HOMEWORK 3

Calculating Optical Flow

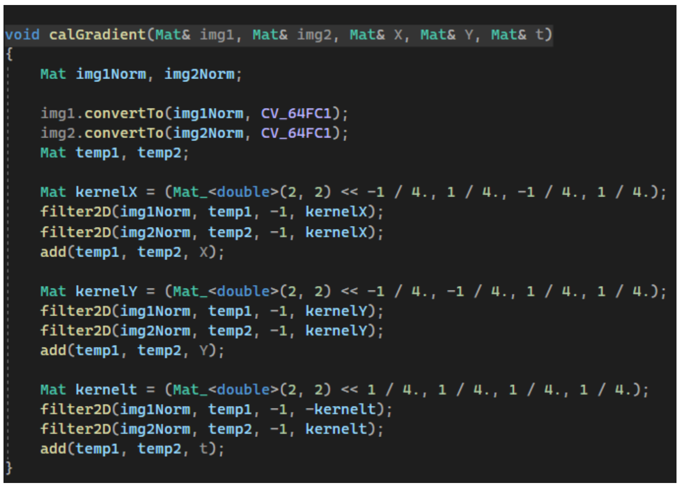
Student Name: Rohit Das **Student ID**: 61047086s

Objective: Calculating optical flow using specific algorithm and synthetically translate lena.bmp one pixel right and downward

Algorithm Used: Horn and Schunck Optical Flow Estimation

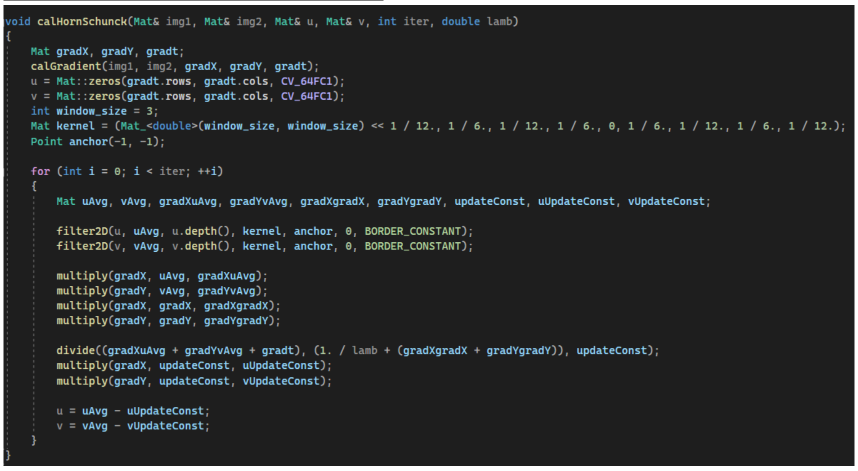
Screenshot of the Algorithm:

Call Gradient:



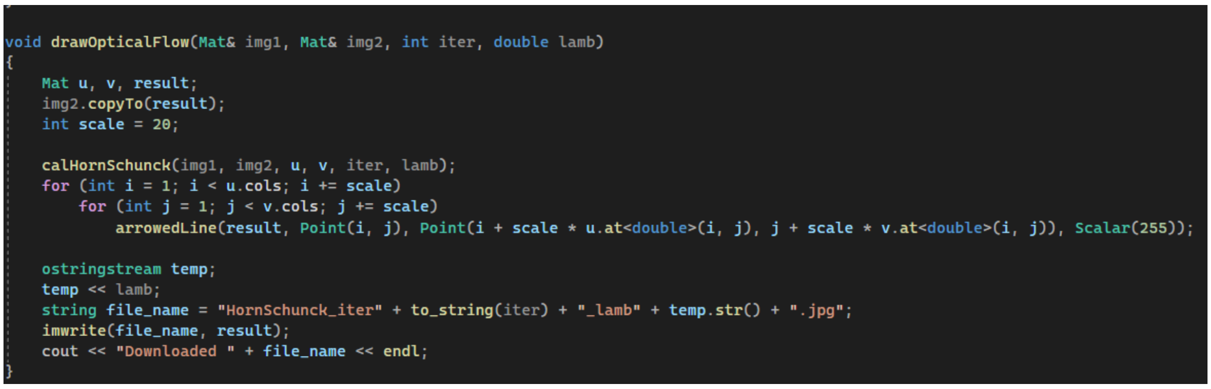
```
void calGradient(Mat& img1, Mat& img2, Mat& X, Mat& Y, Mat& t)
{
    Mat img1Norm, img2Norm;

    img1.convertTo(img1Norm, CV_64FC1);
    img2.convertTo(img2Norm, CV_64FC1);
    Mat temp1, temp2;

    Mat kernelX = (Mat_<double>(2, 2) << -1 / 4., 1 / 4., -1 / 4., 1 / 4.);
    filter2D(img1Norm, temp1, -1, kernelX);
    filter2D(img2Norm, temp2, -1, kernelX);
    add(temp1, temp2, X);

    Mat kernelY = (Mat_<double>(2, 2) << -1 / 4., -1 / 4., 1 / 4., 1 / 4.);
    filter2D(img1Norm, temp1, -1, kernelY);
    filter2D(img2Norm, temp2, -1, kernelY);
    add(temp1, temp2, Y);

    Mat kernelt = (Mat_<double>(2, 2) << 1 / 4., 1 / 4., 1 / 4., 1 / 4.);
    filter2D(img1Norm, temp1, -1, -kernelt);
    filter2D(img2Norm, temp2, -1, kernelt);
    add(temp1, temp2, t);
}
```

Horn and Schunk Calculation



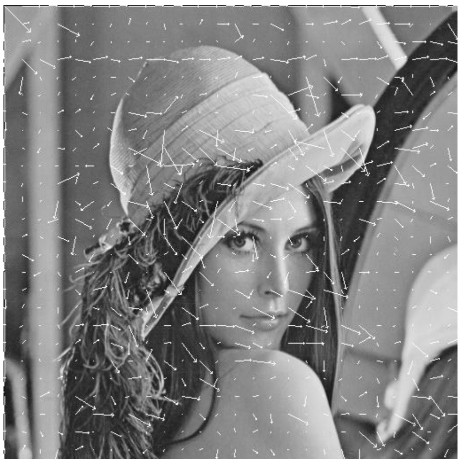
```
void calHornSchunck(Mat& img1, Mat& img2, Mat& u, Mat& v, int iter, double lamb)
{
    Mat gradX, gradY, gradt;
    calGradient(img1, img2, gradX, gradY, gradt);
    u = Mat::zeros(gradt.rows, gradt.cols, CV_64FC1);
    v = Mat::zeros(gradt.rows, gradt.cols, CV_64FC1);
    int window_size = 3;
    Mat kernel = (Mat_<double>(window_size, window_size) << 1 / 12., 1 / 6., 1 / 12., 1 / 6., 0, 1 / 6., 1 / 12., 1 / 6., 1 / 12.);
    Point anchor(-1, -1);

    for (int i = 0; i < iter; ++i)
    {
        Mat uAvg, vAvg, gradXuAvg, gradYvAvg, gradXgradX, gradYgradY, updateConst, uUpdateConst, vUpdateConst;

        filter2D(u, uAvg, u.depth(), kernel, anchor, 0, BORDER_CONSTANT);
        filter2D(v, vAvg, v.depth(), kernel, anchor, 0, BORDER_CONSTANT);

        multiply(gradX, uAvg, gradXuAvg);
        multiply(gradY, vAvg, gradYvAvg);
        multiply(gradX, gradX, gradXgradX);
        multiply(gradY, gradY, gradYgradY);

        divide((gradXuAvg + gradYvAvg + gradt), (1. / lamb + (gradXgradX + gradYgradY)), updateConst);
        multiply(gradX, updateConst, uUpdateConst);
        multiply(gradY, updateConst, vUpdateConst);

        u = uAvg - uUpdateConst;
        v = vAvg - vUpdateConst;
    }
}
```

Draw optical Flow:

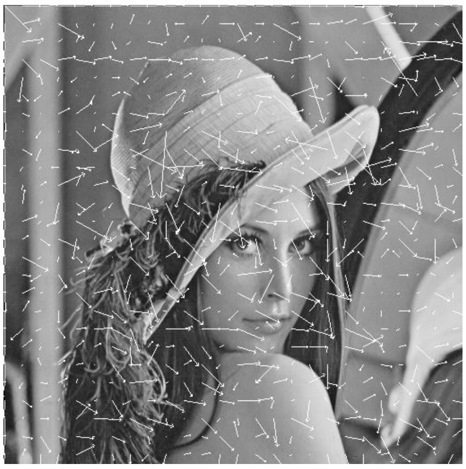


```
void drawOpticalFlow(Mat& img1, Mat& img2, int iter, double lamb)
{
    Mat u, v, result;
    img2.copyTo(result);
    int scale = 20;

    calHornSchunck(img1, img2, u, v, iter, lamb);
    for (int i = 1; i < u.cols; i += scale)
        for (int j = 1; j < v.cols; j += scale)
            arrowedLine(result, Point(i, j), Point(i + scale * u.at<double>(i, j), j + scale * v.at<double>(i, j)), Scalar(255));

    ostringstream temp;
    temp << lamb;
    string file_name = "HornSchunck_iter" + to_string(iter) + "_lamb" + temp.str() + ".jpg";
    imwrite(file_name, result);
    cout << "Downloaded " + file_name << endl;
}
```

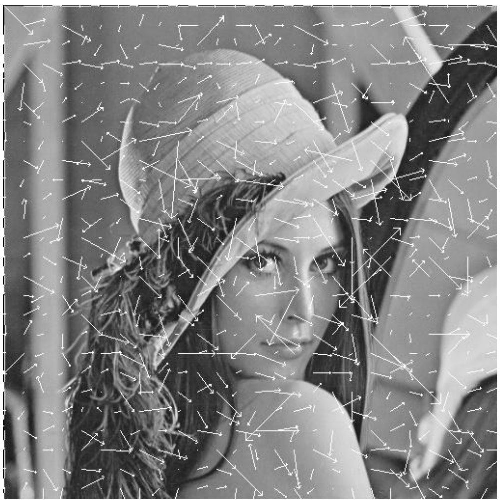
Results



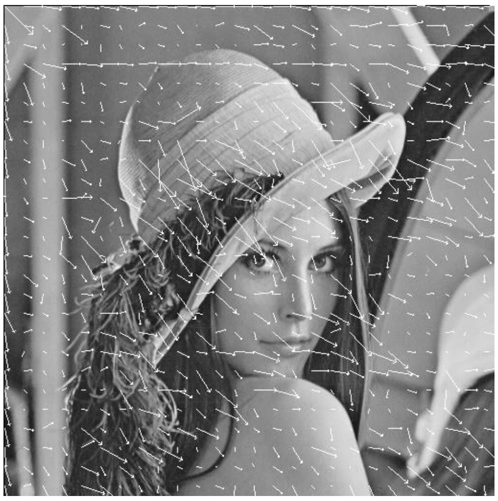
HornSchunck_iter1_lamb0.1



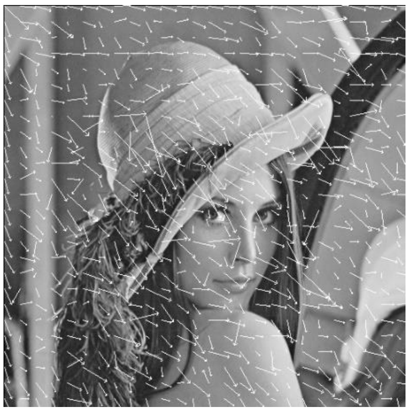
HornSchunck_iter1_lamb1



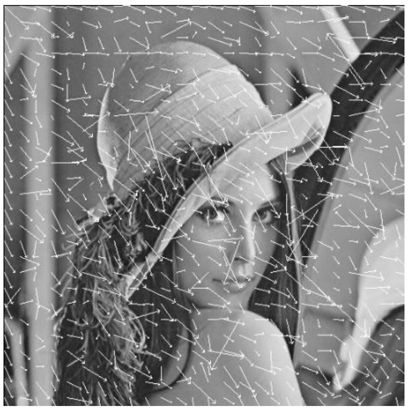
HornSchunck_iter1_lamb10



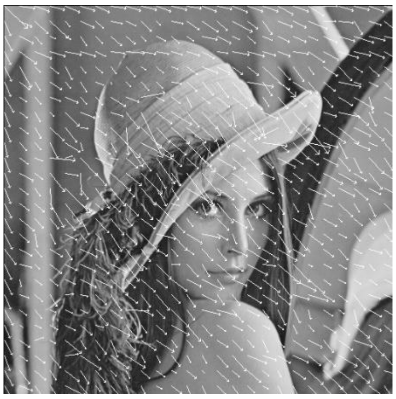
HornSchunck_iter4_lamb0.1



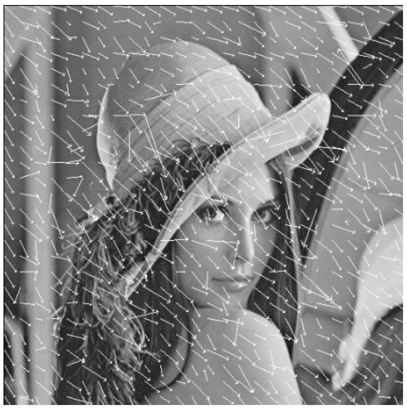
HornSchunck_iter4_lamb1



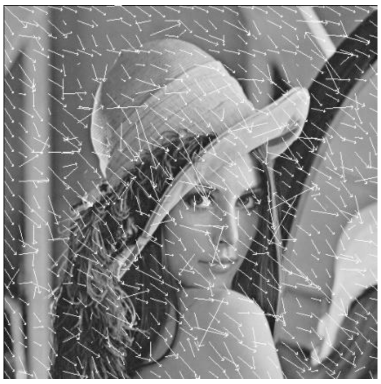
HornSchunck_iter4_lamb10



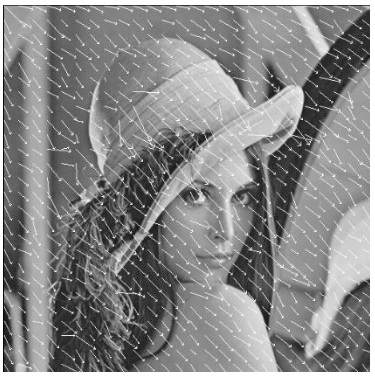
HornSchunck_iter16_lamb0.1



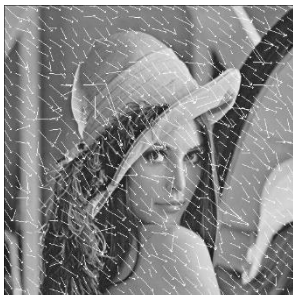
HornSchunck_iter16_lamb1



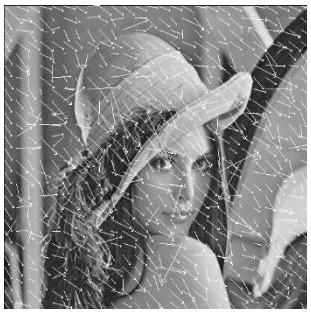
HornSchunck_iter16_lamb10



HornSchunck_iter64_lamb0.1



HornSchunck_iter64_lamb1



HornSchunck_iter64_lamb10