/*********************

- Software License Agreement (BSD License)
  *
- Copyright (c) 2008, Willow Garage, Inc.
- All rights reserved.
  *
- Redistribution and use in source and binary forms, with or without
- modification, are permitted provided that the following conditions
- are met:
  *
- – Redistributions of source code must retain the above copyright
- notice, this list of conditions and the following disclaimer.
- – Redistributions in binary form must reproduce the above
- copyright notice, this list of conditions and the following
- disclaimer in the documentation and/or other materials provided
- with the distribution.
- – Neither the name of the Willow Garage nor the names of its
- contributors may be used to endorse or promote products derived
- from this software without specific prior written permission.
  *
- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
- "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
- LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
- FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
- COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
- INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
- BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
- LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
- CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
- LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
- ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE

- POSSIBILITY OF SUCH DAMAGE.
  **********/

/* Author: Wim Meeussen */

#include <robot_pose_ekf/odom_estimation.h>

using namespace MatrixWrapper;
using namespace BFL;
using namespace tf;
using namespace std;
using namespace ros;

namespace estimation
{
 // constructor
 OdomEstimation::OdomEstimation() : prior(NULL),
                                    filter(NULL),
                                    filter_initialized(false),
                                    odom_initialized(false),
                                    imu_initialized(false),
                                    vo_initialized(false),
                                    gps_initialized(false),
                                    output_frame(std::string("odom_combined")),
                                    base_footprint_frame(std::string("base_footprint"))
 {
  // create SYSTEM MODEL
  ColumnVector sysNoise_Mu(6);
  sysNoise_Mu = 0;
  SymmetricMatrix sysNoise_Cov(6);
  sysNoise_Cov = 0;
  for (unsigned int i = 1; i <= 6; i++)
    sysNoise_Cov(i, i) = pow(1000.0, 2);
  Gaussian system_Uncertainty(sysNoise_Mu, sysNoise_Cov);
  sys_pdf = new NonLinearAnalyticConditionalGaussianOdo(system_Uncertainty);
  sys_model_ = new AnalyticSystemModelGaussianUncertainty(sys_pdf_);

  // create MEASUREMENT MODEL ODOM
  ColumnVector measNoiseOdom_Mu(6);
  measNoiseOdom_Mu = 0;
  SymmetricMatrix measNoiseOdom_Cov(6);
  measNoiseOdom_Cov = 0;
  for (unsigned int i = 1; i <= 6; i++)
    measNoiseOdom_Cov(i, i) = 1;
  Gaussian measurement_Uncertainty_Odom(measNoiseOdom_Mu, measNoiseOdom_Cov);
  Matrix Hodom(6, 6);
  Hodom = 0;
  Hodom(1, 1) = 1;
  Hodom(2, 2) = 1;
  Hodom(6, 6) = 1;

```
odom_meas_pdf_ = new LinearAnalyticConditionalGaussian(Hodom, measurement_Uncertainty_Odom);
odom_meas_model_ = new LinearAnalyticMeasurementModelGaussianUncertainty(odom_meas_pdf_);

// create MEASUREMENT MODEL IMU
ColumnVector measNoiseImu_Mu(3);
measNoiseImu_Mu = 0;
SymmetricMatrix measNoiseImu_Cov(3);
measNoiseImu_Cov = 0;
for (unsigned int i = 1; i <= 3; i++)
  measNoiseImu_Cov(i, i) = 1;
Gaussian measurement_Uncertainty_Imu(measNoiseImu_Mu, measNoiseImu_Cov);
Matrix Himu(3, 6);
Himu = 0;
Himu(1, 4) = 1;
Himu(2, 5) = 1;
Himu(3, 6) = 1;
imu_meas_pdf_ = new LinearAnalyticConditionalGaussian(Himu, measurement_Uncertainty_Imu);
imu_meas_model_ = new LinearAnalyticMeasurementModelGaussianUncertainty(imu_meas_pdf_);

// create MEASUREMENT MODEL VO
ColumnVector measNoiseVo_Mu(6);
measNoiseVo_Mu = 0;
SymmetricMatrix measNoiseVo_Cov(6);
measNoiseVo_Cov = 0;
for (unsigned int i = 1; i <= 6; i++)
  measNoiseVo_Cov(i, i) = 1;
Gaussian measurement_Uncertainty_Vo(measNoiseVo_Mu, measNoiseVo_Cov);
Matrix Hvo(6, 6);
Hvo = 0;
Hvo(1, 1) = 1;
Hvo(2, 2) = 1;
Hvo(3, 3) = 1;
Hvo(4, 4) = 1;
Hvo(5, 5) = 1;
Hvo(6, 6) = 1;
vo_meas_pdf_ = new LinearAnalyticConditionalGaussian(Hvo, measurement_Uncertainty_Vo);
vo_meas_model_ = new LinearAnalyticMeasurementModelGaussianUncertainty(vo_meas_pdf_);

// create MEASUREMENT MODEL GPS
ColumnVector measNoiseGps_Mu(3);
measNoiseGps_Mu = 0;
SymmetricMatrix measNoiseGps_Cov(3);
measNoiseGps_Cov = 0;
for (unsigned int i = 1; i <= 3; i++)
  measNoiseGps_Cov(i, i) = 1;
Gaussian measurement_Uncertainty_GPS(measNoiseGps_Mu, measNoiseGps_Cov);
Matrix Hgps(3, 6);
Hgps = 0;
Hgps(1, 1) = 1;
Hgps(2, 2) = 1;
```

```
  Hgps(3, 3) = 1;
  gps_meas_pdf_ = new LinearAnalyticConditionalGaussian(Hgps, measurement_Uncertainty_GPS);
  gps_meas_model_ = new LinearAnalyticMeasurementModelGaussianUncertainty(gps_meas_pdf_);
};

// destructor
OdomEstimation::~OdomEstimation()
{
  if (filter)
    delete filter;
  if (prior)
    delete prior;
  delete odom_meas_model;
  delete odom_meas_pdf;
  delete imu_meas_model;
  delete imu_meas_pdf;
  delete vo_meas_model;
  delete vo_meas_pdf;
  delete gps_meas_model;
  delete gps_meas_pdf;
  delete sys_pdf;
  delete sys_model;
};

// initialize prior density of filter
void OdomEstimation::initialize(const Transform &prior, const Time &time)
{
  // set prior of filter
  ColumnVector prior_Mu(6);
  decomposeTransform(prior, prior_Mu(1), prior_Mu(2), prior_Mu(3), prior_Mu(4), prior_Mu(5), prior_Mu(6));
  SymmetricMatrix prior_Cov(6);
  for (unsigned int i = 1; i <= 6; i++)
  {
    for (unsigned int j = 1; j <= 6; j++)
    {
      if (i == j)
        prior_Cov(i, j) = pow(0.001, 2);
      else
        prior_Cov(i, j) = 0;
    }
  }
  prior_ = new Gaussian(prior_Mu, prior_Cov);
  filter_ = new ExtendedKalmanFilter(prior_);

  // remember prior
  addMeasurement(StampedTransform(prior, time, output_frame, base_footprint_frame));
  filter_estimate_old_vec_ = prior_Mu;
  filter_estimate_old_ = prior;
  filter_time_old_ = time;
```

```
  // filter initialized
  filter_initialized_ = true;
 }

 // update filter
 bool OdomEstimation::update(bool odom_active, bool imu_active, bool gps_active, bool vo_active, const Time
&filter_time, bool &diagnostics_res)
 {
  // only update filter when it is initialized
  if (!filter_initialized_)
  {
   ROS_INFO("Cannot update filter when filter was not initialized first.");
   return false;
  }

  // only update filter for time later than current filter time
  double dt = (filter_time - filter_time_old_).toSec();
  if (dt == 0)
   return false;
  if (dt < 0)
  {
   ROS_INFO("Will not update robot pose with time %f sec in the past.", dt);
   return false;
  }
  ROS_DEBUG("Update filter at time %f with dt %f", filter_time.toSec(), dt);

  // system update filter
  // --------------------
  // for now only add system noise
  ColumnVector vel_desi(2);
  vel_desi = 0;
  filter->Update(sys_model, vel_desi);

  // process odom measurement
  // ------------------------
  ROS_DEBUG("Process odom meas");
  if (odom_active)
  {
   if (!transformer.canTransform(base_footprint_frame, "wheelodom", filter_time))
   {
    ROS_ERROR("filter time older than odom message buffer");
    return false;
   }
   transformer.lookupTransform("wheelodom", base_footprint_frame, filter_time, odom_meas);
   if (odom_initialized)
   {
    // convert absolute odom measurements to relative odom measurements in horizontal plane
    Transform odom_rel_frame = Transform(tf::createQuaternionFromYaw(filter_estimate_old_vec(6)),
                                         filter_estimate_old.getOrigin()) *
                               odom_meas_old.inverse() * odom_meas;
```

```
      ColumnVector odom_rel(6);
      decomposeTransform(odom_rel_frame, odom_rel(1), odom_rel(2), odom_rel(3), odom_rel(4), odom_rel(5),
odom_rel(6));
      angleOverflowCorrect(odom_rel(6), filter_estimate_old_vec(6));
      // update filter
      odom_meas_pdf->AdditiveNoiseSigmaSet(odom_covariance_ * pow(dt, 2));

      ROS_DEBUG("Update filter with odom measurement %f %f %f %f %f %f",
                odom_rel(1), odom_rel(2), odom_rel(3), odom_rel(4), odom_rel(5), odom_rel(6));
      filter->Update(odom_meas_model, odom_rel);
      diagnostics_odom_rot_rel_ = odom_rel(6);
    }
    else
    {
      odom_initialized_ = true;
      diagnostics_odom_rot_rel_ = 0;
    }
    odom_meas_old_ = odom_meas;
  }
  // sensor not active
  else
    odom_initialized = false;

  // process imu measurement
  // -----------------------
  if (imu_active)
  {
    if (!transformer.canTransform(base_footprint_frame, "imu", filter_time))
    {
      ROS_ERROR("filter time older than imu message buffer");
      return false;
    }
    transformer.lookupTransform("imu", base_footprint_frame, filter_time, imu_meas);
    if (imu_initialized)
    {
      // convert absolute imu yaw measurement to relative imu yaw measurement
      Transform imu_rel_frame = filter_estimate_old_ * imu_meas_old.inverse() * imu_meas;
      ColumnVector imu_rel(3);
      double tmp;
      decomposeTransform(imu_rel_frame, tmp, tmp, tmp, tmp, tmp, imu_rel(3));
      decomposeTransform(imu_meas, tmp, tmp, tmp, imu_rel(1), imu_rel(2), tmp);
      angleOverflowCorrect(imu_rel(3), filter_estimate_old_vec(6));
      diagnostics_imu_rot_rel_ = imu_rel(3);
      // update filter
      imu_meas_pdf->AdditiveNoiseSigmaSet(imu_covariance * pow(dt, 2));
      filter->Update(imu_meas_model, imu_rel);
    }
    else
    {
      imu_initialized_ = true;
```

```
    diagnostics_imu_rot_rel_ = 0;
  }
  imu_meas_old_ = imu_meas;
}
// sensor not active
else
  imu_initialized = false;

// process vo measurement
// ----------------------
if (vo_active)
{
  if (!transformer.canTransform(base_footprint_frame, "vo", filter_time))
  {
    ROS_ERROR("filter time older than vo message buffer");
    return false;
  }
  transformer.lookupTransform("vo", base_footprint_frame, filter_time, vo_meas);
  if (vo_initialized)
  {
    // convert absolute vo measurements to relative vo measurements
    Transform vo_rel_frame = filter_estimate_old_ * vo_meas_old.inverse() * vo_meas;
    ColumnVector vo_rel(6);
    decomposeTransform(vo_rel_frame, vo_rel(1), vo_rel(2), vo_rel(3), vo_rel(4), vo_rel(5), vo_rel(6));
    angleOverflowCorrect(vo_rel(6), filter_estimate_old_vec(6));
    // update filter
    vo_meas_pdf->AdditiveNoiseSigmaSet(vo_covariance_ * pow(dt, 2));
    filter->Update(vo_meas_model, vo_rel);
  }
  else
    vo_initialized_ = true;
  vo_meas_old_ = vo_meas;
}
// sensor not active
else
  vo_initialized = false;

// process gps measurement
// ----------------------
if (gps_active)
{
  if (!transformer.canTransform(base_footprint_frame, "gps", filter_time))
  {
    ROS_ERROR("filter time older than gps message buffer");
    return false;
  }
  transformer.lookupTransform("gps", base_footprint_frame, filter_time, gps_meas);
  if (gps_initialized)
  {
    gps_meas_pdf->AdditiveNoiseSigmaSet(gps_covariance * pow(dt, 2));
```

```
    ColumnVector gps_vec(3);
    double tmp;
    // Take gps as an absolute measurement, do not convert to relative measurement
    decomposeTransform(gps_meas, gps_vec(1), gps_vec(2), gps_vec(3), tmp, tmp, tmp);
    filter->Update(gps_meas_model, gps_vec);
   }
   else
   {
    gps_initialized = true;
    gps_meas_old_ = gps_meas;
   }
  }
  // sensor not active
  else
   gps_initialized = false;

  // remember last estimate
  filter_estimate_old_vec_ = filter->PostGet()->ExpectedValueGet();
  tf::Quaternion q;
  q.setRPY(filter_estimate_old_vec(4), filter_estimate_old_vec(5), filter_estimate_old_vec(6));
  filter_estimate_old_ = Transform(q,
                                   Vector3(filter_estimate_old_vec(1), filter_estimate_old_vec(2), filter_estimate_old_vec(3)));
  filter_time_old = filter_time;
  addMeasurement(StampedTransform(filter_estimate_old, filter_time, output_frame, base_footprint_frame_));

  // diagnostics
  diagnostics_res = true;
  if (odom_active && imu_active)
  {
   double diagnostics = fabs(diagnostics_odom_rot_rel_ - diagnostics_imu_rot_rel_) / dt;
   if (diagnostics > 0.3 && dt > 0.01)
   {
    diagnostics_res = false;
   }
  }

  return true;
};

void OdomEstimation::addMeasurement(const StampedTransform &meas)
{
  ROS_DEBUG("AddMeasurement from %s to %s:  (%f, %f, %f)  (%f, %f, %f, %f)",
            meas.frame_id.c_str(), meas.child_frame_id.c_str(),
            meas.getOrigin().x(), meas.getOrigin().y(), meas.getOrigin().z(),
            meas.getRotation().x(), meas.getRotation().y(),
            meas.getRotation().z(), meas.getRotation().w());
  transformer_.setTransform(meas);
}
```

```
void OdomEstimation::addMeasurement(const StampedTransform &meas, const MatrixWrapper::SymmetricMatrix &covar)
{
  // check covariance
  for (unsigned int i = 0; i < covar.rows(); i++)
  {
    if (covar(i + 1, i + 1) == 0)
    {
      ROS_ERROR("Covariance specified for measurement on topic %s is zero", meas.child_frame_id.c_str());
      return;
    }
  }
  // add measurements
  addMeasurement(meas);
  if (meas.child_frame_id == "wheelodom")
    odom_covariance_ = covar;
  else if (meas.child_frame_id_ == "imu")
    imu_covariance_ = covar;
  else if (meas.child_frame_id_ == "vo")
    vo_covariance_ = covar;
  else if (meas.child_frame_id_ == "gps")
    gps_covariance_ = covar;
  else
    ROS_ERROR("Adding a measurement for an unknown sensor %s", meas.child_frame_id_.c_str());
};

// get latest filter posterior as vector
void OdomEstimation::getEstimate(MatrixWrapper::ColumnVector &estimate)
{
  estimate = filter_estimate_old_vec_;
};

// get filter posterior at time 'time' as Transform
void OdomEstimation::getEstimate(Time time, Transform &estimate)
{
  StampedTransform tmp;
  if (!transformer.canTransform(base_footprint_frame, output_frame, time))
  {
    ROS_ERROR("Cannot get transform at time %f", time.toSec());
    return;
  }
  transformer.lookupTransform(output_frame, base_footprint_frame, time, tmp);
  estimate = tmp;
};

// get filter posterior at time 'time' as Stamped Transform
void OdomEstimation::getEstimate(Time time, StampedTransform &estimate)
{
  if (!transformer.canTransform(output_frame, base_footprint_frame, time))
  {
    ROS_ERROR("Cannot get transform at time %f", time.toSec());
```

```
    return;
  }
  transformer.lookupTransform(output_frame, base_footprint_frame, time, estimate);
};

// get most recent filter posterior as PoseWithCovarianceStamped
void OdomEstimation::getEstimate(geometry_msgs::PoseWithCovarianceStamped &estimate)
{
  // pose
  StampedTransform tmp;
  if (!transformer.canTransform(output_frame, base_footprint_frame, ros::Time()))
  {
    ROS_ERROR("Cannot get transform at time %f", 0.0);
    return;
  }
  transformer.lookupTransform(output_frame, base_footprint_frame, ros::Time(), tmp);
  poseTFToMsg(tmp, estimate.pose.pose);

  // header
  estimate.header.stamp = tmp.stamp;
  estimate.header.frame_id = output_frame;

  // covariance
  SymmetricMatrix covar = filter_->PostGet()->CovarianceGet();
  for (unsigned int i = 0; i < 6; i++)
    for (unsigned int j = 0; j < 6; j++)
      estimate.pose.covariance[6 * i + j] = covar(i + 1, j + 1);
};

// correct for angle overflow
void OdomEstimation::angleOverflowCorrect(double &a, double ref)
{
  while ((a - ref) > M_PI)
    a -= 2 * M_PI;
  while ((a - ref) < -M_PI)
    a += 2 * M_PI;
};

// decompose Transform into x,y,z,Rx,Ry,Rz
void OdomEstimation::decomposeTransform(const StampedTransform &trans,
                                        double &x, double &y, double &z, double &Rx, double &Ry, double &Rz)
{
  x = trans.getOrigin().x();
  y = trans.getOrigin().y();
  z = trans.getOrigin().z();
  trans.getBasis().getEulerYPR(Rz, Ry, Rx);
};
```

```
// decompose Transform into x,y,z,Rx,Ry,Rz
void OdomEstimation::decomposeTransform(const Transform &trans,
                                        double &x, double &y, double &z, double &Rx, double &Ry, double &Rz)
{
  x = trans.getOrigin().x();
  y = trans.getOrigin().y();
  z = trans.getOrigin().z();
  trans.getBasis().getEulerYPR(Rz, Ry, Rx);
};

void OdomEstimation::setOutputFrame(const std::string &output_frame)
{
  output_frame_ = output_frame;
};

void OdomEstimation::setBaseFootprintFrame(const std::string &base_frame)
{
  base_footprint_frame_ = base_frame;
};

}; // namespace
```