 0; i < 2000; i++)
    {
        std::cout << "\n\nIteration#" << i+1 << std::endl;
        std::vector<int> sample_indices = utils::generateRandomVector(0,X1.rows()-1,4);

        sampleFromX1X2(X1,X2,sample_indices,sample_X1,sample_X2);
        estimateFundamentalMatrix(sample_X1,sample_X2,F);

        for(size_t j = 0; j < X1.rows(); j++)
        {
```

```
            if(std::find(sample_indices.begin(),sample_indices.end(),j) !=
    sample_indices.end())
                continue;

            x1 = X1.row(j);
            x2 = X2.row(j);

            if( x2.transpose()*F*x1 <= dist_threshold )
            {
                inlier_indices.push_back(j);
                inlier_avg += x2.transpose()*F*x1;
            }
        }

        std::cout << "\nNumber of inliers: " << inlier_indices.size() << std::endl;
        std::cout << "Inlier avg. reprojecion error: " << inlier_avg / inlier_indices.
    size() << std::endl;
        inlier_avg = 0;

        if(inlier_indices.size() >= X1.rows() * ratio_threshold)
        {
            std::cout << "\nFound a model!\nNumber of inliers: " << inlier_indices.size
    () << std::endl;
            inlier_indices.insert(inlier_indices.end(),sample_indices.begin(),
    sample_indices.end());
            sampleFromX1X2(X1,X2,inlier_indices,sample_X1,sample_X2);
            estimateFundamentalMatrix(sample_X1,sample_X2,F);
            std::cout << "Average error over inliers and sample set: " <<
    calcAvgFundamentalMatrixError(sample_X1,sample_X2,F) << std::endl;
            return;
        }

        else
        {
            if(largest_support.size() < inlier_indices.size())
            {
                largest_support = inlier_indices;
                largest_support.insert(largest_support.end(),sample_indices.begin(),
    sample_indices.end());
            }

            inlier_indices.clear();
        }
    }

    if(largest_support.size() >= 4)
    {
        std::cout << "\nCould not find a model according to threshold!\nSo using largest
     inlier set instead." << std::endl;
        sampleFromX1X2(X1,X2,largest_support,sample_X1,sample_X2);
        estimateFundamentalMatrix(sample_X1,sample_X2,F);
        inlier_indices = largest_support;
        std::cout << "Number of inliers: " << largest_support.size() << std::endl;
        std::cout << "Average error over inliers and sample set: " <<
    calcAvgFundamentalMatrixError(sample_X1,sample_X2,F) << std::endl;
    }

    else
        std::cout << "Could not find a model!" << std::endl;
}

```

Using this estimated F matrix, homographies for rectifying the images are then estimated using OpenCV's `stereoRectifyUncalibrated`. The homographies are then applied to the images using OpenCV's `warpPerspective` to obtain the final rectified images.

1.4 Greedy Matching

After rectifying the images, the images and their corresponding epipolar lines become parallel to each other. This can be exploited to speed up the matching process by searching for the match only along the

corresponding epipolar line, or simply - fixing the search row in the target image to the row the point belongs to in the first image.

```
void StereoMatcher::greedyMatching(const Eigen::MatrixXf &F, const int &window_size,
    std::vector<cv::Point2f> &l_points, std::vector<cv::Point2f> &r_points)
{
    cv::Mat l_img = rectified_stereo_pair[0];
    cv::Mat r_img = rectified_stereo_pair[1];
    //Eigen::Vector3f pt, pt_corres;
    cv::Point2f best_pt;

    std::vector<float> window1,window2;
    float score,best_score;

    for(size_t i = window_size/2; i < l_img.rows - window_size/2; i+=10)
        for(size_t j = window_size/2; j < l_img.cols - window_size/2; j+=10)
        {
            l_points.push_back(cv::Point2f(float(j),float(i)));
            window1 = getWindow(l_img,i,j,window_size);
            best_score = std::numeric_limits<float>::max();

            //pt = Eigen::Vector3f(j,i,1);
            //pt_corres = F * pt;
            //std::cout << pt_corres << std::endl;
            //pt_corres = pt_corres/pt_corres(2);

            //int ii = pt_corres(0);
            size_t ii = i;
            for(size_t jj = window_size/2; jj < r_img.cols - window_size/2; jj+=10)
            {
                window2 = getWindow(r_img,ii,jj,window_size);
                //score = sumSquaredDifference(window1,window2);
                score = normalizedCorrelation(window1,window2);
                if(score < best_score)
                {
                    best_pt = cv::Point2f(float(jj),float(ii));
                    best_score = score;
                }
            }
            r_points.push_back(best_pt);
        }

    // Display matches
    std::vector<cv::KeyPoint> keypoints1,keypoints2;
    std::vector<cv::DMatch> matches;
    for(size_t i = 0; i < l_points.size(); i++)
    {
        cv::DMatch match(i,i,0);
        cv::KeyPoint keypoint1(l_points[i],5);
        cv::KeyPoint keypoint2(r_points[i],5);
        keypoints1.push_back(keypoint1);
        keypoints2.push_back(keypoint2);
        matches.push_back(match);
    }

    //cv::namedWindow("Matches",CV_WINDOW_NORMAL);

    cv::Mat img_matches;
    cv::drawMatches(l_img,keypoints1,r_img,keypoints2,matches,img_matches,cv::Scalar::
    all(-1),
        cv::Scalar::all(-1), std::vector<char>(), cv::DrawMatchesFlags::
    NOT_DRAW_SINGLE_POINTS);
    cv::imwrite(output_path + "/greedy.png",img_matches);
    //cv::imshow("Matches",img_matches);
    //cv::waitKey(0);
    std::cout << "Found " << r_points.size() << " matches!\n";

```

2 Results

All the input and generated images can also be found here - link.

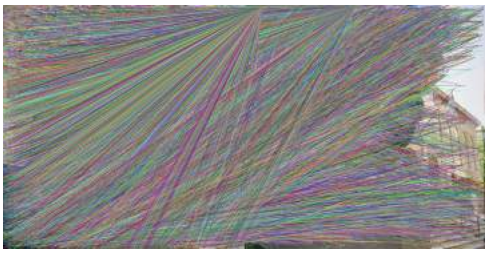
2.1 Dense-SIFT Matching

(a) Pair 1

(b) Pair 2

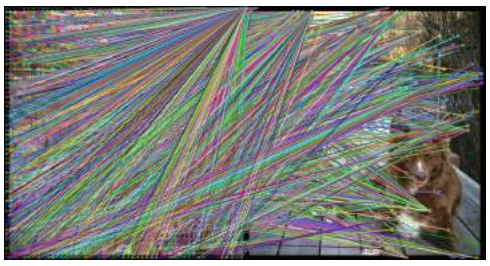
(c) Pair 3

2.2 Intensity window correlation



(a) Pair 1

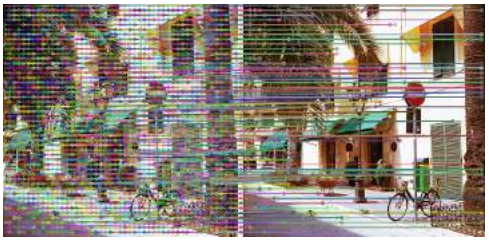
(b) Pair 2



(c) Pair 3

2.3 Greedy matching

(a) Pair 1



(b) Pair 2

(c) Pair 3

3 Comparison

The runtimes (on my PC) for the above techniques are as follows:

Method	**Pair 1**	**Pair 2**	**Pair 3**
Dense SIFT	4.51 s	0.49 s	0.55 s
Intensity Window	2289.59 s	30.86 s	40.25 s
Greedy Matching	15.737 s	0.774 s	0.828 s

Intensity window based matching performed the worst in terms of runtime as well as in terms of the quality of the matches. SSD, Correlation, and Normalized Correlation metrics were tried out but they didn't make much of a difference. Greedy matching improved the runtime significantly - almost by a factor of 50 - but the quality of matches were still bad. Almost all the points along each line matched to the same point on the other image.