Extended Kalman Filter Implementation

The required changes to implement extended Kalman filter are done by editing three files:

1. Fusion EKF.cpp

   Initialization is done first(R_,P_,F_),then data processing

```
is_initialized_ = false;

previous_timestamp_ = 0;

// initializing matrices
R_laser_ = MatrixXd(2, 2);
R_radar_ = MatrixXd(3, 3);
H_laser_ = MatrixXd(2, 4);
ekf_.F_ = MatrixXd(4, 4);
ekf_.P_ = MatrixXd(4,4);
ekf_.Q_ = MatrixXd(4, 4);
Hj_ = MatrixXd(3, 4);

//measurement covariance matrix - laser
R_laser_ << 0.0225, 0,
            0, 0.0225;

//measurement covariance matrix - radar
R_radar_ << 0.09, 0, 0,
            0, 0.0009, 0,
            0, 0, 0.09;

/**
 * TODO: Finish initializing the FusionEKF.
 * TODO: Set the process and measurement noises
 */

// measurement matrix
H_laser_ << 1, 0, 0, 0,
            0, 1, 0, 0;

ekf_.P_ << 1, 0, 0, 0,
           0, 1, 0, 0,
           0, 0, 1000, 0,
           0, 0, 0, 1000;

// state transition matrix
ekf_.F_ << 1, 0, 1, 0,
           0, 1, 0, 1,
           0, 0, 1, 0,
           0, 0, 0, 1;
noise_ax = 9;
noise_ay = 9;
```

```
  /**
   * Initialization
   */
  if (!is_initialized_) {
    /**
     * TODO: Initialize the state ekf_.x_ with the first measurement.
     * TODO: Create the covariance matrix.
     * You'll need to convert radar from polar to cartesian coordinates.
     */


    previous_timestamp_ = measurement_pack.timestamp_;
    // first measurement
    cout << "EKF: " << endl;
    ekf_.x_ = VectorXd(4);
    ekf_.x_ << 1, 1, 1, 1;

    if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
      // TODO: Convert radar from polar to cartesian coordinates
      //         and initialize state.

      float rho = measurement_pack.raw_measurements_[0];
      float phi = measurement_pack.raw_measurements_[1];
      float rho_dot = measurement_pack.raw_measurements_[2];
      float x = rho * cos(phi);
      float y = rho * sin(phi);

      float vx = rho_dot * cos(phi);
      float vy = rho_dot * sin(phi);
      ekf_.x_ << x,y, vx,vy;


    }
    else if (measurement_pack.sensor_type_ == MeasurementPackage::LASER) {
      // TODO: Initialize state.


      ekf_.x_ << measurement_pack.raw_measurements_[0], measurement_pack.raw_measurements_[1], 1, 1;

    }

    is_initialized_ = true;
    return;
```

   Prediction Step

```
  /**
   * Prediction
   */

  /**
   * TODO: Update the state transition matrix F according to the new elapsed time.
   * Time is measured in seconds.
   * TODO: Update the process noise covariance matrix.
   * Use noise_ax = 9 and noise_ay = 9 for your Q matrix.
   */

  double dt = (measurement_pack.timestamp_ - previous_timestamp_) / 1000000.0;
  previous_timestamp_ = measurement_pack.timestamp_;

  ekf_.F_(0, 2) = dt;
  ekf_.F_(1, 3) = dt;
  //TODO: YOUR CODE HERE
  float dt_2 = dt * dt;
  float dt_3 = dt_2 * dt;
  float dt_4 = dt_3 * dt;

  ekf_.Q_ << dt_4/4*noise_ax, 0, dt_3/2*noise_ax, 0,
             0, dt_4/4*noise_ay, 0, dt_3/2*noise_ay,
             dt_3/2*noise_ax, 0, dt_2*noise_ax, 0,
             0, dt_3/2*noise_ay, 0, dt_2*noise_ay;

  ekf_.Predict();
```

Update step:

Update is based if I have Lidar or radar measurements.

```
/**
 * Update
 */

/**
 * TODO:
 * - Use the sensor type to perform the update step.
 * - Update the state and covariance matrices.
 */

if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
    // TODO: Radar updates

    Hj_ = tools.CalculateJacobian(ekf_.x_);
    ekf_.Init(ekf_.x_, ekf_.P_, ekf_.F_, Hj_, R_radar_, ekf_.Q_);


    //ekf_.H_ =Hj_;
    //ekf_.R_ = R_radar_;
    ekf_.UpdateEKF(measurement_pack.raw_measurements_);

} else {
    // TODO: Laser updates
    ekf_.Init(ekf_.x_, ekf_.P_, ekf_.F_, H_laser_, R_laser_,ekf_.Q_);
    //ekf_.H_ = H_laser_;
    //ekf_.R_ = R_laser_;
    ekf_.Update(measurement_pack.raw_measurements_);

}
// print the output
cout << "x_ = " << ekf_.x_ << endl;
cout << "P_ = " << ekf_.P_ << endl;
```

2. Kalman_filter.cpp

In this file, Predict, update, and updateEKF are implemented.

```
void KalmanFilter::Predict() {
  /**
   * TODO: predict the state
   */
    x_ = F_ * x_;
    MatrixXd Ft = F_.transpose();
    P_ = F_ * P_ * Ft + Q_;

}
```

```
void KalmanFilter::Update(const VectorXd &z) {
  /**
   * TODO: update the state by using Kalman Filter equations
   */
  z_pred = H_ * x_;
  y = z - z_pred;
  Ht = H_.transpose();
  S = H_ * P_ * Ht + R_;
  Si = S.inverse();
  K = P_ * Ht * Si;

    //new estimate
  x_ = x_ + (K * y);
  long x_size = x_.size();
  MatrixXd I = MatrixXd::Identity(x_size, x_size);
  P_ = (I - K * H_) * P_;
}
```

```
void KalmanFilter::UpdateEKF(const VectorXd &z) {
  /**
   * TODO: update the state by using Extended Kalman Filter equations
   */

    hx = VectorXd(3);

    double px = x_(0);
    double py = x_(1);
    double px2 = pow(x_(0), 2);
    double py2 = pow(x_(1), 2);
    double theta = atan2(py,px);

    hx << sqrt(px2 + py2), theta, ((x_(0) * x_(2)) + (x_(1) * x_(3))) / sqrt(px2 + py2);

    y = z - hx;

    while(y(1)>M_PI || y(1) <-1*M_PI)
    {
      if (y(1)>M_PI)
      {
        y(1)-=M_PI;
      }
      if (y(1)<-M_PI)
      {
        y(1)+=M_PI;
      }
    }
  Ht = H_.transpose();
  S = H_ * P_ * Ht + R_;
  Si = S.inverse();
  K = P_ * Ht * Si;

  //new estimate
  x_ = x_ + (K * y);
  long x_size = x_.size();
  MatrixXd I = MatrixXd::Identity(x_size, x_size);
  P_ = (I - K * H_) * P_;

}
```

3. Tools.cpp
   In this file, I implemented the Jacobian and RMSE same as in course notes.

Please find the behavior in case of running on DataSet1 and DataSet2