unscented_kalman_filter.h

Go to the documentation of this file.

```
/*
 * Copyright 2016 The Cartographer Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#ifndef CARTOGRAPHER_KALMAN_FILTER_UNSCENTED_KALMAN_FILTER_H_
#define CARTOGRAPHER_KALMAN_FILTER_UNSCENTED_KALMAN_FILTER_H_

#include <algorithm>
#include <cmath>
#include <functional>
#include <vector>

#include "Eigen/Cholesky"
#include "Eigen/Core"
#include "Eigen/Eigenvalues"
#include "cartographer/kalman_filter/gaussian_distribution.h"
#include "glog/logging.h"

namespace cartographer {
namespace kalman_filter {

template <typename FloatType>
constexpr FloatType sqr(FloatType a) {
  return a * a;
}

template <typename FloatType, int N>
Eigen::Matrix<FloatType, N, N> OuterProduct(
    const Eigen::Matrix<FloatType, N, 1>& v) {
  return v * v.transpose();
}

// Checks if 'A' is a symmetric matrix.
template <typename FloatType, int N>
void CheckSymmetric(const Eigen::Matrix<FloatType, N, N>& A) {
  // This should be pretty much Eigen::Matrix<>::Zero() if the matrix is
  // symmetric.
  const FloatType norm = (A - A.transpose()).norm();
  CHECK(!std::isnan(norm) && std::abs(norm) < 1e-5)
      << "Symmetry check failed with norm: '" << norm << "' from matrix:\n"
      << A;
}

// Returns the matrix square root of a symmetric positive semidefinite matrix.
template <typename FloatType, int N>
Eigen::Matrix<FloatType, N, N> MatrixSqrt(
    const Eigen::Matrix<FloatType, N, N>& A) {
  CheckSymmetric(A);

  Eigen::SelfAdjointEigenSolver<Eigen::Matrix<FloatType, N, N>>
```

```
      adjoint_eigen_solver((A + A.transpose()) / 2.);
  const auto& eigenvalues = adjoint_eigen_solver.eigenvalues();
  CHECK_GT(eigenvalues.minCoeff(), -1e-5)
      << "MatrixSqrt failed with negative eigenvalues: "
      << eigenvalues.transpose();

  return adjoint_eigen_solver.eigenvectors() *
         adjoint_eigen_solver.eigenvalues()
             .cwiseMax(Eigen::Matrix<FloatType, N, 1>::Zero())
             .cwiseSqrt()
             .asDiagonal() *
         adjoint_eigen_solver.eigenvectors().transpose();
}

// Implementation of a Kalman filter. We follow the nomenclature from
// Thrun, S. et al., Probabilistic Robotics, 2006.
//
// Extended to handle non-additive noise/sensors inspired by Kraft, E., A
// Quaternion-based Unscented Kalman Filter for Orientation Tracking.
template <typename FloatType, int N>
class UnscentedKalmanFilter {
 public:
  using StateType = Eigen::Matrix<FloatType, N, 1>;
  using StateCovarianceType = Eigen::Matrix<FloatType, N, N>;

  explicit UnscentedKalmanFilter(
      const GaussianDistribution<FloatType, N>& initial_belief,
      std::function<StateType(const StateType& state, const StateType& delta)>
          add_delta = [](const StateType& state,
                         const StateType& delta) { return state + delta; },
      std::function<StateType(const StateType& origin, const StateType& target)>
          compute_delta =
              [](const StateType& origin, const StateType& target) {
                return target - origin;
              })
      : belief_(initial_belief),
        add_delta_(add_delta),
        compute_delta_(compute_delta) {}

  // Does the control/prediction step for the filter. The control must be
  // implicitly added by the function g which also does the state transition.
  // 'epsilon' is the additive combination of control and model noise.
  void Predict(std::function<StateType(const StateType&)> g,
               const GaussianDistribution<FloatType, N>& epsilon) {
    CheckSymmetric(epsilon.GetCovariance());

    // Get the state mean and matrix root of its covariance.
    const StateType& mu = belief_.GetMean();
    const StateCovarianceType sqrt_sigma = MatrixSqrt(belief_.GetCovariance());

    std::vector<StateType> Y;
    Y.reserve(2 * N + 1);
    Y.emplace_back(g(mu));

    const FloatType kSqrtNPlusLambda = std::sqrt(N + kLambda);
    for (int i = 0; i < N; ++i) {
      // Order does not matter here as all have the same weights in the
      // summation later on anyways.
      Y.emplace_back(g(add_delta_(mu, kSqrtNPlusLambda * sqrt_sigma.col(i))));
      Y.emplace_back(g(add_delta_(mu, -kSqrtNPlusLambda * sqrt_sigma.col(i))));
    }
    const StateType new_mu = ComputeMean(Y);

    StateCovarianceType new_sigma =
        kCovWeight0 * OuterProduct<FloatType, N>(compute_delta_(new_mu, Y[0]));
    for (int i = 0; i < N; ++i) {
      new_sigma += kCovWeightI * OuterProduct<FloatType, N>(
                                     compute_delta_(new_mu, Y[2 * i + 1]));
      new_sigma += kCovWeightI * OuterProduct<FloatType, N>(
                                     compute_delta_(new_mu, Y[2 * i + 2]));
    }
    CheckSymmetric(new_sigma);

    belief_ = GaussianDistribution<FloatType, N>(new_mu, new_sigma) + epsilon;
  }
```

```

  // The observation step of the Kalman filter. 'h' transfers the state
  // into an observation that should be zero, i.e., the sensor readings should
  // be included in this function already. 'delta' is the measurement noise and
  // must have zero mean.
  template <int K>
  void Observe(
      std::function<Eigen::Matrix<FloatType, K, 1>(const StateType&)> h,
      const GaussianDistribution<FloatType, K>& delta) {
    CheckSymmetric(delta.GetCovariance());
    // We expect zero mean delta.
    CHECK_NEAR(delta.GetMean().norm(), 0., 1e-9);

    // Get the state mean and matrix root of its covariance.
    const StateType& mu = belief_.GetMean();
    const StateCovarianceType sqrt_sigma = MatrixSqrt(belief_.GetCovariance());

    // As in Kraft's paper, we compute W containing the zero-mean sigma points,
    // since this is all we need.
    std::vector<StateType> W;
    W.reserve(2 * N + 1);
    W.emplace_back(StateType::Zero());

    std::vector<Eigen::Matrix<FloatType, K, 1>> Z;
    Z.reserve(2 * N + 1);
    Z.emplace_back(h(mu));

    Eigen::Matrix<FloatType, K, 1> z_hat = kMeanWeight0 * Z[0];
    const FloatType kSqrtNPlusLambda = std::sqrt(N + kLambda);
    for (int i = 0; i < N; ++i) {
      // Order does not matter here as all have the same weights in the
      // summation later on anyways.
      W.emplace_back(kSqrtNPlusLambda * sqrt_sigma.col(i));
      Z.emplace_back(h(add_delta_(mu, W.back())));

      W.emplace_back(-kSqrtNPlusLambda * sqrt_sigma.col(i));
      Z.emplace_back(h(add_delta_(mu, W.back())));

      z_hat += kMeanWeightI * Z[2 * i + 1];
      z_hat += kMeanWeightI * Z[2 * i + 2];
    }

    Eigen::Matrix<FloatType, K, K> S =
        kCovWeight0 * OuterProduct<FloatType, K>(Z[0] - z_hat);
    for (int i = 0; i < N; ++i) {
      S += kCovWeightI * OuterProduct<FloatType, K>(Z[2 * i + 1] - z_hat);
      S += kCovWeightI * OuterProduct<FloatType, K>(Z[2 * i + 2] - z_hat);
    }
    CheckSymmetric(S);
    S += delta.GetCovariance();

    Eigen::Matrix<FloatType, N, K> sigma_bar_xz =
        kCovWeight0 * W[0] * (Z[0] - z_hat).transpose();
    for (int i = 0; i < N; ++i) {
      sigma_bar_xz +=
          kCovWeightI * W[2 * i + 1] * (Z[2 * i + 1] - z_hat).transpose();
      sigma_bar_xz +=
          kCovWeightI * W[2 * i + 2] * (Z[2 * i + 2] - z_hat).transpose();
    }

    const Eigen::Matrix<FloatType, N, K> kalman_gain =
        sigma_bar_xz * S.inverse();
    const StateCovarianceType new_sigma =
        belief_.GetCovariance() - kalman_gain * S * kalman_gain.transpose();
    CheckSymmetric(new_sigma);

    belief_ = GaussianDistribution<FloatType, N>(
        add_delta_(mu, kalman_gain * -z_hat), new_sigma);
  }

  const GaussianDistribution<FloatType, N>& GetBelief() const {
    return belief_;
  }

 private:
```

```
  StateType ComputeWeightedError(const StateType& mean_estimate,
                                 const std::vector<StateType>& states) {
    StateType weighted_error =
        kMeanWeight0 * compute_delta_(mean_estimate, states[0]);
    for (int i = 1; i != 2 * N + 1; ++i) {
      weighted_error += kMeanWeightI * compute_delta_(mean_estimate, states[i]);
    }
    return weighted_error;
  }

  // Algorithm for computing the mean of non-additive states taken from Kraft's
  // Section 3.4, adapted to our implementation.
  StateType ComputeMean(const std::vector<StateType>& states) {
    CHECK_EQ(states.size(), 2 * N + 1);
    StateType current_estimate = states[0];
    StateType weighted_error = ComputeWeightedError(current_estimate, states);
    int iterations = 0;
    while (weighted_error.norm() > 1e-9) {
      double step_size = 1.;
      while (true) {
        const StateType next_estimate =
            add_delta_(current_estimate, step_size * weighted_error);
        const StateType next_error =
            ComputeWeightedError(next_estimate, states);
        if (next_error.norm() < weighted_error.norm()) {
          current_estimate = next_estimate;
          weighted_error = next_error;
          break;
        }
        step_size *= 0.5;
        CHECK_GT(step_size, 1e-3) << "Step size too small, line search failed.";
      }
      ++iterations;
      CHECK_LT(iterations, 20) << "Too many iterations.";
    }
    return current_estimate;
  }

  // According to Wikipedia these are the normal values. Thrun does not
  // mention those.
  constexpr static FloatType kAlpha = 1e-3;
  constexpr static FloatType kKappa = 0.;
  constexpr static FloatType kBeta = 2.;
  constexpr static FloatType kLambda = sqr(kAlpha) * (N + kKappa) - N;
  constexpr static FloatType kMeanWeight0 = kLambda / (N + kLambda);
  constexpr static FloatType kCovWeight0 =
      kLambda / (N + kLambda) + (1. - sqr(kAlpha) + kBeta);
  constexpr static FloatType kMeanWeightI = 1. / (2. * (N + kLambda));
  constexpr static FloatType kCovWeightI = kMeanWeightI;

  GaussianDistribution<FloatType, N> belief_;
  const std::function<StateType(const StateType& state, const StateType& delta)>
      add_delta_;
  const std::function<StateType(const StateType& origin,
                                const StateType& target)>
      compute_delta_;
};

template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kAlpha;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kKappa;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kBeta;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kLambda;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kMeanWeight0;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kCovWeight0;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kMeanWeightI;
template <typename FloatType, int N>
constexpr FloatType UnscentedKalmanFilter<FloatType, N>::kCovWeightI;
```

```
}  // namespace kalman_filter
}  // namespace cartographer

#endif  // CARTOGRAPHER_KALMAN_FILTER_UNSCENTED_KALMAN_FILTER_H_
```

---

cartographer

Author(s):

autogenerated on Wed Jun 5 2019 21:57:59