****

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

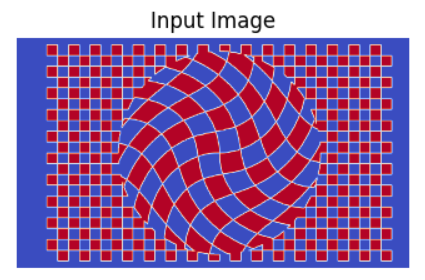
λmin, Harris, Harmonic Mean, other?

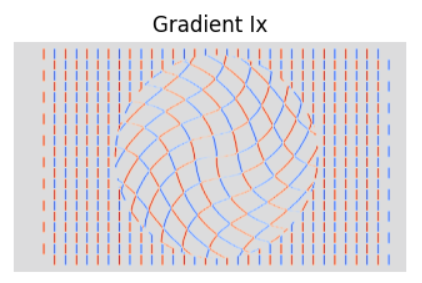
Threshold the corner metric image and find local maxima

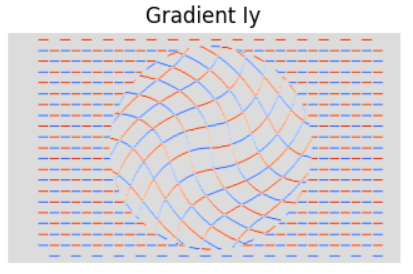
Morphological operations

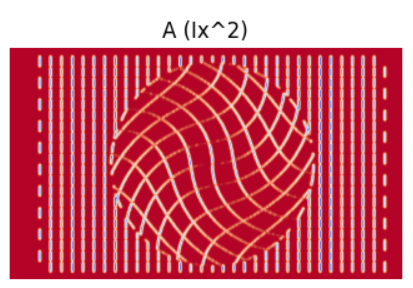
Logical operations

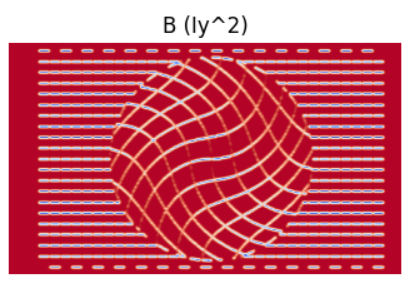
使用coolwarm\_r来进行更加清晰地可视化

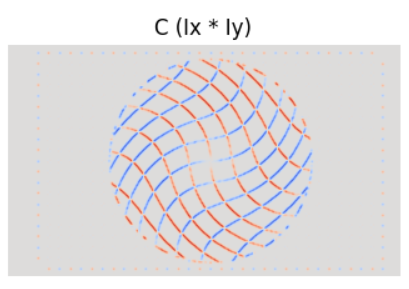


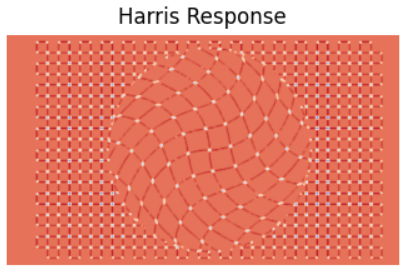


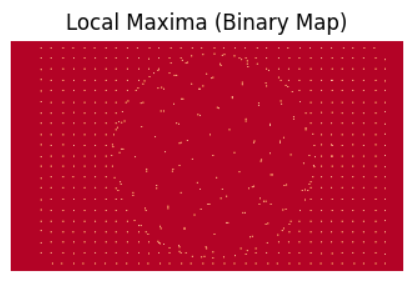


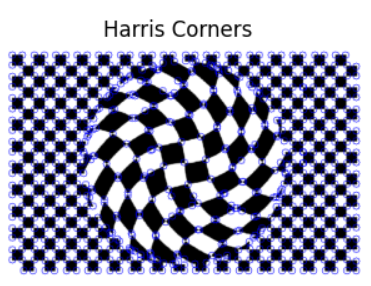


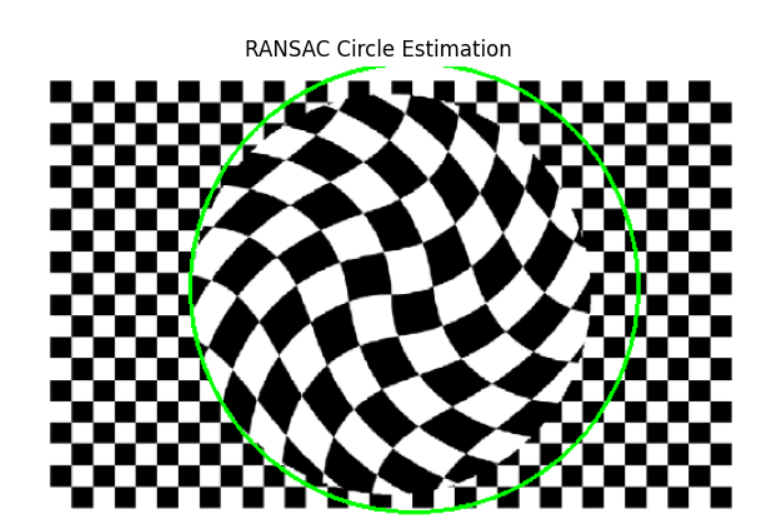












附件：代码

#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>();

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});

}