IPMV-Experiment-6

Lab3 Circles from corner detection

课程名称：图像处理与机器视觉

实验地点：嘉定校区智信馆 131

指导教师：Lei Jiang，Rui FAN

姓　　名：姚天亮

学　　号：2150248

Task

- Create your own corner key point detector
- Use corner key points and RANSAC to find a circle from picture below.

Main steps in brief

Compute the image gradients using derivative of Gaussian filters

Compute the images A, B, C

Element wise products of the gradients

Apply windowing by convolving these with a (bigger) Gaussian

Use A, B, and C to compute a corner metric for the entire image

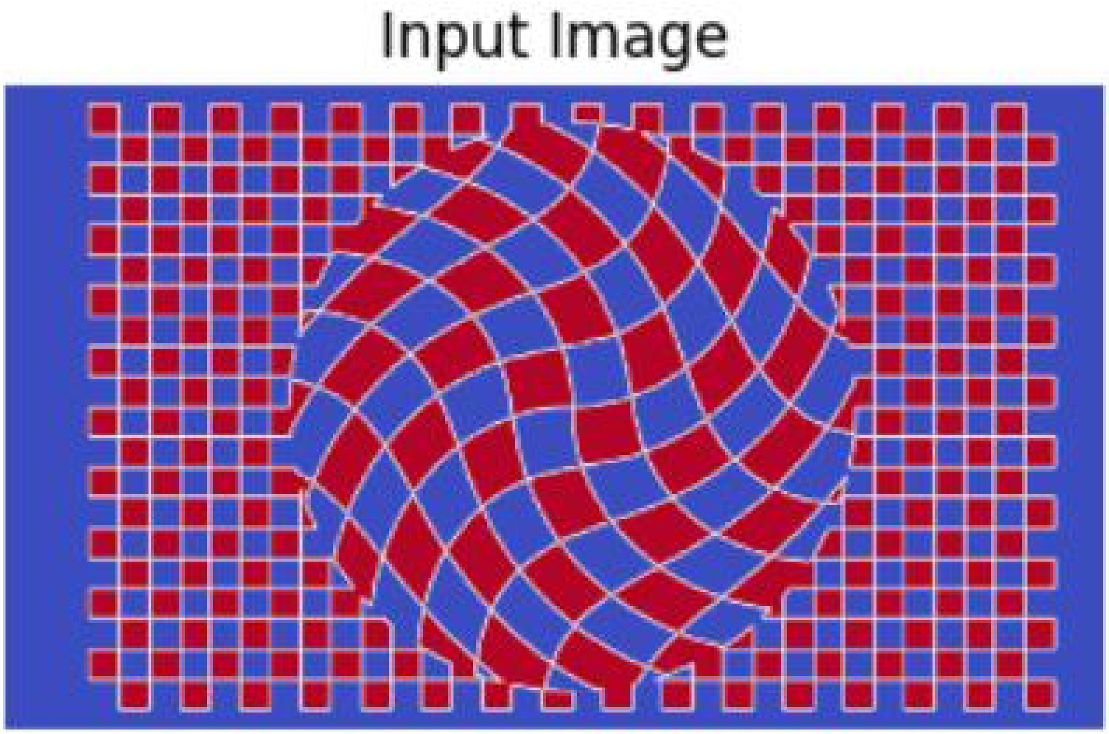
λmin, Harris, Harmonic Mean, other?

Threshold the corner metric image and find local maxima

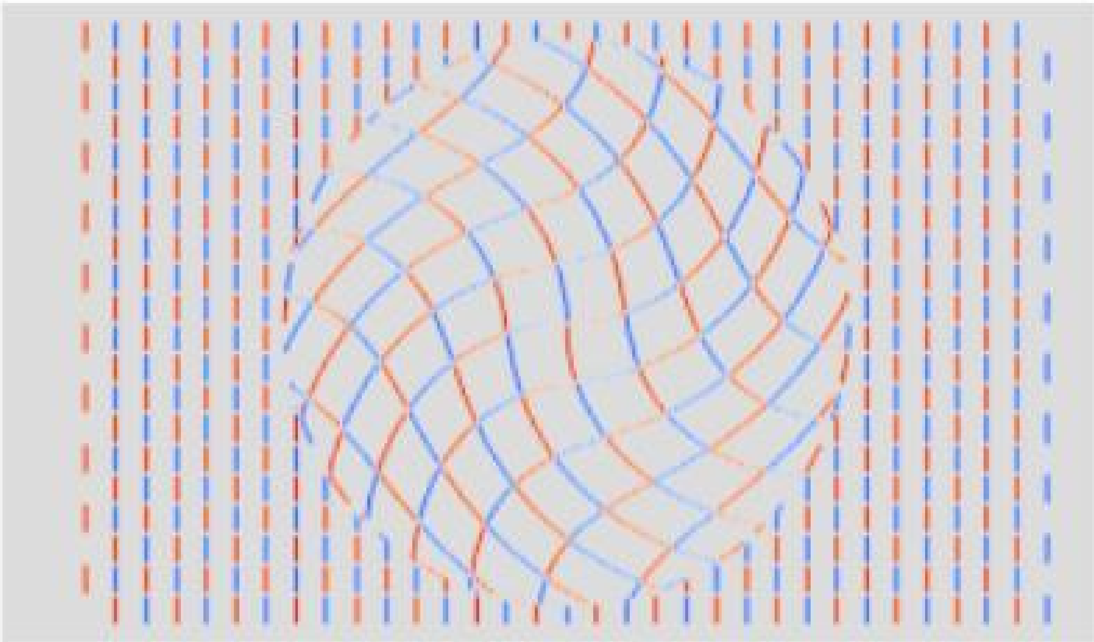
Morphological operations

Logical operations

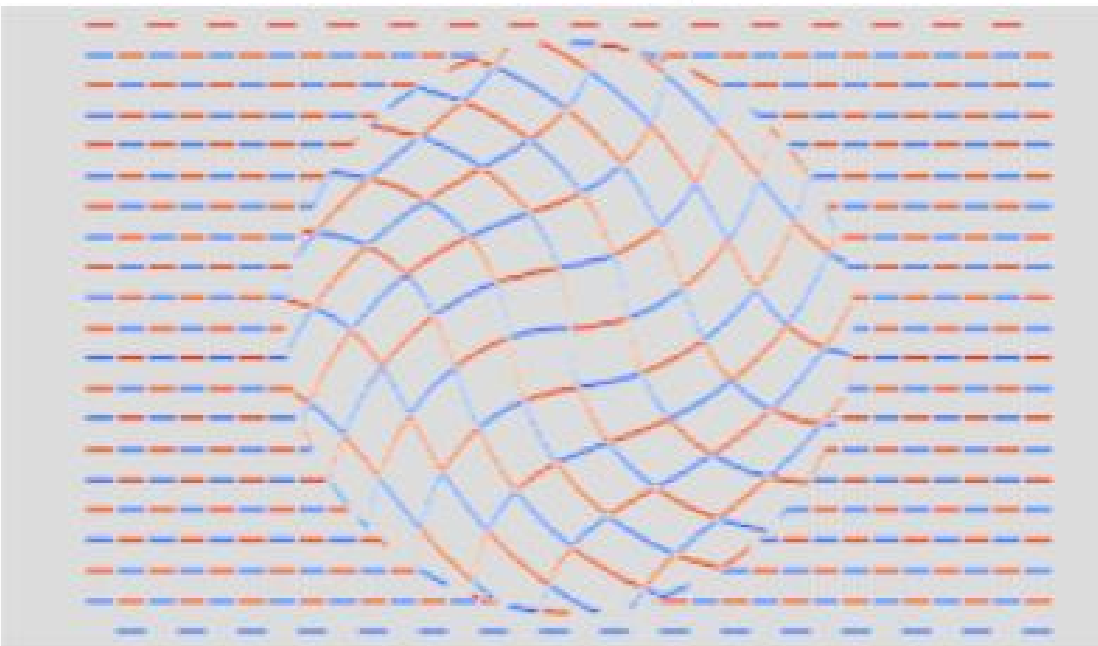
使用 coolwarm_r 来进行更加清晰地可视化



Gradient Ix

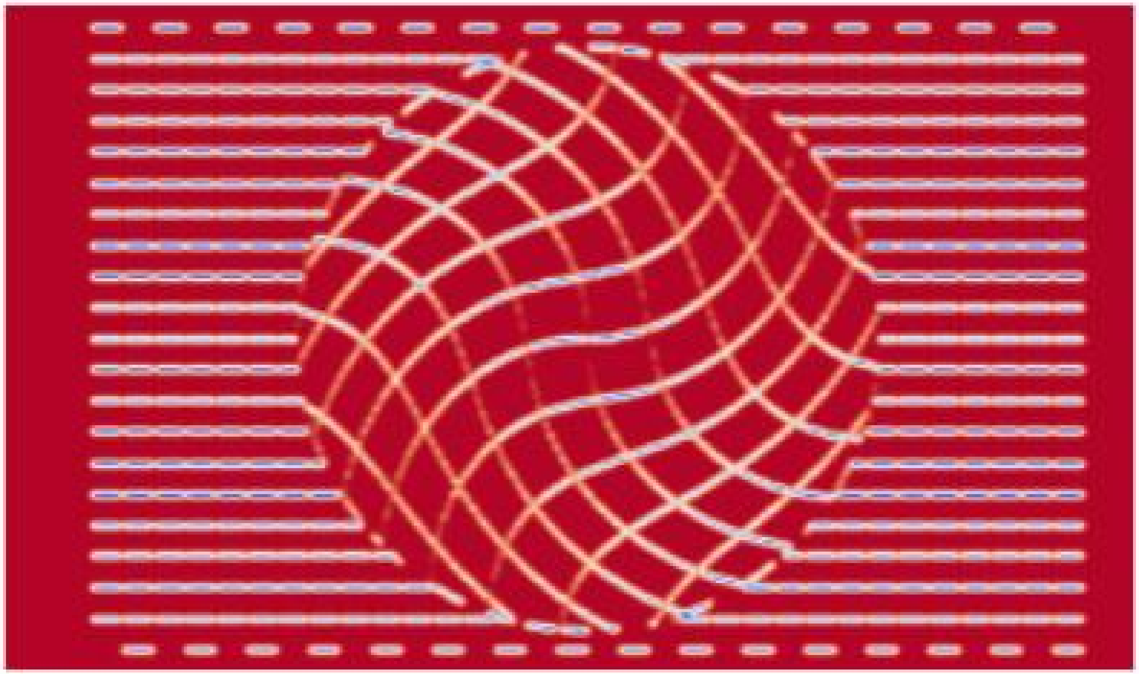


Gradient Iy

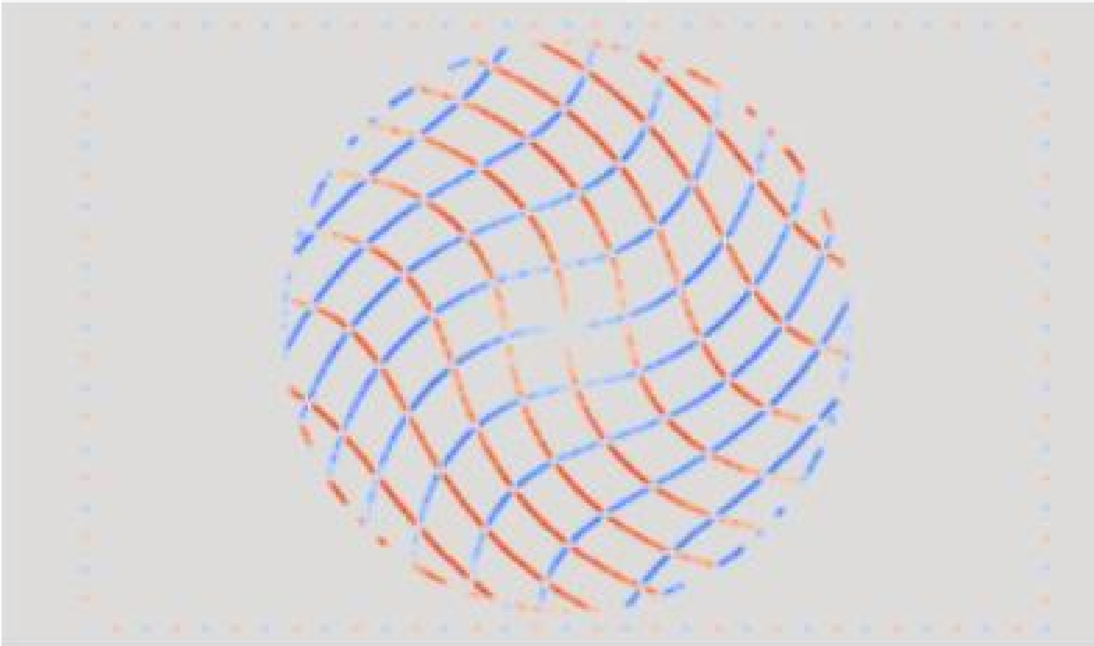


A (Ix^2)

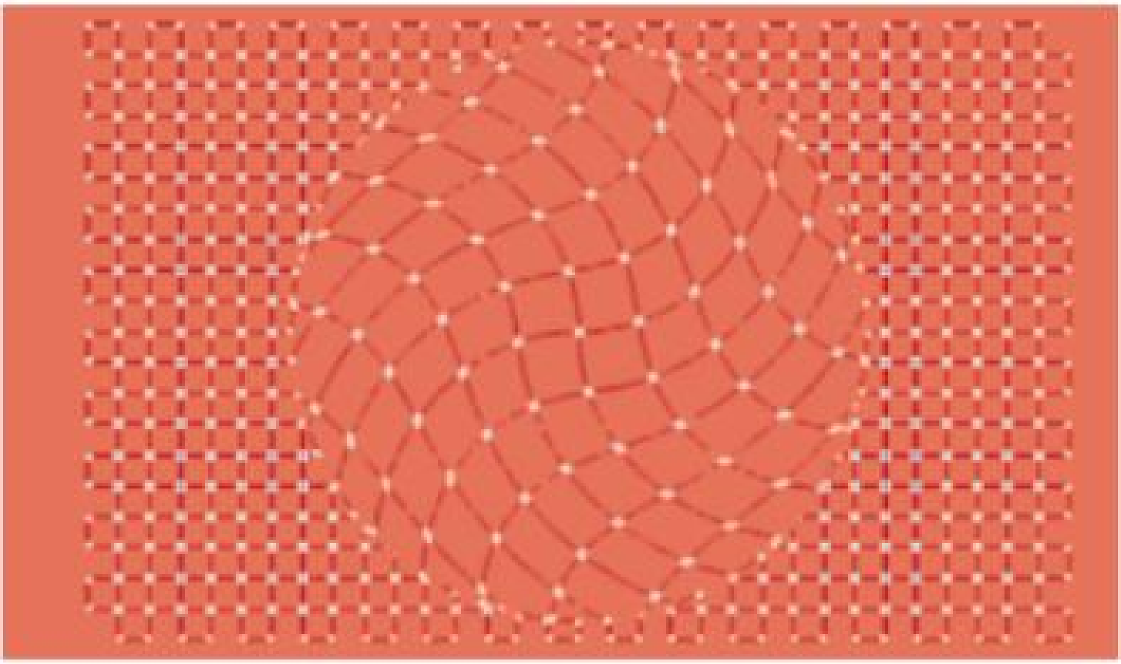
B (Iy^2)



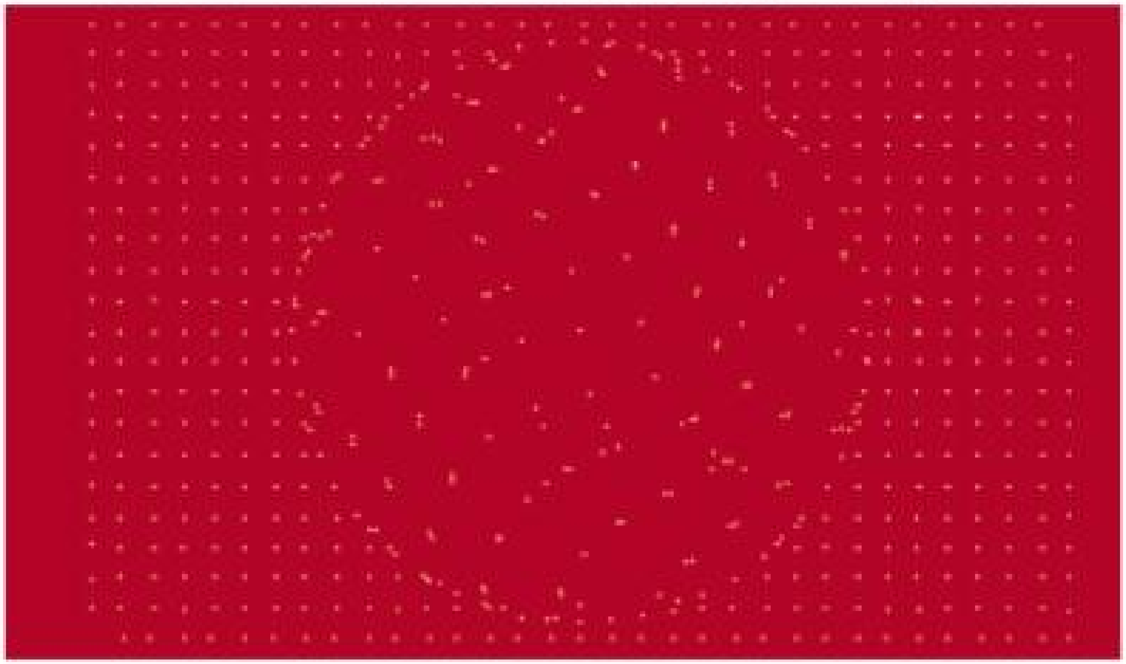
C (Ix * Iy)



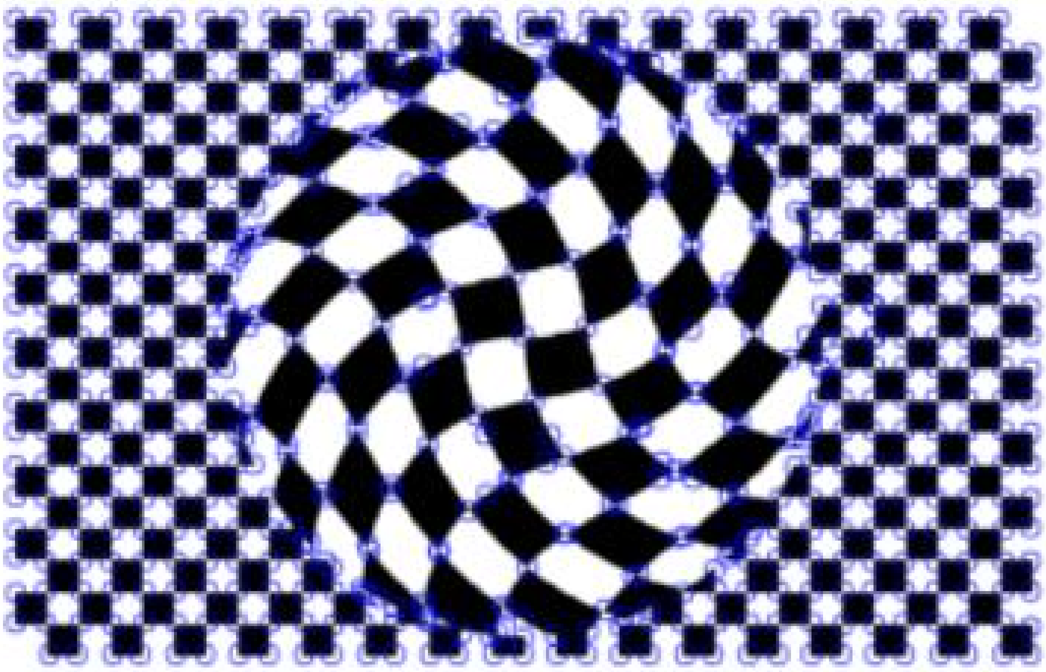
Harris Response

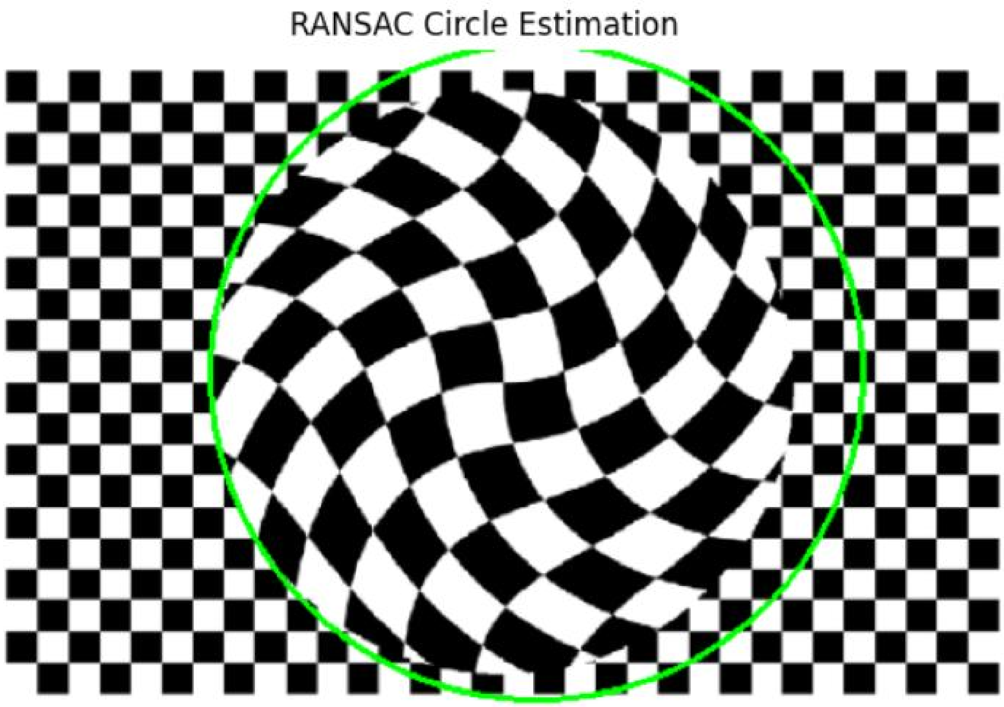


Local Maxima (Binary Map)



Harris Corners





附件：代码

```
#include "circle_estimator.h"
#include <random>

CircleEstimator::CircleEstimator(double p, float distance_threshold)
        : p_{p}
        , distance_threshold_{distance_threshold}
{}

CircleEstimate CircleEstimator::estimate(const Eigen::Matrix2Xf& points) const
{
    if (points.cols() < 3)
    {
        // Too few points to estimate any circle.
        return {};
    }

    // Estimate circle using RANSAC.
    CircleEstimate estimate = ransacEstimator(points);
    if (estimate.num_inliers == 0)
    { return {}; }
```

```
    // Extract the inlier points.
    Eigen::Matrix2Xf inlier_pts = extractInlierPoints(estimate, points);

    // Estimate circle based on all the inliers.
    estimate.circle = leastSquaresEstimator(inlier_pts);

    return estimate;
}

CircleEstimate CircleEstimator::ransacEstimator(const Eigen::Matrix2Xf& pts) const
{
    // Initialize best set.
    Eigen::Index best_num_inliers{0};
    Circle best_circle;
    LogicalVector best_is_inlier;

    // Set up random number generator.
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> uni_dist(0, static_cast<int>(pts.cols()-1));

    // Initialize maximum number of iterations.
    int max_iterations = std::numeric_limits<int>::max();

    // Perform RANSAC.
    int iterations{0};
    for (; iterations < max_iterations; ++iterations)
    {
        // Determine test circle by drawing minimal number of samples.
        Circle tst_circle(pts.col(uni_dist(gen)),
                                          pts.col(uni_dist(gen)),
                                          pts.col(uni_dist(gen)));

        // Count number of inliers.
        LogicalVector is_inlier = tst_circle.distance(pts).array() < distance_threshold_;
        Eigen::Index tst_num_inliers = is_inlier.count();

        // Check if this estimate gave a better result.
        // Todo 8: Remove break and perform the correct test!
        break;          // Remove!
        if (false) // Perform the correct test!
        {
            // Update circle with largest inlier set.
            best_circle = tst_circle;
```

```
            best_num_inliers = tst_num_inliers;
            best_is_inlier = is_inlier;

            // Update max iterations.
            double inlier_ratio = static_cast<double>(best_num_inliers) / static_cast<double>(pts.cols());
            max_iterations = static_cast<int>(std::log(1.0 - p_) / std::log(1.0 - inlier_ratio*inlier_ratio*inlier_ratio));
        }
    }

    return {best_circle, iterations, best_num_inliers, best_is_inlier};
}

Circle CircleEstimator::leastSquaresEstimator(const Eigen::Matrix2Xf& pts) const
{
    // The equations for the points (x_i, y_i) on the circle (x_c, y_c, r) is:
    //      (x_i - x_c)^2 + (y_i - y_c)^2 = r^2
    //
    // By multiplying out, we get the linear equations
    //      (2*x_c)*x_i + (2*y_c)*y_i + (r^2 - x_c^2 - y_x^2) = x_i^2 + y_i^2
    //
    // The least-squares problem then has the form A*p = b, where
    //      A = [x_i, y_i, 1],
    //      p = [2*x_c, 2*y_c, r^2 - x_c^2 - y_x^2]^T,
    //      b = [x_i^2 + y_i^2]
    //
    // by solving for p = [p_0, p_1, p_2], we get the following estimates for the circle parameters:
    //      x_c = 0.5 * p_0,
    //      y_c = 0.5 * p_1,
    //      r = sqrt(p_2 + x_c^2 + y_c^2)

    // Construct A and b.
    Eigen::MatrixXf A(pts.cols(), 3);
    A.leftCols(2) = pts.transpose();
    A.col(2).setConstant(1.0f);
    Eigen::VectorXf b = pts.colwise().squaredNorm();

    // Determine solution for p.
    // See https://eigen.tuxfamily.org/dox-devel/group__LeastSquares.html
    Eigen::Vector3f p = A.colPivHouseholderQr().solve(b);

    // Extract center point and radius from the parameter vector p.
    Eigen::Vector2f center_point = 0.5f * p.head<2>();
```

```cpp
    float radius = std::sqrt(p(2) + center_point.squaredNorm());

    return {center_point, radius};
}

Eigen::Matrix2Xf CircleEstimator::extractInlierPoints(const CircleEstimate& estimate, const Eigen::Matrix2Xf& pts) const
{
    Eigen::Matrix2Xf inliers(2, estimate.num_inliers);

    int curr_col = 0;
    for (int i = 0; i < pts.cols(); ++i)
    {
        if (estimate.is_inlier(i))
        {
            inliers.col(curr_col++) = pts.col(i);
        }
    }

    return inliers;
}


#include "corner_detector.h"

CornerDetector::CornerDetector(CornerMetric metric,
                               bool do_visualize,
                               float quality_level,
                               float gradient_sigma,
                               float window_sigma)
        : metric_type_{metric}
        , do_visualize_{do_visualize}
        , quality_level_{quality_level}
        , window_sigma_{window_sigma}
        , g_kernel_{create1DGaussianKernel(gradient_sigma)}
        , dg_kernel_{create1DDerivatedGaussianKernel(gradient_sigma)}
        , win_kernel_{create1DGaussianKernel(window_sigma_)}
{ }

std::vector<cv::KeyPoint> CornerDetector::detect(const cv::Mat& image) const
{
    // Estimate image gradients Ix and Iy using g_kernel_ and dg_kernel.
    // Todo 2: Estimate image gradients Ix and Iy using g_kernel_ and dg_kernel_.
    cv::Mat Ix;
```

```
    cv::Mat Iy;

//    cv::sepFilter2D(image, Ix, CV_32F, ?, ?);
//    cv::sepFilter2D(image, Iy, CV_32F, ?, ?);

    // Compute the elements of M; A, B and C from Ix and Iy.
    // Todo 3.1: Compute the elements of M; A, B and C from Ix and Iy.
    cv::Mat A;
    cv::Mat B;
    cv::Mat C;

    // Apply the windowing gaussian win_kernel_ on A, B and C.
    // Todo 3.2: Apply the windowing gaussian.

    // Compute corner response.
    // Todo 4: Finish all the corner response functions.
    cv::Mat response;
    switch (metric_type_)
    {
    case CornerMetric::harris:
      response = harrisMetric(A, B, C); break;

    case CornerMetric::harmonic_mean:
      response = harmonicMeanMetric(A, B, C); break;

    case CornerMetric::min_eigen:
      response = minEigenMetric(A, B, C); break;
    }

    // Todo 5: Dilate image to make each pixel equal to the maximum in the neighborhood.
    cv::Mat local_max;

    // Todo 6: Compute the threshold.
    // Compute the threshold by using quality_level_ on the maximum response.
    double max_val = 10.0;

    // Todo 7: Extract local maxima above threshold.
    cv::Mat is_strong_and_local_max; // = response > threshold and response == local_max
    std::vector<cv::Point> max_locations;

    // ----- Now detect() is finished! -----
    // Add all strong local maxima as keypoints.
    const float keypoint_size = 3.0f * window_sigma_;
    std::vector<cv::KeyPoint> key_points;
```

```
    for (const auto& point : max_locations)
    {
        key_points.emplace_back(cv::KeyPoint{point,                     keypoint_size,                   -1,
response.at<float>(point)});
    }

    // Show additional debug/educational figures.
    if (do_visualize_)
    {
        if (!Ix.empty()) { cv::imshow("Gradient Ix", Ix); };
        if (!Iy.empty()) { cv::imshow("Gradient Iy", Iy); };
        if (!A.empty()) { cv::imshow("Image A", A); };
        if (!B.empty()) { cv::imshow("Image B", B); };
        if (!C.empty()) { cv::imshow("Image C", C); };
        if (!response.empty()) { cv::imshow("Response", response/(0.9*max_val)); };
        if          (!is_strong_and_local_max.empty())          {          cv::imshow("Local          max",
is_strong_and_local_max); };
    }

    return key_points;
}

cv::Mat CornerDetector::harrisMetric(const cv::Mat& A, const cv::Mat& B, const cv::Mat& C)
const
{
    // Compute the Harris metric for each pixel.
    // Todo 4.1: Finish the Harris metric.
    const float alpha = 0.06f;

    return cv::Mat();
}

cv::Mat CornerDetector::harmonicMeanMetric(const cv::Mat& A, const cv::Mat& B, const
cv::Mat& C) const
{
    // Compute the Harmonic mean metric for each pixel.
    // Todo 4.2: Finish the Harmonic Mean metric.
    return cv::Mat();
}

cv::Mat CornerDetector::minEigenMetric(const cv::Mat& A, const cv::Mat& B, const cv::Mat& C)
const
{
    // Compute the Min. Eigen metric for each pixel.
```

```
    // Todo 4.3: Finish minimum eigenvalue metric.
    return cv::Mat();
}

#include "lab_corners.h"
#include "corner_detector.h"
#include <chrono>

// Make shorthand aliases for timing tools.
using Clock = std::chrono::high_resolution_clock;
using DurationInMs = std::chrono::duration<double, std::milli>;

void runLabCorners()
{
    // Open video stream from camera.
    const int camera_id = 0; // Should be 0 or 1 on the lab PCs.
    cv::VideoCapture cap(camera_id);
    cap.set(cv::CAP_PROP_FRAME_WIDTH, 640);
    cap.set(cv::CAP_PROP_FRAME_HEIGHT, 480);

    if (!cap.isOpened())
    {
        throw std::runtime_error("Could not open camera by index " + std::to_string(camera_id));
    }

    // Create window.
    const std::string win_name = "Lab: Estimating circles from corners";
    cv::namedWindow(win_name);

    // Construct the corner detector.
    // Play around with the parameters!
    // When the second argument is true, additional debug visualizations are shown.
    CornerDetector det(CornerMetric::harris, true);

    // Construct the circle estimator.
    CircleEstimator estimator;
    while (true)
    {
        // Read a frame from the camera.
        cv::Mat frame;
        cap >> frame;

        if (frame.empty())
        { break; }
```

```
        // Convert frame to gray scale image.
        cv::Mat gray_frame;
        cv::cvtColor(frame, gray_frame, cv::COLOR_BGR2GRAY);

        // Perform corner detection.
        // Measure how long the processing takes.
        auto start = Clock::now();
        std::vector<cv::KeyPoint> corners = det.detect(gray_frame);
        auto end = Clock::now();
        DurationInMs corner_proc_duration = end - start;

        // Keep the highest scoring points.
        const int num_to_keep = 1000;
        cv::KeyPointsFilter::retainBest(corners, num_to_keep);

        // Convert corners to Eigen points.
        Eigen::Matrix2Xf points = convertToPoints(corners);

        // Estimate circle from points.
        // Measure how long the processing takes.
        start = Clock::now();
        CircleEstimate estimate = estimator.estimate(points);
        end = Clock::now();
        DurationInMs circle_proc_duration = end - start;

        // Visualize the results.
        cv::Mat vis_img = frame.clone();
        drawCornerResult(vis_img, corners, corner_proc_duration.count());
        drawCircleResult(vis_img, corners, estimate, circle_proc_duration.count());
        cv::imshow(win_name, vis_img);
        if (cv::waitKey(30) >= 0) break;
    }
}

Eigen::Matrix2Xf convertToPoints(const std::vector<cv::KeyPoint>& keypoints)
{
    // Convert each OpenCV point to column vector in Eigen matrix.
    Eigen::Matrix2Xf points(2, keypoints.size());
    for (size_t i=0; i < keypoints.size(); ++i)
    {
        points.col(i) = Eigen::Vector2f(keypoints[i].pt.x, keypoints[i].pt.y);
    }
```

```
    return points;
}

void drawCornerResult(const cv::Mat& img, const std::vector<cv::KeyPoint>& keypoints, double
time)
{
    // Print processing duration.
    std::stringstream duration_info;
    duration_info << std::fixed << std::setprecision(0); // Set to 0 decimals.
    duration_info << "Corner time: " << time << "ms";
    cv::putText(img, duration_info.str(), {10, 20}, cv::FONT_HERSHEY_PLAIN, 1.0, {0, 255,
0});

    // Draw corners.
    cv::drawKeypoints(img, keypoints, img, cv::Scalar{0,255,0});
}

void drawCircleResult(const cv::Mat& img, const std::vector<cv::KeyPoint>& keypoints, const
CircleEstimate& estimate,
                                    double time)
{
    // Check if there is a result.
    if (estimate.num_inliers == 0)
    {
        return;
    }

    // Print processing duration.
    std::stringstream duration_info;
    duration_info << std::fixed << std::setprecision(0); // Set to 0 decimals.
    duration_info << "Circle time: " << time << "ms";
    cv::putText(img, duration_info.str(), {10, 40}, cv::FONT_HERSHEY_PLAIN, 1.0, {0, 0,
255});

    // Extract and draw circle point inliers.
    std::vector<cv::KeyPoint> inlier_keypts;
    for (size_t i=0; i<keypoints.size(); ++i)
    {
        if (estimate.is_inlier(i))
        {
            inlier_keypts.push_back(keypoints[i]);
        }
    }
    cv::drawKeypoints(img, inlier_keypts, img, cv::Scalar{0,0,255});
```

```
    // Draw estimated circle
    const Eigen::Vector2i center = estimate.circle.center().array().round().cast<int>();
    cv::Point center_point{center.x(), center.y()};
    int radius = static_cast<int>(std::round(estimate.circle.radius()));
    cv::circle(img, center_point, radius, cv::Scalar(0, 0, 255), cv::LINE_4, cv::LINE_AA);

    // Print some information about the estimation.
    std::stringstream iterations_info;
    iterations_info << "Iterations: " << estimate.num_iterations;
    cv::putText(img, iterations_info.str(), {10, 60}, cv::FONT_HERSHEY_PLAIN, 1.0, {0, 0,
255});

    std::stringstream inliers_info;
    inliers_info << "Inliers: " << estimate.num_inliers;
    cv::putText(img, inliers_info.str(), {10, 80}, cv::FONT_HERSHEY_PLAIN, 1.0, {0, 0, 255});
}
```