UDACITY

‹ Return to "Self-Driving Car Engineer" in the classroom

DISCUSS ON STUDENT HUB

Unscented Kalman Filters

REVIEW

CODE REVIEW 6

HISTORY

▾ src/ukf.cpp 6

```
#include "ukf.h"
#include "Eigen/Dense"
#include <math.h>
#include <iostream>

using namespace std;
using Eigen::MatrixXd;
using Eigen::VectorXd;
using std::vector;

/**
 * Initializes Unscented Kalman filter
 * This is scaffolding, do not modify
 */
UKF::UKF() {
  // if this is false, laser measurements will be ignored (except during init)
  use_laser_ = true;

  // if this is false, radar measurements will be ignored (except during init)
  use_radar_ = true;

  // initial state vector
  x_ = VectorXd(5);

  // initial covariance matrix
  P_ = MatrixXd(5, 5);
```

```
// Process noise standard deviation longitudinal acceleration in m/s^2
std_a_ = 2;
```

AWESOME

Good job tuning the `Longitudinal Acceleration Parameter`! Other students have used values up to `3

```

// Process noise standard deviation yaw acceleration in rad/s^2
std_yawdd_ = 1;
```

AWESOME

Good job tuning the `Yaw Acceleration Noise Parameter`.Seems you have a good intuition and chose a

```

//DO NOT MODIFY measurement noise values below these are provided by the sen
// Laser measurement noise standard deviation position1 in m
std_laspx_ = 0.15;

// Laser measurement noise standard deviation position2 in m
std_laspy_ = 0.15;

// Radar measurement noise standard deviation radius in m
std_radr_ = 0.3;

// Radar measurement noise standard deviation angle in rad
std_radphi_ = 0.03; // Original value.

// Radar measurement noise standard deviation radius change in m/s
std_radrd_ = 0.3; // Original value.
//DO NOT MODIFY measurement noise values above these are provided by the sen

// state vector: [pos1 pos2 vel_abs yaw_angle yaw_rate] in SI units and rad
x_ << 0.0, 0.0, 0.0, 0.0, 0.0;

 ///* state covariance matrix
P_ << 10.0, 0.0, 0.0, 0.0, 0.0,
      0.0, 10.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 10.0, 0.0, 0.0,
      0.0, 0.0, 0.0, 10.0, 0.0,
      0.0, 0.0, 0.0, 0.0, 10.0;

time_us_ = 0;
is_initialized_ = false;

///* State dimension
n_x_ = 5;

// Augmented state dimension
n_aug_ = n_x_ + 2;

// Radar measurement dimension
n_z_ = 3;

```

```
  // Lidar measurement dimension
  n_l_ = 2;

  // Initializing measurement noise matrix R.
  R_radar_ = MatrixXd(n_z_, n_z_);
  R_radar_.fill(0.0);
  R_radar_(0, 0) = pow(std_radr_, 2);
  R_radar_(1, 1) = pow(std_radphi_, 2);
  R_radar_(2, 2) = pow(std_radrd_, 2);

  // Initializing measurement noise matrix R.
  R_lidar_ = MatrixXd(n_l_, n_l_);
  R_lidar_.fill(0.0);
  R_lidar_(0, 0) = pow(std_laspx_, 2);
  R_lidar_(1, 1) = pow(std_laspy_, 2);

  ///* Sigma point spreading parameter
  lambda_= 3 - static_cast<int>(n_aug_);

  // Predicted sigma points. Need to be shared between prediction and radar me
  Xsig_pred_ = MatrixXd(n_x_, 2 * n_aug_ + 1);

  // Create and set vector for weights of sigma points.
  weights_ = VectorXd(2 * n_aug_ + 1);
  weights_(0) = lambda_ / (lambda_ + n_aug_);
  float weight_other = 1 / (lambda_ + n_aug_) / 2;
  for (unsigned i = 1; i < 2 * n_aug_ + 1; ++i) {
      weights_(i) = weight_other;
  }
}

UKF::~UKF() {}

/**
 * @param {MeasurementPackage} meas_package The latest measurement data of
 * either radar or laser.
 */
void UKF::ProcessMeasurement(const MeasurementPackage & meas_package) {
  /***************************************************************************
   *  Initialization
   ***************************************************************************
  if (!is_initialized_) {
    // First measurement.
    cout << "UKF: initializing..." << endl;

    if (meas_package.sensor_type_ == MeasurementPackage::RADAR) {
      /**
      Convert radar from polar to Cartesian coordinates and initialize state.
      */
      float rho = meas_package.raw_measurements_[0];
      float phi = meas_package.raw_measurements_[1];
      //float delta_rho = meas_package.raw_measurements_[2];

      x_(0) = rho * cos(phi);
      x_(1) = rho * sin(phi);

      x_(2) = 0;
      x_(3) = 0;
      x_(4) = 0;

    }
```

```
    else if (meas_package.sensor_type_ == MeasurementPackage::LASER) {
      /**
      Initialize state.
      */
      x_(0) = meas_package.raw_measurements_[0];
      x_(1) = meas_package.raw_measurements_[1];

    }

    // Initializing time of the state vector.
    time_us_ = meas_package.timestamp_;
    // Done initializing, no need to predict or update.
    is_initialized_ = true;
    return;
  }

  /*****************************************************************************
   *  Prediction
   ****************************************************************************
  float delta_t = (meas_package.timestamp_ - time_us_) / 1000000.0;
  // Updating time-measurement state of KF.
  time_us_ = meas_package.timestamp_;

  // Updating state transition matrix based on elapsed time. For brevity, only
  // elements that depend on time.

  // Noise `ax` and `ay` are given as constants in this problem.
  // Updating process covariance matrix based on elapsed time. For readability
  Prediction(delta_t);

  /*****************************************************************************
   *  Update
   ****************************************************************************

  if (meas_package.sensor_type_ == MeasurementPackage::RADAR) {
    // Radar updates.
    UpdateRadar(meas_package);
  } else {
    // Laser updates.
    UpdateLidar(meas_package);
  }

  // Print the output of the state and its covariance.
  //cout << "x_ = " << x_ << endl;
  //cout << "P_ = " << P_ << endl;
}

/**
 * Predicts sigma points, the state, and the state covariance matrix.
 * @param {double} delta_t the change in time (in seconds) between the last
 * measurement and this one.
 */
void UKF::Prediction(double delta_t) {
  MatrixXd Xsig_aug;
  // Create sigma point matrix.
  Xsig_aug = MatrixXd(n_aug_, 2 * n_aug_ + 1);

  GenerateSigmaPoints(Xsig_aug);
  PredictSigmaPoints(Xsig_aug, Xsig_pred_, delta_t);
  PredictMeanAndCovariance(Xsig_pred_);
}
```

```

/**
 * Updates the state and the state covariance matrix using a laser measurement
 * @param {MeasurementPackage} meas_package
 */
void UKF::UpdateLidar(const MeasurementPackage & meas_package) {
```

AWESOME

Good job using the standard equations for the linear LIDAR case.

```

  VectorXd z(2);
  z(0) = meas_package.raw_measurements_[0];
  z(1) = meas_package.raw_measurements_[1];

  MatrixXd H_laser = MatrixXd(n_l_, n_x_);
  H_laser << 1.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 1.0, 0.0, 0.0, 0.0;

  VectorXd z_pred = H_laser * x_;
  VectorXd y = z - z_pred;
  MatrixXd S = H_laser * P_ * H_laser.transpose() + R_lidar_;
  MatrixXd PHt = P_ * H_laser.transpose();
  MatrixXd K = PHt * S.inverse();

  // Updating estimate.
  x_ = x_ + (K * y);
  MatrixXd I = MatrixXd::Identity(n_x_,n_x_);
  P_ = (I - K * H_laser) * P_;

  // Calculating NIS value
  float epsilon = y.transpose() * S.inverse() * y;
  cout << "Lidar NIS value=" << epsilon << endl;
}

/**
 * Updates the state and the state covariance matrix using a radar measurement
 * @param {MeasurementPackage} meas_package
 */
void UKF::UpdateRadar(const MeasurementPackage & meas_package) {

  VectorXd z(n_z_), z_pred(n_z_);
  float rho = meas_package.raw_measurements_[0];
  float phi = meas_package.raw_measurements_[1];
  float delta_rho = meas_package.raw_measurements_[2];
  z << rho, phi, delta_rho;

  // Predicted state covariance in the measurement space.
  MatrixXd S = MatrixXd(n_z_, n_z_);

  // Predicted sigma points in the radar measurement space.
  MatrixXd Zsig = MatrixXd(n_z_, 2 * n_aug_ + 1);

  PredictRadarMeasurement(Xsig_pred_, Zsig, z_pred, S);

  // Matrix for cross-correlation Tc
  MatrixXd Tc = MatrixXd(n_x_, n_z_);
  Tc.fill(0.0);
```

```

  for (unsigned i = 0; i < Zsig.cols(); ++i) {
    // Calculating state difference.
    VectorXd x_diff = Xsig_pred_.col(i) - x_;
    // Angle psi normalization to [-PI, PI].
    while (x_diff(3) > M_PI) x_diff(3) -= 2. * M_PI;
    while (x_diff(3) < -M_PI) x_diff(3) += 2. * M_PI;

    // Calculating Z-residual.
    VectorXd z_diff = Zsig.col(i) - z_pred;
    // Angle phi normalization to [-PI, PI].
    while (z_diff(1) > M_PI) z_diff(1) -= 2. * M_PI;
    while (z_diff(1) < -M_PI) z_diff(1) += 2. * M_PI;

    Tc += weights_(i) * x_diff * z_diff.transpose();
  }

  // Calculate Kalman gain K.
  MatrixXd K = MatrixXd(n_x_, n_x_);
  K.fill(0.0);
  K = Tc * S.inverse();

  // Calculate measurement residual.
  VectorXd z_diff = z - z_pred;

  // Angle phi normalization to [-PI, PI].
  while (z_diff(1) > M_PI) z_diff(1) -= 2. * M_PI;
  while (z_diff(1) < -M_PI) z_diff(1) += 2. * M_PI;

  x_ = x_ + K * z_diff;
  P_ = P_ - K * S * K.transpose();

  // Calculating NIS value
  float epsilon = z_diff.transpose() * S.inverse() * z_diff;
  cout << "Radar NIS value=" << epsilon << endl;
}

/**
 * Populates provided matrix with sigma points based on the current state and
 * @param pointer to MatrixD matrix that will store results.
 */
void UKF::GenerateSigmaPoints(MatrixXd & Xsig_aug) {
  // Create augmented mean vector.
  VectorXd x_aug  = VectorXd(n_aug_);
  x_aug.head(n_x_) = x_;
  x_aug(n_x_) = 0;
  x_aug(n_x_ + 1) = 0;

  // Create augmented covariance matrix.
  MatrixXd P_aug = MatrixXd(n_aug_, n_aug_);
  P_aug.fill(0.0);
  P_aug.topLeftCorner(n_x_, n_x_) = P_;
  P_aug(n_x_, n_x_) = std_a_ * std_a_;
  P_aug(n_x_ + 1, n_x_ + 1) = std_yawdd_ * std_yawdd_;
  // Calculate square root of P_aug.
  MatrixXd A = P_aug.llt().matrixL();
  // Actually placing the values inside the matrix.
  // 1. Assigning the mean vector as a first column.
  Xsig_aug.col(0)  = x_aug;
  // 2. Assigning the values of the second and third group of sigmas.
  for (unsigned i = 0; i < n_aug_; i++)
```

```
  {
    Xsig_aug.col(i + 1)     = x_aug + sqrt(lambda_ + n_aug_) * A.col(i);
    Xsig_aug.col(i + 1 + n_aug_) = x_aug - sqrt(lambda_ + n_aug_) * A.col(i);
  }

}

void UKF::PredictSigmaPoints(const MatrixXd & Xsig_aug, MatrixXd & Xsig_pred,
  // Set state and augmented vector dimensions.
  Xsig_pred.fill(0.0);

  // Predict sigma points.
  for (unsigned i = 0; i < 2* n_aug_ + 1; ++i) {
      // The following five items of the vector is CTRV model vector
      float px = Xsig_aug(0, i);
      float py = Xsig_aug(1, i);
      float v = Xsig_aug(2, i);
      float psi = Xsig_aug(3, i);
      float psi_dot = Xsig_aug(4, i);
      // The following two items are noise vector nu for longitudinal accelera
      float nu_a = Xsig_aug(5, i);
      float nu_psi = Xsig_aug(6, i);

      // Computing stochastic part common for both cases when psi is either 0
      float px_stochastic = pow(delta_t, 2) * cos(psi) * nu_a / 2;
      float py_stochastic = pow(delta_t, 2) * sin(psi) * nu_a / 2;
      float v_stochastic = delta_t * nu_a;
      float psi_stochastic = pow(delta_t, 2) * nu_psi / 2;
      float psi_dot_stochastic = delta_t * nu_psi;

      float px_gain = 0; // Initializing, re-defining below according to situa
      float py_gain = 0; // Initializing, re-defining below according to situa
      float v_gain = 0;
      float psi_gain = psi_dot * delta_t;
      float psi_dot_gain = 0;

      // Computing gain factor between two states. Avoiding division by zero.
      if (fabs(psi_dot) < 0.001) {
```

AWESOME

Good job checking for 0.

You could save the value `0.001` as a constant to enable the compiler to optimise the implementation.

If you want to optimise this even further, you can replace all constants in your code with MACROs.

Check out this post to learn more about MACROs.

```
          px_gain = v * cos(psi) * delta_t;
          py_gain = v * sin(psi) * delta_t;
      }
      else {
          px_gain = v / psi_dot * (sin(psi + psi_dot * delta_t)  - sin(psi));
          py_gain = v / psi_dot * (-cos(psi + psi_dot * delta_t)  + cos(psi));
      }

      Xsig_pred(0, i) = px + px_gain + px_stochastic;
      Xsig_pred(1, i) = py + py_gain + py_stochastic;
      Xsig_pred(2, i) = v + v_gain + v_stochastic;
      Xsig_pred(3, i) = psi + psi_gain + psi_stochastic;
```

```
      Xsig_pred(4, i) = psi_dot + psi_dot_gain + psi_dot_stochastic;
  }

}

void UKF::PredictMeanAndCovariance(const MatrixXd & Xsig_pred){
  // Create vector for predicted state.
  VectorXd x = VectorXd(n_x_);
  x.fill(0.0);

  // Create covariance matrix for prediction.
  MatrixXd P = MatrixXd(n_x_, n_x_);
  P.fill(0.0);

  // Predict state mean.
  for (unsigned i = 0; i < Xsig_pred.cols(); ++i) {
      x += weights_(i) * Xsig_pred.col(i);
  }

  // Predict state covariance matrix.
  for (unsigned i = 0; i < Xsig_pred.cols(); ++i) {
      VectorXd x_diff = Xsig_pred.col(i) - x;
      // Angle normalization.
      while (x_diff(3) > M_PI) x_diff(3) -= 2. * M_PI;
      while (x_diff(3) <-M_PI) x_diff(3) += 2. * M_PI;
      P += weights_(i) * x_diff * x_diff.transpose();
  }

  // Write result to existing state and covariance variables.
  x_ = x;
  P_ = P;
}

void UKF::PredictRadarMeasurement(const MatrixXd & Xsig_pred, MatrixXd & Zsig,
  // Initializing measurement space sigma points matrix.
  Zsig.fill(0.0);
  // Transform predicted sigma points into measurement space.
  for (unsigned i = 0; i < Zsig.cols(); ++i) {
    VectorXd z_measurement = VectorXd(n_z_);
    float px = Xsig_pred(0, i);
    float py = Xsig_pred(1, i);
    float v = Xsig_pred(2, i);
    float yaw = Xsig_pred(3, i);

    // Calculating rho.
    z_measurement(0) = sqrt(pow(px, 2) + pow(py, 2));
    // Calculating phi.
    z_measurement(1) = atan2(py, px);
```

SUGGESTION

Be aware that `atan2(0.0,0.0)` is undefined. You would want to prevent this case in a robust industry so

```
    // Calculating rho_dot.
    z_measurement(2) = (px * cos(yaw) * v + py * sin(yaw) * v) / z_measurement
```

SUGGESTION

> You should make sure, that a zero measurement does not result in a divide by zero here. You could do soı small eps.

```
    Zsig.col(i) = z_measurement;

  }

  // Calculate predicted mean measurement vector.
  z_pred.fill(0.0);
  for (unsigned i = 0; i < Zsig.cols(); ++i){
    z_pred += weights_(i) * Zsig.col(i);
  }

  // Calculate measurement covariance matrix S.
  S.fill(0.0);
  for (unsigned i = 0; i < Zsig.cols(); ++i) {
    VectorXd z_diff = Zsig.col(i) -z_pred;
    // Angle normalization for phi.
    while (z_diff(1) > M_PI) z_diff(1) -= 2. * M_PI;
    while (z_diff(1) < -M_PI) z_diff(1) += 2. * M_PI;
    S += weights_(i) * z_diff * z_diff.transpose();
  }

  // Obtaining updated covariance matrix.
  S += R_radar_;

}

```

- src/ukf.h
- src/tools.h
- src/tools.cpp
- src/stdout.txt
- src/radar_NIS.txt
- src/plot_NIS.py
- src/measurement_package.h
- src/main.cpp
- src/lidar_NIS.txt
- src/CMakeLists.txt

- readme.txt
- install-ubuntu.sh
- install-mac.sh
- cmakepatch.txt
- README.md
- LICENSE
- CMakeLists.txt

RETURN TO PATH

Rate this review