**Terence Henriod**

**CS 791X: Mobile Sensor Networks**

**Instructor: Jim La**

**Project 01: Kalman Filter**

**Abstract**

In this assingment, the Kalman Filter is explored. The Kalman Filtering technique is applied to data collected from driving a robot around a circular path in order to produce enhanced localization results. Use of varying variances are used to explore the effect of the Kalman Filter.

**Introduction**

The Kalman Filter is a tool that is suited for removing the noise from noisy data, often producing estimates that are more accurate than those produced by individual sensors. It is easy to implement and relies on linear models of the systems the technique it is applied to, making the technique very popular. The Kalman Filter is regularly applied in many fields, ranging from guidance and navigation, signal processing, and econometrics.

The Kalman Filter operates by first making a prediction as to what the state of the phenomenon being modeled is and what the expected observations will be, based on the models of the world provided to the Filter. After the prediction step, the estimate is updated based on the difference between the estimated measurement and the actual measurement. How much of the difference that is taken into consideration is determined by a gain factor that is computed based on the error observed in previous estimates and the models of the noise expected in the state and the measurements of the state.In this way, prior knowledge of the state, measurements of the current state, and knowledge of how the system evolves can be combined for an effective estimate of the state.

**Kalman Filter Performance**

The Kalman Filter is able to outperform individual, unfiltered data because it is able to, in essence, find the best way to average a prediction of what measurements we expect to see with the measurements we actually receive along with information from the previous state in order to make a sound estimate of the true value of the current estimate.

Assuming the models that are provided to the filter are accurate (which can be a challenge in and of itself), the filter can produce a good estimate of the state of things at the next time step. However, the filter then uses the probability distributions of the noise for the state and the observations to determine which to trust more: the prediction or the measurements. The Kalman Filter does this by first using the error covariance estimate from the previous iteration and state error noise model to adjust the prediction of the covariance of the error for the current iteration. The filter then follows up by comparing the predicted error as it transformed to the state space with the error adjusted using the measurement error model (sometimes called the “Innovation Covariance”).

The fact that the Kalman Filter does all of this predicting and estimating using error information from the previous iteration of filtering means that the Kalman Filter is able to maintain some “momentum” in the case that measurement qualities suddenly decrease and keep the estimate close to what would be predicted by the natural model. It also works the other direction, in that if the last estimate was poor, the Kalman Filter can rapidly adjust its estimate in subsequent iterations.

**Initial Results**

The results of applying the basic Kalman Filter are depicted in the following figures. The filter was only used on GPS and IMU data, since odometry data tends to be error prone. Odometry data was included in the figures for comparison. As can be seen, the Kalman Filter is capable of providing marked improvement over sensor data for localization when sensors become noisy.

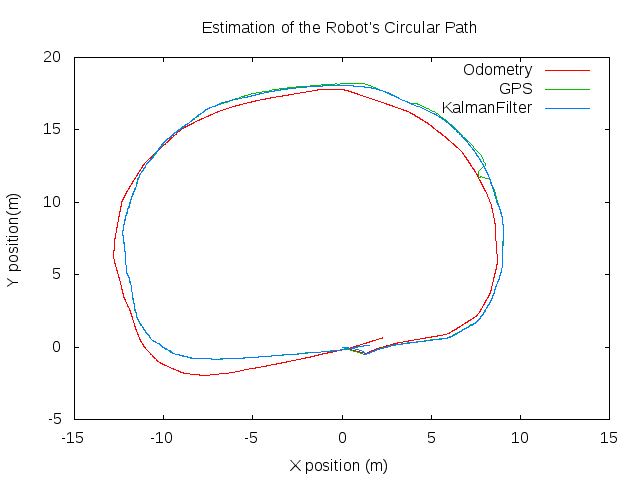


Figure 1: The Kalman Filter estimates demonstrate a behavior more true to the actual circular path of the robot.

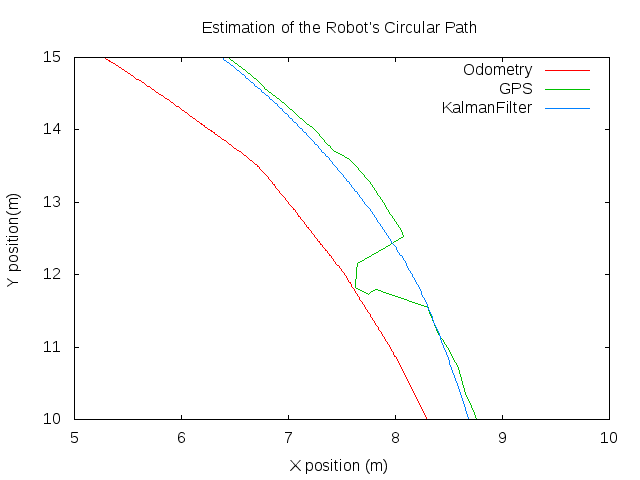


Figure 2: Even when the GPS data is disturbed and becomes very noisy (this particular figure shows an instance where GPS readings were disturbed by a nearby tree), the Kalman Filter can quickly adapt and produce accurate estimates that ignore the GPS information when it becomes noisy.

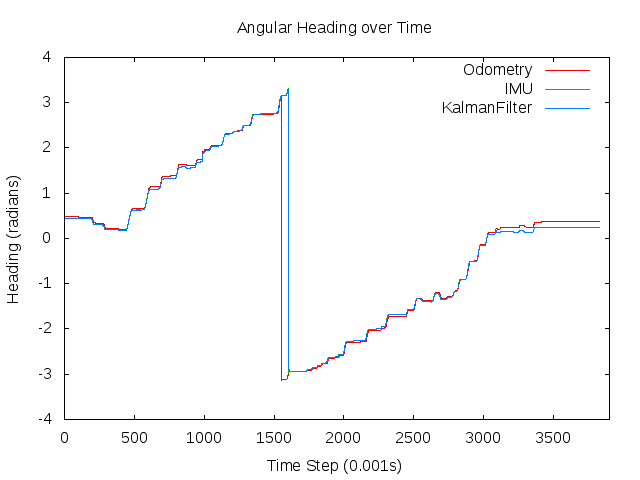


Figure 3: Here we can see that the Kalman Filter can remove noise from what appears to be a high quality sensor already. Note that the IMU data could be "calibrated" by simply adding a constant value to all of the measurement data, but was not to allow the graph lines to be seen more easily.

It should be noted that while the Kalman Filter that was implemented did estimate a state that included linear and angular velocity, but since these values were kept/assumed to be constant, they are not interesting to visualize. If the instructor needs these, they can be easily produced on request.

**Results After Data Modification**

Two variations to the state matrix were applied: dividing the variances by 10 in order to produce a filter that “trusts” the natural model even more than before, and one that multiplies the state error variances by 100 to produce a filter that relies heavily on input measurements. Note that the effects of changing the state error variances can also be produced by altering the measurement error variances (by applying the opposite type of transformation).

*Results from a Filter with Variances Multiplied by 0.1*

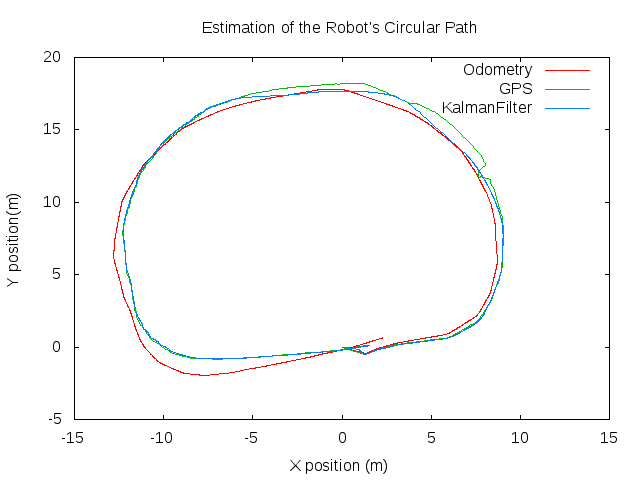


Figure 4: With a filter that assumes that the state error is even smaller, we can see how the estimate ignores the input measurements even more and produces a path that is probably somewhat inaccurate. Notice how the estimate is a shape that is much closer to an oval than what is likely the true path.

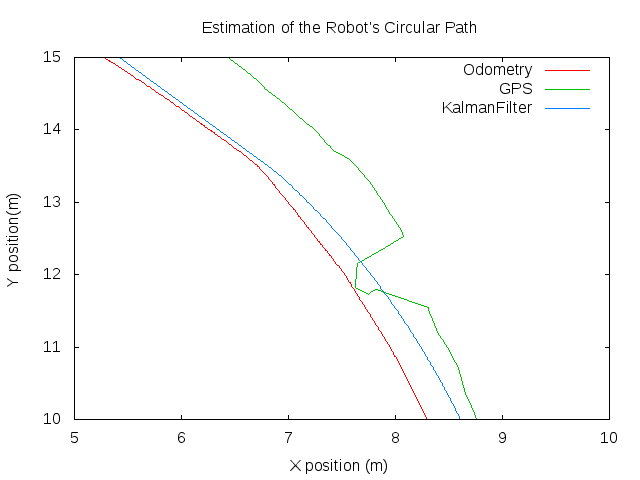


Figure 5: When observed more closely, we can see in better detail how the filter has produced an estimate that all but ignores the input measurements. The estimate is not even swayed by the error in the GPS readings caused by a tree.

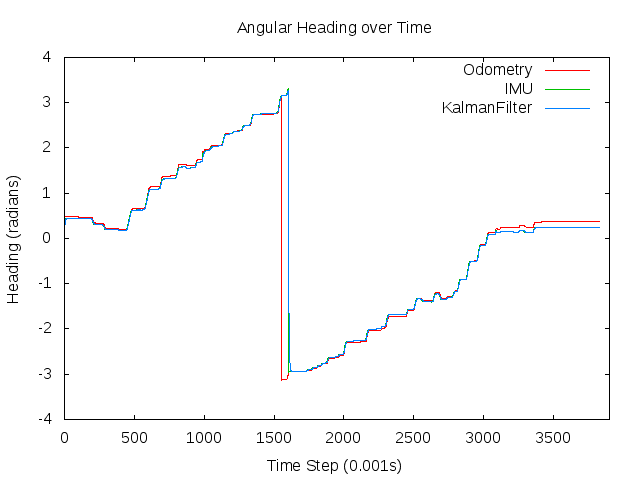


Figure 6: There was little difference between the heading estimate produced by the filter and the measurement from the IMU before, so weighting the “natural model” estimate more has very little effect.

*Results from a Filter with Variances Multiplied by 100*

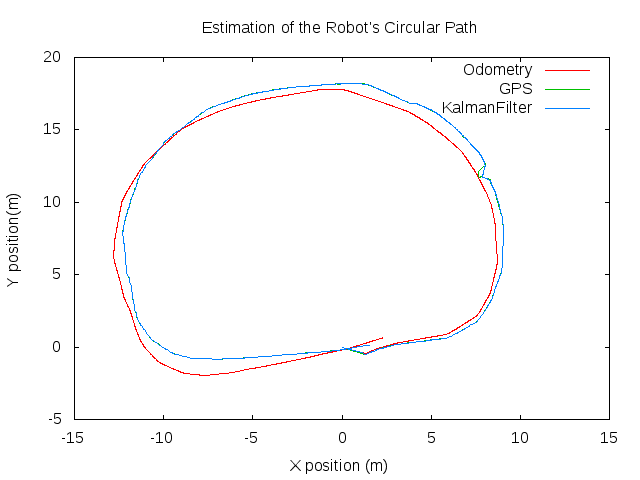


Figure 7: By increasing the state error variance, we can see that the filter will produce an estimate that favors measurements more strongly. In this figure it is clear that the estimated path follows the path produced by the measurements almost perfectly.

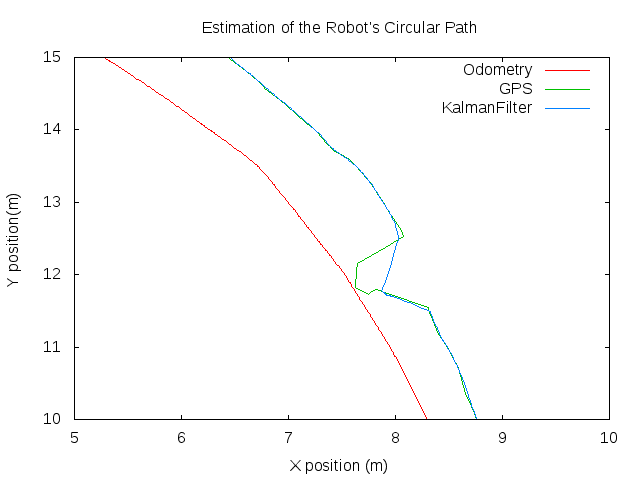


Figure 8: A closer look shows that the estimate produced by the filter is similar to the state of the world produced by the sensors in all but the most extreme cases.

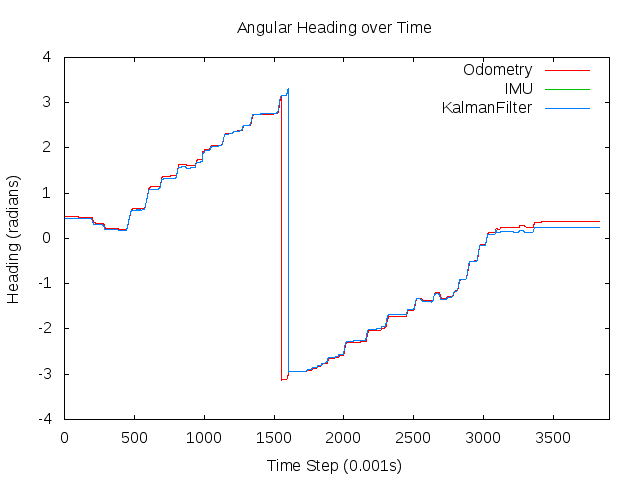


Figure 9: It may be difficult to discern the difference between this figure and the other ones, but by looking closely, it can be seen that the estimate is *very* similar to the heading read by the IMU.

Once again, the velocities were not included for consideration.

**Conclusion**

The Kalman Filter is a useful technique for data noise filtering and sensor fusion. It is relatively easy to implement, however it depends on properly defined models and carefully estimated distribution parameters for noise.

For the sake of brevity, much content was left out of this report. If more was desired, it is easily produced and available upon request.

It should be mentioned that the code found in the appendix was written to produce a Kalman Filter that would support observations with a different dimensionality than the estimated state. This feature was not utilized in this project however, because (1) we did not have a model that could support combining measurements from different sensors; and (2) nothing could have been gained from fusing odometry data (which is notoriously inadequate) with GPS or IMU data (which, given the particular devices used to collect this data, are very good), averaging bad data with good data produces a mediocre result. The observations used had the same dimensionality as the estimated state. However, in a situation where multiple sensors of the same caliber to measure the same variable are used, combining these data with a carefully constructed transition, or ‘H’, matrix and carefully modeled measurement noises could produce interesting results.

**Appendix: Code Used**

Compiling:

Simply type make in the project directory to build the program.

Dependencies: the Eigen Matrix Library (Makefile references this)

Output: KalmanFilter (executable)

Usage:

./KalmanFilter

-data <data file name>

-xy <the name of the xy position output data file (sans '.txt')>

-theta <the name of the theta output data file (sans '.txt')>

-v <the name of the linear velocity output data file (sans '.txt')>

-w <the name of the angular velocity output data file (sans '.txt')>

-cov <factor to adjust effects of state variance on filter>

Example:

./KalmanFilter -data SensorDataForKF.txt -xy xyPosition -theta heading -v linear\_velocity -w angular\_velocity -cov 1

Code:

*Driver Program*

#ifndef \_\_KALMANFILTER\_MAIN\_CPP\_\_

#define \_\_KALMANFILTER\_MAIN\_CPP\_\_

#include <cstring>

#include <string>

#include <cstdio>

#include <vector>

#include <cmath>

#include <iostream>

#include <fstream>

#include <iomanip>

#include <Eigen/Dense>

#include "KalmanFilter.hpp"

const int STATE\_DIMENSIONS = 5;

const int MEASUREMENT\_DIMENSIONS = 5;

const int NUM\_MEASUREMENTS = 6;

const double DELTA\_T = 0.001;

const double LINEAR\_VELOCITY = 0.14;

const double ANGULAR\_VELOCITY = LINEAR\_VELOCITY \* tan(0);

const double ARBITRARY\_VARIANCE = 0.1;

/\* These indices map the given data into a vector with the

following arrangement for processing:

\*/

Measurements: Covariances:

0 odometer.x Something arbitrary

1 GPS.x GPS.x covariance

2 odometer.y Something arbitrary

3 GPS.y GPS.y covariance

4 odometer.\theta Something arbitrary

5 IMU.\theta IMU.\theta covariance

\*/

const int ODOM\_X = 0;

const int ODOM\_Y = 2;

const int ODOM\_T = 4;

const int GPS\_X = 1;

const int GPS\_Y = 3;

const int IMU\_T = 5;

const int X\_ESTIMATE = 0;

const int Y\_ESTIMATE = 1;

const int V\_ESTIMATE = 2;

const int T\_ESTIMATE = 3;

const int W\_ESTIMATE = 4;

typedef struct {

std::string dataFilename;

std::string xyFilename;

std::string tFilename;

std::string vFilename;

std::string wFilename;

double stateCovarianceModifier;

} Configurations;

typedef struct {

Eigen::VectorXd measurements;

Eigen::VectorXd variances;

} DataItem;

Configurations processCmdLineArgs(int pArgc, char\*\* pArgs);

std::vector<DataItem> readDataFile(const std::string& dataFileName);

DataItem readDataLine(std::ifstream& fin);

std::vector<KalmanFilter::KalmanState> applyKalmanFilterToData(

Configurations& pConfigurations,

std::vector<DataItem>& pData);

Eigen::MatrixXd computeUpdatedNaturalModelMatrix(

int pStateDimensionality,

double pTimeStepMagnitude,

Eigen::VectorXd& pPreviousState);

Eigen::MatrixXd computeUpdatedStateNoiseCovariance(

int pStateDimensionality,

double pXPosCovariance,

double pYPosCovariance);

Eigen::MatrixXd computeUpdatedObservationNoiseCovariance(

int pObservationDimensionality,

double pXPosCovariance,

double pYPosCovariance,

double pTPosCovariance);

Eigen::VectorXd reduceDataToInputMeasurements(

Eigen::VectorXd& pMeasurementData,

int pNewVectorSize,

const std::vector<std::pair<int, int>>& pDataIndices);

void writeDataToFile(

std::string pFileName,

std::vector<DataItem>& pInputData,

const std::vector<std::string>& pDataColumnLabels,

const std::vector<int>& pDataIndices,

std::vector<KalmanFilter::KalmanState>& pKalmanOutputData,

const std::vector<std::string>& pOutputColumnLabels,

const std::vector<int>& pOutputIndices);

void testSimpleKalmanFilter();

int main(int argc, char\*\* argv) {

Configurations configurations;

try {

puts("Initializing program...");

configurations = processCmdLineArgs(argc, argv);

puts("Reading data...");

std::vector<DataItem> data = readDataFile(configurations.dataFilename);

puts("Applying KalmanFilter...");

std::vector<KalmanFilter::KalmanState> results =

applyKalmanFilterToData(configurations, data);

puts("Writing results to file...");

writeDataToFile(

configurations.xyFilename,

data,

{"Odom\_X", "Odom\_Y", "GPS\_X", "GPS\_Y"},

{ODOM\_X, ODOM\_Y, GPS\_X, GPS\_Y},

results,

{"X-estimate", "Y-estimate"},

{X\_ESTIMATE, Y\_ESTIMATE}

);

writeDataToFile(

configurations.tFilename,

data,

{"Odom\_T", "IMU\_T"},

{ODOM\_T, IMU\_T},

results,

{"T-estimate",},

{T\_ESTIMATE}

);

writeDataToFile(

configurations.vFilename,

data,

{},

{},

results,

{"v-estimate"},

{V\_ESTIMATE}

);

writeDataToFile(

configurations.wFilename,

data,

{},

{},

results,

{"w-estimate",},

{W\_ESTIMATE}

);

} catch (std::exception& e) {

puts("An error occured - program failure!");

puts(e.what());

}

return 0;

}

Configurations processCmdLineArgs(int pArgc, char\*\* pArgs) {

Configurations configurations;

if (pArgc < (3 + 1)) {

throw std::exception();

}

// TODO: use strtok for cleaner arg parsing

for (int i = 1; i < pArgc; i += 2) {

if (strcmp(pArgs[i], "-data") == 0) {

configurations.dataFilename = pArgs[i + 1];

} else if (strcmp(pArgs[i], "-xy") == 0) {

configurations.xyFilename = pArgs[i + 1];

} else if (strcmp(pArgs[i], "-theta") == 0) {

configurations.tFilename = pArgs[i + 1];

} else if (strcmp(pArgs[i], "-v") == 0) {

configurations.vFilename = pArgs[i + 1];

} else if (strcmp(pArgs[i], "-w") == 0) {

configurations.wFilename = pArgs[i + 1];

} else if (strcmp(pArgs[i], "-cov") == 0) {

configurations.stateCovarianceModifier = atof(pArgs[i + 1]);

}

}

return configurations;

}

std::vector<DataItem> readDataFile(const std::string& dataFileName) {

std::vector<DataItem> data;

try {

std::ifstream fin;

fin.clear();

fin.open(dataFileName.c\_str());

std::string dummyString;

int dummy;

char delimeter;

DataItem temp;

fin >> dummyString;

temp = readDataLine(fin);

while (fin.good()) {

data.push\_back(temp);

temp = readDataLine(fin);

}

fin.close();

} catch (std::exception& e) {

throw e;

}

return data;

}

DataItem readDataLine(std::ifstream& fin) {

DataItem dataItem;

dataItem.measurements.resize(NUM\_MEASUREMENTS);

dataItem.variances.resize(NUM\_MEASUREMENTS);

try {

int dummy;

char delimeter;

dataItem.measurements.setZero();

dataItem.variances.setConstant(NUM\_MEASUREMENTS, 1, ARBITRARY\_VARIANCE);

fin >>

dummy >> delimeter >>

dataItem.measurements(ODOM\_X) >> delimeter >> // odometer.x

dataItem.measurements(ODOM\_Y) >> delimeter >> // odometer.y

dataItem.measurements(ODOM\_T) >> delimeter >> // odometer.\theta

dataItem.measurements(IMU\_T) >> delimeter >> // IMU.\theta

dataItem.variances(IMU\_T) >> delimeter >> // IMU.\theta covariance

dataItem.measurements(GPS\_X) >> delimeter >> // GPS.x

dataItem.measurements(GPS\_Y) >> delimeter >> // GPS.y

dataItem.variances(GPS\_X) >> delimeter >> // GPS.x covariance

dataItem.variances(GPS\_Y); // GPS.y covariance

dataItem.measurements(IMU\_T) += 0.172; // calibration - just matches things to start

} catch (std::exception& e) {

throw e;

}

return dataItem;

}

std::vector<KalmanFilter::KalmanState> applyKalmanFilterToData(

Configurations& pConfigurations,

std::vector<DataItem>& pData) {

std::vector<KalmanFilter::KalmanState> results;

try {

KalmanFilter kalmanFilter(STATE\_DIMENSIONS, MEASUREMENT\_DIMENSIONS);

Eigen::VectorXd tempMeasurements(NUM\_MEASUREMENTS);

tempMeasurements = pData[0].measurements;

Eigen::VectorXd Z(MEASUREMENT\_DIMENSIONS);

Z = reduceDataToInputMeasurements(

pData[0].measurements,

MEASUREMENT\_DIMENSIONS,

{

{GPS\_X, X\_ESTIMATE},

{GPS\_Y, Y\_ESTIMATE},

{IMU\_T, T\_ESTIMATE}

}

);

Eigen::MatrixXd A(STATE\_DIMENSIONS, STATE\_DIMENSIONS);

A = computeUpdatedNaturalModelMatrix(

STATE\_DIMENSIONS,

DELTA\_T,

tempMeasurements

);

Eigen::MatrixXd B(STATE\_DIMENSIONS, STATE\_DIMENSIONS);

B.setIdentity(STATE\_DIMENSIONS, STATE\_DIMENSIONS);

Eigen::VectorXd u(STATE\_DIMENSIONS);

u.setZero();

Eigen::MatrixXd H(MEASUREMENT\_DIMENSIONS, STATE\_DIMENSIONS);

H.setIdentity(MEASUREMENT\_DIMENSIONS, STATE\_DIMENSIONS);

Eigen::MatrixXd R(MEASUREMENT\_DIMENSIONS, MEASUREMENT\_DIMENSIONS);

R = computeUpdatedObservationNoiseCovariance(MEASUREMENT\_DIMENSIONS, 0.1, 0.1, 0.01);

Eigen::MatrixXd Q(STATE\_DIMENSIONS, STATE\_DIMENSIONS);

Q = computeUpdatedStateNoiseCovariance(STATE\_DIMENSIONS, 0.00001, 0.00001);

Q \*= pConfigurations.stateCovarianceModifier;

KalmanFilter::KalmanState previousState;

previousState.state.resize(STATE\_DIMENSIONS, 1);

previousState.state.setZero();

previousState.errorCovariance.resize(STATE\_DIMENSIONS, STATE\_DIMENSIONS);

previousState.errorCovariance.setIdentity();

previousState.errorCovariance \*= 0.01;

kalmanFilter.setNaturalModel(A);

kalmanFilter.setControlModel(B);

kalmanFilter.setTransitionModel(H);

kalmanFilter.setStateNoiseCovariance(Q);

kalmanFilter.setMeasurementNoiseCovariance(R);

for (int i = 0; i < pData.size(); i++) {

A = computeUpdatedNaturalModelMatrix(

STATE\_DIMENSIONS,

DELTA\_T,

previousState.state

);

R = computeUpdatedObservationNoiseCovariance(

MEASUREMENT\_DIMENSIONS,

pData[i].variances(GPS\_X),

pData[i].variances(GPS\_Y),

pData[i].variances(IMU\_T)

);

Z = reduceDataToInputMeasurements(

pData[i].measurements,

MEASUREMENT\_DIMENSIONS,

{

{GPS\_X, X\_ESTIMATE},

{GPS\_Y, Y\_ESTIMATE},

{IMU\_T, T\_ESTIMATE}

}

);

kalmanFilter.setNaturalModel(A);

kalmanFilter.setMeasurementNoiseCovariance(R);

previousState = kalmanFilter.KalmanFilterIteration(previousState, Z, u);

results.push\_back(previousState);

}

} catch (std::exception& e) {

throw e;

}

return results;

}

Eigen::MatrixXd computeUpdatedNaturalModelMatrix(

int pStateDimensionality,

double pTimeStepMagnitude,

Eigen::VectorXd& pPreviousState) {

double previousAngle = pPreviousState(T\_ESTIMATE);

double previousAngularVelocity = pPreviousState(W\_ESTIMATE);

Eigen::MatrixXd A(pStateDimensionality, pStateDimensionality);

A << 1, 0, pTimeStepMagnitude \* cos(previousAngle), 0, 0,

0, 1, pTimeStepMagnitude \* sin(previousAngle), 0, 0,

0, 0, 1, 0, 0,

0, 0, 0, 1, pTimeStepMagnitude,

0, 0, 0, 0, 1;

return A;

}

Eigen::MatrixXd computeUpdatedStateNoiseCovariance(

int pStateDimensionality,

double pXPosCovariance,

double pYPosCovariance) {

Eigen::MatrixXd Q(pStateDimensionality, pStateDimensionality);

Q << pXPosCovariance, 0, 0, 0, 0,

0, pYPosCovariance, 0, 0, 0,

0, 0, 0.001, 0, 0,

0, 0, 0, 0.001, 0,

0, 0, 0, 0, 0.001;

return Q;

}

Eigen::MatrixXd computeUpdatedObservationNoiseCovariance(

int pObservationDimensionality,

double pXPosCovariance,

double pYPosCovariance,

double pTPosCovariance) {

Eigen::MatrixXd R(pObservationDimensionality, pObservationDimensionality);

R << pXPosCovariance, 0, 0, 0, 0,

0, pYPosCovariance, 0, 0, 0,

0, 0, 0.01, 0, 0,

0, 0, 0, pTPosCovariance, 0,

0, 0, 0, 0, 0.01;

return R;

}

Eigen::VectorXd reduceDataToInputMeasurements(

Eigen::VectorXd& pMeasurementData,

int pNewVectorSize,

const std::vector<std::pair<int, int>>& pDataIndices) {

Eigen::VectorXd newMeasurementVector(pNewVectorSize);

newMeasurementVector.setZero();

for (std::pair<int, int> indices : pDataIndices) {

newMeasurementVector(indices.second) = pMeasurementData(indices.first);

}

return newMeasurementVector;

}

void writeDataToFile(

std::string pFileName,

std::vector<DataItem>& pInputData,

const std::vector<std::string>& pDataColumnLabels,

const std::vector<int>& pDataIndices,

std::vector<KalmanFilter::KalmanState>& pKalmanOutputData,

const std::vector<std::string>& pOutputColumnLabels,

const std::vector<int>& pOutputIndices) {

try {

std::ofstream fout;

fout.clear();

fout.open(pFileName + ".txt");

std::string fileComment = "# ";

fileComment += pFileName;

fileComment += "data";

fout << fileComment << std::endl;

fileComment = "# time\t";

for (std::string label : pDataColumnLabels) {

fileComment += label;

fileComment += "\t";

}

for (std::string label : pOutputColumnLabels) {

fileComment += label;

fileComment += "\t";

}

fout << fileComment << std::endl;

for (int i = 0; i < pKalmanOutputData.size(); i++) {

fout << std::setw(5);

fout << i << '\t';

for (int index : pDataIndices) {

fout << std::setw(14);

fout << pInputData[i].measurements(index) << "\t";

}

for (int index : pOutputIndices) {

fout << std::setw(14);

fout << pKalmanOutputData[i].state(index) << "\t";

}

fout << std::endl;

}

fout.close();

} catch (std::exception& e) {

throw e;

}

}

void testSimpleKalmanFilter() {

Eigen::MatrixXd A(1,1); A << 1;

Eigen::MatrixXd B(1,1); B << 1;

Eigen::MatrixXd H(1,1); H << 1;

KalmanFilter::KalmanState state;

state.state.resize(1);

state.state << 0.0;

state.errorCovariance.resize(1,1);

state.errorCovariance << 1.0;

Eigen::VectorXd Z(1); Z << 0.390;

Eigen::VectorXd u(1); u << 0.0;

Eigen::MatrixXd Q(1,1); Q << 0.0;

Eigen::MatrixXd R(1,1); R << 0.1;

KalmanFilter kalmanFilter(1, 1);

kalmanFilter.setNaturalModel(A);

kalmanFilter.setControlModel(B);

kalmanFilter.setTransitionModel(H);

kalmanFilter.setStateNoiseCovariance(Q);

kalmanFilter.setMeasurementNoiseCovariance(R);

Z << 0.390;

state = kalmanFilter.KalmanFilterIteration(state, Z, u);

std::cout << "state\n" << state.state << std::endl <<

"P\n" << state.errorCovariance << std::endl << std::endl;

Z << 0.5;

state = kalmanFilter.KalmanFilterIteration(state, Z, u);

std::cout << "state\n" << state.state << std::endl <<

"P\n" << state.errorCovariance << std::endl << std::endl;

Z << 0.480;

state = kalmanFilter.KalmanFilterIteration(state, Z, u);

std::cout << "state\n" << state.state << std::endl <<

"P\n" << state.errorCovariance << std::endl << std::endl;

}

#endif //\_\_KALMANFILTER\_MAIN\_CPP\_\_

*Kalman Filter Class*

#ifndef \_\_KALMANFILTER\_HPP\_\_

#define \_\_KALMANFILTER\_HPP\_\_

#include <Eigen/Dense>

#include <utility>

class KalmanFilter {

public:

struct KalmanState {

Eigen::VectorXd state;

Eigen::MatrixXd errorCovariance;

};

KalmanFilter(

const int pStateDimensionality,

const int pMeasurementDimensionality);

void setNaturalModel(const Eigen::MatrixXd& pNewModel);

void setControlModel(const Eigen::MatrixXd& pNewModel);

void setTransitionModel(const Eigen::MatrixXd& pNewModel);

void setStateNoiseCovariance(const Eigen::MatrixXd& pNewCovariance);

void setMeasurementNoiseCovariance(const Eigen::MatrixXd& pNewCovariance);

KalmanState KalmanFilterIteration(

const KalmanState& pPreviousState,

const Eigen::MatrixXd& pMeasurementVector,

const Eigen::VectorXd& pControlVector);

Eigen::MatrixXd computeStatePrediction(

const Eigen::VectorXd& pPreviousState,

const Eigen::VectorXd& pControlVector);

Eigen::MatrixXd computeObservationPrediction(

const Eigen::VectorXd& pStatePrediction);

Eigen::MatrixXd computeErrorCovariancePrediction(

const Eigen::MatrixXd& pPreviousPredictionCovariance);

Eigen::MatrixXd computeKalmanGainFactor(

const Eigen::MatrixXd& pPredictionCovarianceEstimate);

Eigen::MatrixXd computeStateEstimate(

const Eigen::VectorXd& pStatePrediction,

const Eigen::MatrixXd& pKalmanGainFactor,

const Eigen::VectorXd& pMeasurementVector,

const Eigen::VectorXd& pObservationPrediction);

Eigen::MatrixXd computeErrorCovariance(

const Eigen::MatrixXd& pPredictionCovarianceEstimate,

const Eigen::MatrixXd& pKalmanGainFactor);

private:

int mStateDimensionality; // aka n

int mMeasurementDimensionality; // aka m

Eigen::MatrixXd mNaturalModel; // aka A; n x n; describes how the system

// evolves naturally, i.e. without controls

// or noise

Eigen::MatrixXd mControlModel; // aka B; n x n; describes how controls alter

// the system

Eigen::MatrixXd mTransitionModel; // aka H; m x n; describes how to map from

// a state to an observation

Eigen::VectorXd mStateNoise; // aka \epsilon; n x 1; has covariance Q

Eigen::MatrixXd mStateNoiseCovariance; // aka Q;

Eigen::VectorXd mMeasurementNoise; // aka \sigma; m x 1; has covariance R

Eigen::MatrixXd mMeasurementNoiseCovariance; // aka R; m x m

};

#endif //\_\_KALMANFILTER\_HPP\_\_

#ifndef \_\_KALMANFILTER\_CPP\_\_

#define \_\_KALMANFILTER\_CPP\_\_

#include "KalmanFilter.hpp"

#include <iostream>

KalmanFilter::KalmanFilter(

const int pStateDimensionality,

const int pMeasurementDimensionality) :

mStateDimensionality(pStateDimensionality),

mMeasurementDimensionality(pMeasurementDimensionality) {

mNaturalModel.resize(mStateDimensionality, mStateDimensionality);

mNaturalModel.setIdentity(mStateDimensionality, mStateDimensionality);

mControlModel.resize(mStateDimensionality, mStateDimensionality);

mControlModel.setIdentity(mStateDimensionality, mStateDimensionality);

mTransitionModel.resize(mMeasurementDimensionality, mStateDimensionality);

mTransitionModel.setIdentity(mMeasurementDimensionality, mStateDimensionality);

mStateNoise.resize(mStateDimensionality);

mStateNoise.setZero(mStateDimensionality);

mStateNoiseCovariance.resize(mStateDimensionality, mStateDimensionality);

mStateNoiseCovariance.setOnes(mStateDimensionality, mStateDimensionality);

mMeasurementNoise.resize(mMeasurementDimensionality);

mMeasurementNoise.setZero(mMeasurementDimensionality);

mMeasurementNoiseCovariance.resize(mMeasurementDimensionality, mMeasurementDimensionality);

mMeasurementNoiseCovariance.setOnes(mMeasurementDimensionality, mMeasurementDimensionality);

}

KalmanFilter::KalmanState KalmanFilter::KalmanFilterIteration(

const KalmanState& pPreviousState,

const Eigen::MatrixXd& pMeasurementVector,

const Eigen::VectorXd& pControlVector) {

// puts("X^hat prediction...");

Eigen::MatrixXd statePrediction =

computeStatePrediction(pPreviousState.state, pControlVector);

// puts("Y prediction...");

Eigen::MatrixXd observationPrediction =

computeObservationPrediction(statePrediction);

// puts("P^hat estimation...");

Eigen::MatrixXd predictionCovarianceEstimate =

computeErrorCovariancePrediction(pPreviousState.errorCovariance);

// puts("computing K...");

Eigen::MatrixXd kalmanGainFactor =

computeKalmanGainFactor(predictionCovarianceEstimate);

// puts("Final Estimation");

KalmanState currentState = {

computeStateEstimate(

statePrediction,

kalmanGainFactor,

pMeasurementVector,

observationPrediction

),

computeErrorCovariance(predictionCovarianceEstimate, kalmanGainFactor)

};

/\*

std::cout << "X^hat\n" << statePrediction << std::endl <<

"Y\n" <<observationPrediction << std::endl <<

"P^hat\n" <<predictionCovarianceEstimate << std::endl <<

"kalman gain\n" <<kalmanGainFactor << std::endl <<

"X\n" <<currentState.state << std::endl <<

"P\n" <<currentState.errorCovariance << std::endl;

\*/

return currentState;

}

Eigen::MatrixXd KalmanFilter::computeStatePrediction(

const Eigen::VectorXd& pPreviousState,

const Eigen::VectorXd& pControlVector) {

/\*

puts("debug");

std::cout << "A:\n" << mNaturalModel << std::endl <<

"X^hat:\n" << pPreviousState << std::endl <<

"B:\n" << mControlModel << std::endl <<

"u:\n" << pControlVector << std::endl <<

"e:\n" << mStateNoise << std::endl;

\*/

// X^hat = A \* X(k-1) + B \* u + \epsilon

return mNaturalModel \* pPreviousState +

mControlModel \* pControlVector +

mStateNoise;

}

Eigen::MatrixXd KalmanFilter::computeObservationPrediction(

const Eigen::VectorXd& pStatePrediction) {

// Y = H \* X^hat + \sigma

return mTransitionModel \* pStatePrediction + mMeasurementNoise;

}

Eigen::MatrixXd KalmanFilter::computeErrorCovariancePrediction(

const Eigen::MatrixXd& pPreviousPredictionCovariance) {

//std::cout << "=========\n" << mNaturalModel << "\n\*\n" << pPreviousPredictionCovariance << "\n\*\n" << mNaturalModel.transpose() << "\n+\n" << mMeasurementNoiseCovariance << "\n========\n";

// P^hat(t) = A \* P^hat(t - 1) \* A^t + Q

return

mNaturalModel \* pPreviousPredictionCovariance \* mNaturalModel.transpose() +

mStateNoiseCovariance;

}

Eigen::MatrixXd KalmanFilter::computeKalmanGainFactor(

const Eigen::MatrixXd& pPredictionCovarianceEstimate) {

// K = (P^hat \* H) / (H \* P^hat \* H^t + R)

Eigen::MatrixXd numerator =

pPredictionCovarianceEstimate \* mTransitionModel.transpose();

Eigen::MatrixXd denominator =

(mTransitionModel \* pPredictionCovarianceEstimate \* mTransitionModel.transpose()) +

mMeasurementNoiseCovariance;

/\*std::cout << "numerator: " << pPredictionCovarianceEstimate << ' ' << mTransitionModel.transpose() << " = " << numerator << std::endl <<

"denominator: " << mTransitionModel << " \* " << pPredictionCovarianceEstimate << " \* " << mTransitionModel.transpose() << " + " << mStateNoiseCovariance << " = " << denominator << std::endl;

\*/

return numerator \* denominator.inverse(); // inverse works? make sure

}

Eigen::MatrixXd KalmanFilter::computeStateEstimate(

const Eigen::VectorXd& pStatePrediction,

const Eigen::MatrixXd& pKalmanGainFactor,

const Eigen::VectorXd& pMeasurementVector,

const Eigen::VectorXd& pObservationPrediction) {

//std::cout << pStatePrediction << "\n + \n" << pKalmanGainFactor << "\n \* \n(" << pMeasurementVector << "\n - \n" << pObservationPrediction << ")\n" << std::endl;

// X = X^hat + K(Z - Y)

return pStatePrediction +

pKalmanGainFactor \* (pMeasurementVector - pObservationPrediction);

}

Eigen::MatrixXd KalmanFilter::computeErrorCovariance(

const Eigen::MatrixXd& pPredictionCovarianceEstimate,

const Eigen::MatrixXd& pKalmanGainFactor) {

// P = P^hat - K \* H \* P^hat or P = (I - K\*H) \* P^hat

return pPredictionCovarianceEstimate -

pKalmanGainFactor \* mTransitionModel \* pPredictionCovarianceEstimate;

}

void KalmanFilter::setNaturalModel(const Eigen::MatrixXd& pNewModel) {

assert(pNewModel.rows() == mStateDimensionality &&

pNewModel.cols() == mStateDimensionality);

mNaturalModel = pNewModel;

}

void KalmanFilter::setControlModel(const Eigen::MatrixXd& pNewModel) {

assert(pNewModel.rows() == mStateDimensionality &&

pNewModel.cols() == mStateDimensionality);

mControlModel = pNewModel;

}

void KalmanFilter::setTransitionModel(const Eigen::MatrixXd& pNewModel) {

mTransitionModel = pNewModel;

}

void KalmanFilter::setStateNoiseCovariance(const Eigen::MatrixXd& pNewCovariance) {

mStateNoiseCovariance.resize(pNewCovariance.rows(), pNewCovariance.cols());

mStateNoiseCovariance = pNewCovariance;

}

void KalmanFilter::setMeasurementNoiseCovariance(const Eigen::MatrixXd& pNewCovariance) {

mMeasurementNoiseCovariance.resize(pNewCovariance.rows(), pNewCovariance.cols());

mMeasurementNoiseCovariance = pNewCovariance;

}

#endif //\_\_KALMANFILTER\_CPP\_\_

Gnuplot Graph Plotting Scripts:

*plotXY.gp:*

plot [-15:15] [-5:20] "$0" using 2:3 title "Odometry" with lines, \

"$0" using 4:5 title "GPS" with lines, \

"$0" using 6:7 title "KalmanFilter" with lines;

set title "Estimation of the Robot's Circular Path";

set xlabel "X position (m)"; show xlabel;

set ylabel "Y position(m)"; show ylabel;

*plotXY\_zoom.gp:*

plot [5:10] [10:15] "$0" using 2:3 title "Odometry" with lines, \

"$0" using 4:5 title "GPS" with lines, \

"$0" using 6:7 title "KalmanFilter" with lines;

set title "Estimation of the Robot's Circular Path";

set xlabel "X position (m)"; show xlabel;

set ylabel "Y position(m)"; show ylabel;

*plotTheta.gp:*

plot [0:3900] [-4:4] "$0" using 2 title "Odometry" with lines, \

"$0" using 3 title "IMU" with lines, \

"$0" using 4 title "KalmanFilter" with lines;

set title "Angular Heading over Time";

set xlabel "Time Step (0.001s)"; show xlabel;

set ylabel "Heading (radians)"; show ylabel;

*replotToFile.gp:*

# This 'macro' comes from

# "Gnuplot in Action: Understanding Data Wth Graphs"

# by Philipp K. Janert

set terminal push # save the current terminal settings

set terminal png # change terminal to PNG

set output "$0" # set the output filename to the first option

replot # repeat the most recent plot command

set output # restore output to interactive mode

set terminal pop # restore the terminal