Getting Started Guide
Bayesian Filtering Library

Tinne De Laet, Wim Meeussen, Klaas Gadeyne

September 29, 2020

Chapter 1

Introduction to BFL

1.1 What is BFL?

The Bayesian Filtering Library (BFL) has been designed with the following requirements in mind:

Bayesian BFL provides a fully Bayesian software framework, i.e. all Bayesian methods fit in the library design. Therefore the library does not impose restrictions on the nature of the Random Variables nor on the representation of the probability density function (analytical, sample based, . . .). The common Bayesian background of BFL allows to implement different Bayesian algorithms with a maximum of code reuse. Moreover, the performance of these algorithms can be compared with a minimum effort.

Open The Open Source Initiative offers an enormous potential for the maximum reuse of code and is ideal to study differences between different algorithms or different implementations of a particular algorithm. Furthermore open source software is a key factor implementing the idea of *reproducible research*. The state of the art can only benefit from the availability of the source code, in order to reproduce the obtained results and to gain better understanding by altering certain chosen values of the experiments.

Independent The first meaning of independent is the independence to the numerical and stochastical library. At present, there is no *standard* numerical nor stochastical library for C++ available, there's a wide range of libraries available providing the necessary functionality. An estimation library is only a part of the whole software infrastructure in order to perform a certain task. To avoid ending up with multiple libraries for one project, BFL is decoupled possible from one particular numerical/stochastical library.

The second meaning of independent is the independence of BFL of a particular application (such as Autonomous Compliant Motion, mobile robotics, econometrics, . . .). This means both its interface and implementation are decoupled from particular sensors, assumptions, algorithms, . . . that are specific to a certain application.

Furthermore, BFL is integrated in our robot control software written in C++ and therefore, C++ is chosen as programming language.

1.2 BFL's History

BFL was born during the PhD research of Klaas Gadeyne (See `http://people.mech.kuleuven.be/~kgadeyne/`). Through an analysis of the general Bayesian framework underneath the above mentioned Bayesian filters, he designed the state estimation software framework/library BFL, containing support for different filters (in particular Sequential Monte Carlo methods and Kalman filters, but also e.g. grid based methods) and easily extendible towards other Bayesian methods. Late 2001, BFL was released as an open source project, and is since then slowly growing and maturing.

1.3 Getting support - the BFL community

There are different ways to get some help/support:

- This tutorial!
- The website `http://www.orocos.org/bfl` with the doxygen documentation of BFL. A symbolic link is included to one of the svn copies so you can even browse the source code.

- Klaas Gadeyne's PhD thesis contains a chapter about BFL's design (again, see the website)
- The mailing list (see `http://lists.mech.kuleuven.be/mailman/listinfo/bfl/`) for questions and discussions.

Chapter 2

Tutorial

2.1 How to prepare the tutorial examples

First you get the source of the tutorial examples. To see the source code of the examples, you need to get the BFL source:

- You can download the bfl tarball from `http://www.orocos.org/bfl/source`, or
- get BFL from subversion `http://www.orocos.org/bfl/subversion`.

Both cases are explained in more detail in the BFL installation guide on `http://www.orocos.org/bfl`. After doing this, you will find the example programs in bfl/examples.

Now you are already set to start the tutorial. However, if you also want to execute the examples, and seen the numerical results yourself, you need to choose one of these two options:

- Compile the BFL source as explained in the installation guide, and find the binary examples in bfl/build/examples
- Or you can get the precompiled BFL examples from our aptitude server. Therefore you need to:
  1. First add the following line to your /etc/apt/sources.list:

     ```
     deb http://people.mech.kuleuven.be/~tdelaet/bfl _mydistro_ main
     ```

     where *_mydistro_* is breezy, dapper, edgy or feisty.
  2. Then apt-get the packages:

     ```
     sudo apt-get update
     sudo apt-get install orocos-bfl-examples
     ```

     Now you find the precompiled BFL examples in /usr/bin/bfl.

2.2 The general procedure to construct a filter

To constuct a filter one has to construct:

- the system model (daughter of SystemModel),
- the measurement model (daughter of MeasurementModel),
- the prior density (daughter of Pdf), and finally
- the filter itself (daughter of Filter).

A system model contains the ConditionalPdf representing $P(x_k|x_{k-1}, u_k)$, with x the state variable, k the time step and u the input. The ConditionalPdf of the system often makes use of some basic type of uncertainty, which can be modeled using the Pdf class (e.g. simple Gaussian). Summarized, the construction of the system model (often) needs the following steps:

- create a Pdf to model the system uncertainty,
- create a ConditionalPdf representing $P(x_k|x_{k-1}, u_k)$ using the Pdf of the system uncertainty,

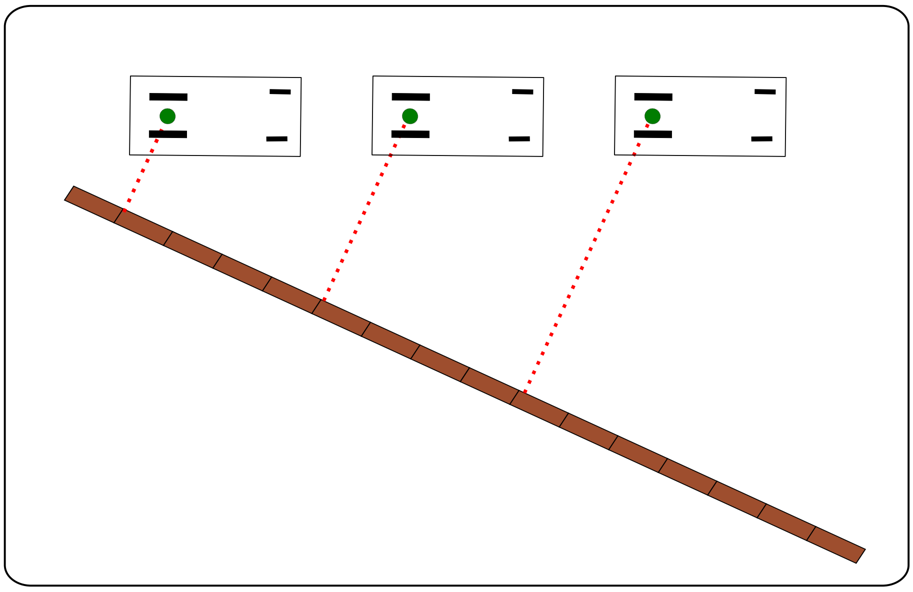

Figure 2.1: Mobile robot

- create the SystemModel itself, using the ConditionalPdf.

Similar to the system model, the measurement model contains the ConditionalPdf representing $P(z_k|x_k, s_k)$, with z the measurement variable, x the state variable, k the time step and s the (optional) sensing parameters. The ConditionalPdf of the measurement often makes use of some basic type of uncertainty, which can be modeled using the Pdf class (e.g. simple Gaussian). Summarized, the construction of the measurement model (often) needs the following steps:

- create a Pdf to model the measurement uncertainty,
- create a ConditionalPdf representing $P(z_k|x_k, s_k)$ using the Pdf of the measurement uncertainty,
- create the MeasurementModel itself, using the ConditionalPdf.

The prior density is a simple Pdf, or a daughter of the Pdf class. The final step consist of creating a filter (daughter of Filter) and in this tutorial the prior density is passed along with the constructor.

2.3 Your first Kalman filter in 15 minutes

This section will guide you step-by-step through the implementation of a Kalman filter with a linear system model and a linear measurement model. We'll describe all implementation steps completely, but we'll assume that you have some basic knowledge about Bayesian estimation theory.

In this first example we use a Kalman filter to estimate the position of a mobile robot. The mobile robot drives around in an open space with one wall. In this example, the mobile robot can only drive forward and backwards, and not turn. Using a distance sensor, the mobile robot measures its distance to this wall.

2.3.1 Preparing the .cpp file

In the folder

```
BFL/examples/linear_kalman/
```

you find the source file called

```
test_linear_kalman.cpp
```

This file contains the implementation we discuss in this first example. The beginning of the file contains some C++ overhead, such as the inclusion of the BFL header files we'll use

```
  #include <filter/extendendkalmanfilter.h>
  #include <model/linearanalyticsystemmodel_gaussianuncertainty.h>
  #include <model/linearanalyticmeasurementmodel_gaussianuncertainty.h>
  #include <pdf/linearanalyticsystemmodel_gaussianuncertainty.h>
  #include <pdf/linearanalyticmeasurementmodel_gaussianuncertainty.h>,
```

the mobile robot simulator

```
#include "../mobile_robot.h",
```

the file from which some properties are read (e.g. the initial state estimate)

```
#include "../mobile_robot_wall_cts.h",
```

and the namespaces that we'll use

```
  using namespace MatrixWrapper;
  using namespace BFL;
  using namespace std;
```

2.3.2 The linear system model

We'll now build the system model, to make a prediction of the position of the mobile robot at each time step. The prediction is based on the velocity of the mobile robot. Because the mobile robot cannot turn, the prediction of the position is a linear equation. Since the model is linear, BFL already includes a ready-to-use implementation.

First we build the linear model that calculates the position of the mobile robot x_{k+1}, given its previous position x_k and its velocity u_k consisting of the translational velocity v and the rotational velocity ω (which in this first example is zero). The linear model is defined by:

$$x_{k+1} = Ax_k + Bu_k \tag{2.1}$$

We build the matrices A and B:

```
  Matrix A(2,2);
  A(1,1) = 1.0;
  A(1,2) = 0.0;
  A(2,1) = 0.0;
  A(2,2) = 1.0;

  Matrix B(2,2);
  B(1,1) = cos(0.8);
  B(1,2) = 0.0;
  B(2,1) = sin(0.8);
  B(2,2) = 0.0;
```

and we combine these two matrices in a vector AB:

```
  vector<Matrix> AB(2);
  AB[0] = A;
  AB[1] = B;
```

Then we create a Gaussian distribution, which is defined by a mean (Mu) and a covariance (Cov). This Gaussian distribution represents the uncertainty on the predicted position of the mobile robot. The mean is defined by $MU\_SYSTEM\_NOISE\_X$ and $MU\_SYSTEM\_NOISE\_Y$[1]:

```
  ColumnVector sysNoise_Mu(2);
  sysNoise_Mu(1) = MU_SYSTEM_NOISE_X;
  sysNoise_Mu(2) = MU_SYSTEM_NOISE_Y;
```

and the covariance is chosen as a diagonal matrix, with diagonals $SIGMA\_SYSTEM\_NOISE\_X$ and $SIGMA\_SYSTEM\_NOISE\_Y$:

[1] $MU\_SYSTEM\_NOISE\_X$ and $MU\_SYSTEM\_NOISE\_Y$ are defined in the file *mobile_robot_wall_cts.h*. This remarks holds for all constant, indicated with CAPITALS in this tutorial.

```
SymmetricMatrix sysNoise_Cov(2);
sysNoise_Cov = 0.0;
sysNoise_Cov(1,1) = SIGMA_SYSTEM_NOISE_X;
sysNoise_Cov(1,2) = 0.0;
sysNoise_Cov(2,1) = 0.0;
sysNoise_Cov(2,2) = SIGMA_SYSTEM_NOISE_Y;
```

The mean and covariance together define a two dimensional Gaussian distribution:

```
Gaussian system_Uncertainty(sysNoise_Mu, sysNoise_Cov);
```

Now we create a linear conditional probability density function (pdf) which represents the probability of the predicted position given the current position of the mobile robot. This pdf is defined by the vector AB which represents the linear model, together with the Gaussian distribution which represents the system model's extra uncertainty.

```
LinearAnalyticConditionalGaussian sys_pdf(AB, system_Uncertainty);
```

Finally we create the system model from the system pdf:

```
LinearAnalyticSystemModelGaussianUncertainty sys_model(&sys_pdf);
```

2.3.3 The linear measurement model

We'll now build the measurement model, to correct the prediction of the position of the mobile robot, based on the distance measurement to the wall. Since we use a linear measurement model in this example, BFL already includes a ready-to-use implementation.

First we build the linear model that links the distance measurement to the position of the mobile robot.

$$z_{k+1} = Hx_{k+1} \tag{2.2}$$

We build the matrix H:

```
Matrix H(1,2);
double wall_ct = 2/(sqrt(pow(RICO_WALL,2.0) + 1));
H = 0.0;
H(1,1) = wall_ct * RICO_WALL;
H(1,2) = 0 - wall_ct;
```

where *wall_ct* is a helper constant and $RICO\_WALL$ is the slope of the wall. Then we create a Gaussian distribution, which is defined by a mean (Mu) and a covariance (Cov). This Gaussian distribution represents the uncertainty on the measured distance to the wall. The mean is MU_MEAS_NOISE:

```
ColumnVector measNoise_Mu(1);
measNoise_Mu(1) = MU_MEAS_NOISE;
```

and the covariance is diagonal matrix, with $SIGMA_MEAS_NOISE$ as the σ^2 boundary:

```
SymmetricMatrix measNoise_Cov(1);
measNoise_Cov(1,1) = SIGMA_MEAS_NOISE;
```

The mean and covariance together define a two dimensional Gaussian distribution:

```
Gaussian measurement_Uncertainty(measNoise_Mu, measNoise_Cov);
```

Now we create a linear probability density function (pdf) which represents the probability of the distance measurement. This pdf is defined by the matrix H which represents the linear model, together with the Gaussian distribution which represents the measurement uncertainty.

```
LinearAnalyticConditionalGaussian meas_pdf(H, measurement_Uncertainty);
```

Finally we create the measurement model from the measurement pdf:

```
LinearAnalyticMeasurementModelGaussianUncertainty meas_model(&meas_pdf);
```

2.3.4 Prior Distribution

We'll now build the prior distribution representing the initial estimate and the uncertainty associated with this initial estimate.
The prior distribution is a Gaussian distribution with mean (prior_Mu) and covariance (prior_Cov). The mean is the initial estimated state:

```
ColumnVector prior_Mu(2);
prior_Mu(1) = PRIOR_MU_X;
prior_Mu(2) = PRIOR_MU_Y;
```

and the covariance is a diagonal matrix, with $PRIOR\_COV\_X$ and $PRIOR\_COV\_Y$ as the σ^2 boundary:

```
SymmetricMatrix prior_Cov(2);
prior_Cov(1,1) = PRIOR_COV_X;
prior_Cov(1,2) = 0.0;
prior_Cov(2,1) = 0.0;
prior_Cov(2,2) = PRIOR_COV_Y;
```

The mean and covariance together define a two dimensional Gaussian distribution:

```
Gaussian prior(prior_Mu, prior_Cov);
```

2.3.5 Construction of the filter

In this example we want to use a Kalman filter to estimate the unknown position of the mobile robot. BFL provides the class ExtendedKalmanFilter for this purpose:

```
ExtendedKalmanFilter filter(&prior_cont);
```

The only argument that has to be provided to the ExtendedKalmanFilter is the prior distribution.

Remark: Remark that an extended Kalman filter can also handle nonlinear system and/or measurement models. When looking at BFL one could first be tempted to create a simple Kalman filter by using the class KalmanFilter (kalmanfilter.h). KalmanFilter however is a class representing the family of all Kalman Filters (EKF, IEKF, ...). The system of updating the Posterior density is implemented in this base class. However, the parameters used for this update differ for different KFs (Simple KF,EKF,IEKF): that's why the xUpdate members are still pure virtual functions. Therefore, KalmanFilter is not the class which is closest to our needs: ExtendedKalmanFilter is the class closest to our needs.

2.3.6 Mobile Robot Simulator

A mobile robot simulator *MobileRobot* is created to simulate a mobile robot driving around in a world with one wall. In a real world experiment this simulator is of course replaced by a real mobile robot. The properties of the mobile robot are defined in *mobile_robot_wall_cts.h* (e.g. it's initial position, the noise on the simulated system, c...) The mobile robot simulator is created as follows:

```
MobileRobot mobile_robot;
```

The simulator has three methods:

- Move(input)
  This method drives the mobile robot with the specified velocity input, during one second.
- Measure( )
  This method returns one distance measurements to the wall, from the current position of the mobile robot.
- GetState( )
  This method returns the true position of the mobile robot and is obviously not used in the estimation but just to verify the estimation results.

Every time step, the mobile robot has to receive the velocity input to move it as desired

```
mobile_robot.Move( input );
```

Arriving at it's new position, the distance to the wall is measured:

```
ColumnVector measurement = mobile_robot.Measure();
```

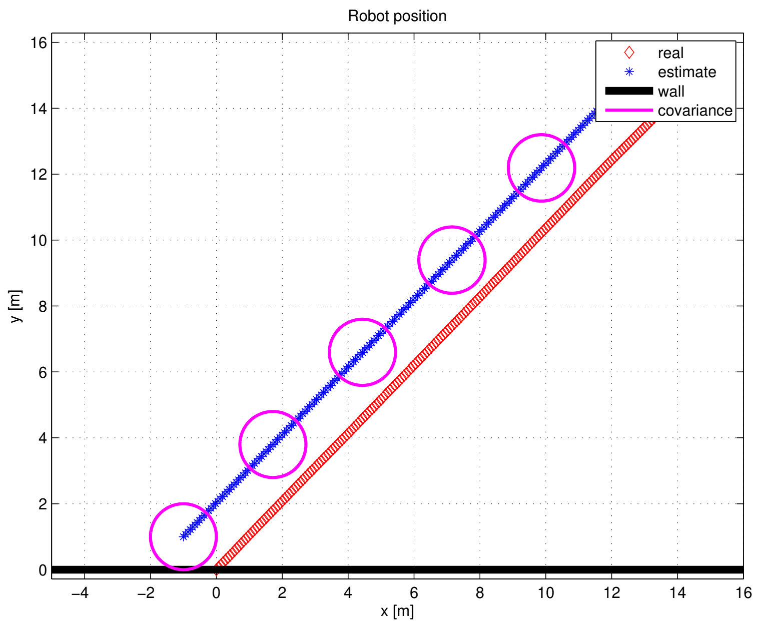

Figure 2.2: Result of estimation without measurement information

2.3.7 Update of the filter and new estimation

At every time step, a new estimation can be made based on the information provided by the system model and the measurement model. If the measurement information and the information from the system model will be used to update the filter, the update of the filter is:

```
filter->Update(&sys_model,input,&meas_model, measurement);
```

If only a prediction according to the system model will be used the update reduces to:

```
filter->Update(&sys_model, input);
```

To get the posterior of the updated filter which is the result of all the system model and measurement information:

```
Pdf<ColumnVector> * posterior = filter->PostGet();
```

The expected value and covariance can be obtained according to:

```
posterior->ExpectedValueGet();
posterior->CovarianceGet();
```

2.3.8 Results

Results of the implementation are shown in Figure 2.2 without making use of the measurement information and Figure 2.3 with measurement information.

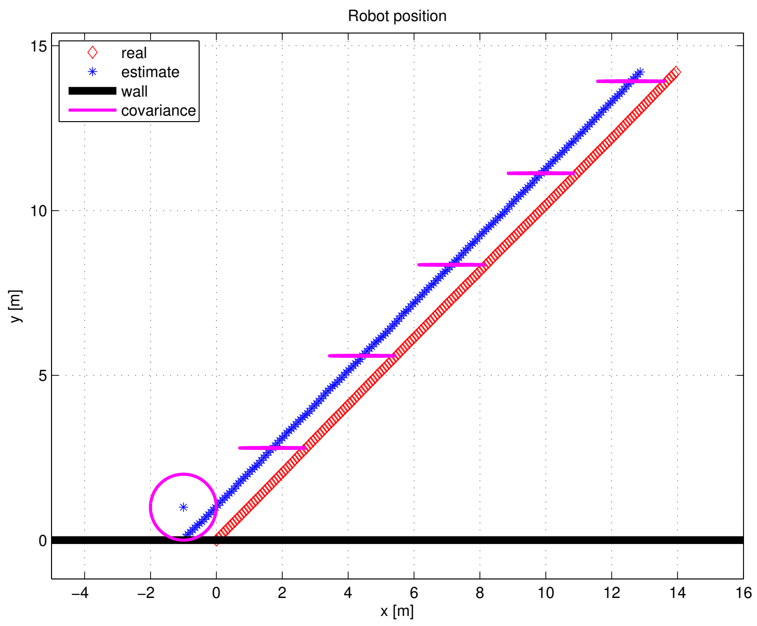

Figure 2.3: Result of estimation with measurement information

2.4 An extended Kalman filter with nonlinear system model in only 10 minutes extra

After completing the tutorial of the linear Kalman filter, this section will guide you step-by-step through the implementation of an extended Kalman filter using an non linear system model and a linear measurement model. This tutorial assumes you have completed the previous tutorial.

In this second example we use a Kalman filter to estimate the position and the orientation of a mobile robot. The mobile robot drives around in an open space with one wall. Again, using a distance sensor, the mobile robot measures its distance to this wall.

2.4.1 Preparing the .cpp file

In the folder

```
BFL/examples/nonlinear_kalman/
```

you find the source file called

```
test_nonlinear_kalman.cpp
```

This file contains the implementation we discuss in this second example. The beginning of the file contains the same C++ overhead as the first tutorial except for the fact we have to include the proper —This file contains the implementation we discuss in this second filter:

```
#include <filter/extendedkalmanfilter.h>
```

and one extra inclusion for the class we will create ourselves in this tutorial:

```
#include "nonlinearanalyticconditionalgaussianmobile.h"
```

2.4.2 The nonlinear system model

In contrast with the system model of the previous model, the system model which includes the orientation of the robot is no longer linear.

The noise on the system model is still considered Gaussian and is therefore implemented as explained in the first tutorial. We create a Gaussian distribution, which is defined by a mean (Mu) and a covariance (Cov). This Gaussian distribution represents the uncertainty on the predicted position of the mobile robot. The mean values are defined in the file *mobile_robot_wall_cts.h*:

```
ColumnVector sys_noise_Mu(STATE_SIZE);
sys_noise_Mu(1) = MU_SYSTEM_NOISE_X;
sys_noise_Mu(2) = MU_SYSTEM_NOISE_Y;
sys_noise_Mu(3) = MU_SYSTEM_NOISE_THETA;
```

and the covariance is chosen as a diagonal matrix, with the digaonal values defined in the file *mobile_robot_wall_cts.h*:

```
SymmetricMatrix sys_noise_Cov(STATE_SIZE);
sys_noise_Cov = 0.0;
sys_noise_Cov(1,1) = SIGMA_SYSTEM_NOISE_X;
sys_noise_Cov(1,2) = 0.0;
sys_noise_Cov(1,3) = 0.0;
sys_noise_Cov(2,1) = 0.0;
sys_noise_Cov(2,2) = SIGMA_SYSTEM_NOISE_Y;
sys_noise_Cov(2,3) = 0.0;
sys_noise_Cov(3,1) = 0.0;
sys_noise_Cov(3,2) = 0.0;
sys_noise_Cov(3,3) = SIGMA_SYSTEM_NOISE_THETA;
```

The mean and covariance together define a three dimensional Gaussian distribution:

```
Gaussian system_Uncertainty(sys_noise_Mu, sys_noise_Cov);
```

Now we have to create a non linear conditional probability density function (pdf) which represents the probability of the predicted position given the current position of the mobile robot. BFL does not provide a class for such a non linear conditional pdf[2]. Therefore we have to implement it ourselves. We give the class we'll create a specific name for this example: NonLinearAnalyticConditionalGaussianMobile. BFL provides a general class of an analytic conditional pdf with Gaussian uncertainty from which we can inherit the functionality: AnalyticConditionalGaussianAdditiveNoise. Indeed, our pdf is analytic and has some additive Gaussian noise.
To generate our own class we start with the implementation of the header file: *nonlinearanalyticconditionalgaussian-mobile.h*.
We start the class implementation with the inclusion of the header of the class we inherit from:

```
#include <pdf/analyticconditionalgaussian_additivenoise.h>
```

The class will be created in the BFL namespace and obviously needs a constructor and destructor. The constructor expects a Gaussian representing the system noise.

```
namespace BFL
{
 class NonLinearAnalyticConditionalGaussianMobile :
  public AnalyticConditionalGaussianAdditiveNoise
 {
  public:
   NonLinearAnalyticConditionalGaussianMobile(const Gaussian& additiveNoise);
   virtual ~NonLinearAnalyticConditionalGaussianMobile();
 };
}
```

When we look at the documentation of AnalyticConditionalGaussianAdditiveNoise, we notice that there are at least two methods we have to implement when inheriting:

- ColumnVector ExpectedValueGet(),
  which will return the expected value of the conditional pdf and depends on our problem specific process equations.
- Matrix dfGet(unsigned int i),
  which will return the derivative to the i'th conditional argument. In this case the derivative to the state (first conditional argument) is only relevant.

Moreover this are the only two functions which needs re-implementation. Therefore we add to following declarations to the public part of the header file:

[2]Actually it does, if you use the symbolic toolbox GiNaC. This option is in some cases easier to use, but results in a rather slow implementation of a non linear conditional pdf.

```
virtual MatrixWrapper::ColumnVector    ExpectedValueGet()  const;
virtual MatrixWrapper::Matrix          dfGet(unsigned int i)  const;
```

The header file is ready, and we can now switch to the cpp file for the implementation: *nonlinearanalyticconditional-gaussianmobile.cpp*.
The conditional arguments of the conditional pdf, the state and the input (velocity), are private variables of the conditional pdf.
Now, we switch to the implementation of ExpectedValueGet(). To calculate the expected value of the conditional pdf, the current state and input are necessary:

```
ColumnVector state = ConditionalArgumentGet(0);
ColumnVector vel  = ConditionalArgumentGet(1);
```

The expected value is then calculated and returned by:

```
state(1) += cos(state(3)) * vel(1);
state(2) += sin(state(3)) * vel(1);
state(3) += vel(2);
return state + AdditiveNoiseMuGet();
```

Now, only the implementation of dfGet(unsigned int i) is left. We will only need the derivative according to the first conditional variable which is the state. To calculate this derivative again the current conditional arguments are necessary:

```
ColumnVector state = ConditionalArgumentGet(0);
ColumnVector vel = ConditionalArgumentGet(1);
```

The derivative according to the state is a 3x3 matrix calculated by:

```
Matrix df(3,3);
df(1,1)=1;
df(1,2)=0;
df(1,3)=-vel(1)*sin(state(3));
df(2,1)=0;
df(2,2)=1;
df(2,3)=vel(1)*cos(state(3));
df(3,1)=0;
df(3,2)=0;
df(3,3)=1;
```

Finally the function still has to return the calculated derivative:

```
return df;
```

The implementation of our own system model is terminated. We can now create an instance of the non linear conditional probability density function.

```
NonLinearAnalyticConditionalGaussianMobile sys_pdf(system_Uncertainty);
```

Finally we create the system model from the system pdf:

```
AnalyticSystemModelGaussianUncertainty sys_model(&sys_pdf);
```

2.4.3 The linear measurement model

We'll now build the measurement model, to correct the prediction of the position of the mobile robot, based on the distance measurement to the wall. Compared to the measurement model of the previous tutorial example (the Kalman filter), not much changes are needed except adding an extra zero to the measurement matrix (for the orientation in the state). The matrix H:

```
double wall_ct = 2/(sqrt(pow(RICO_WALL,2.0) + 1));
Matrix H(MEAS_SIZE,STATE_SIZE);
H = 0.0;
H(1,1) = wall_ct * RICO_WALL;
H(1,2) = 0 - wall_ct;
H(1,3) = 0.0;
```

All other steps are identical to the previous tutorial example.

2.4.4 Prior Distribution

As compared to the previous tutorial example we need to add an extra state variable, the orientation, to the prior distribution.
The mean which is the initial estimated state therefore becomes:

```
ColumnVector prior_Mu(STATE_SIZE);
prior_Mu(1) = PRIOR_MU_X;
prior_Mu(2) = PRIOR_MU_Y;
prior_Mu(3) = PRIOR_MU_THETA;
```

and the covariance is a diagonal matrix, with 1.0 for the position and 0.8^2 for the orientation as the σ^2 boundary:

```
SymmetricMatrix prior_Cov(STATE_SIZE);
prior_Cov(1,1) = PRIOR_COV_X;
prior_Cov(1,2) = 0.0;
prior_Cov(1,3) = 0.0;
prior_Cov(2,1) = 0.0;
prior_Cov(2,2) = PRIOR_COV_Y;
prior_Cov(2,3) = 0.0;
prior_Cov(3,1) = 0.0;
prior_Cov(3,2) = 0.0;
prior_Cov(3,3) = PRIOR_COV_THETA;
```

The mean and covariance together define a three dimensional Gaussian distribution:

```
Gaussian prior(prior_Mu, prior_Cov);
```

2.4.5 Construction of the filter

In this example we want to use an extended Kalman filter to estimate the unknown position of the mobile robot. Therefore the filter construction is identical to the previous tutorial example:

```
ExtendedKalmanFilter filter(&prior_cont);
```

2.4.6 Mobile Robot Simulator

Again, an instance of an extra class, MobileRobot, is created to simulate the mobile robot. All steps are identical to the previous tutorial example.

2.4.7 Update of the filter and new estimation

All steps for updating the filter and getting a new estimate are identical to the previous tutorial example.

2.4.8 Results

Results of the implementation are shown in Figure 2.4 without making use of the measurement information and Figure 2.5 with measurement information.

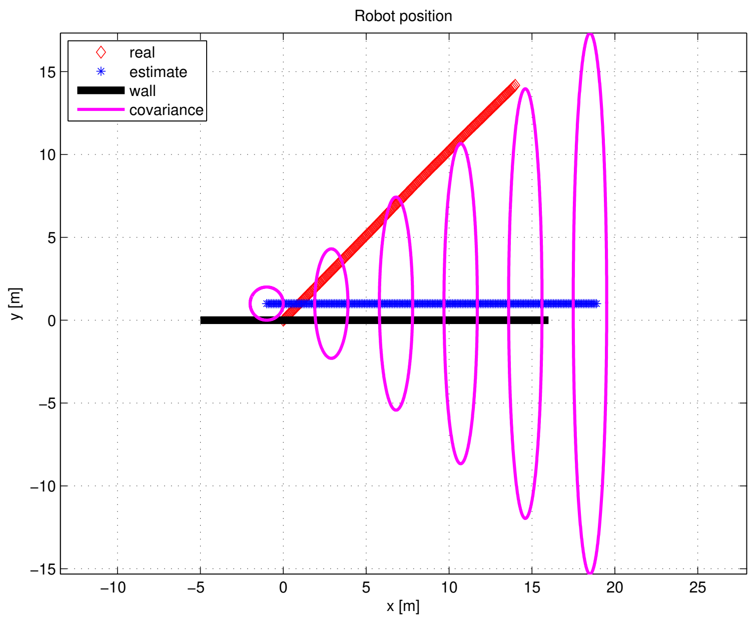

Figure 2.4: Result of estimation without measurement information

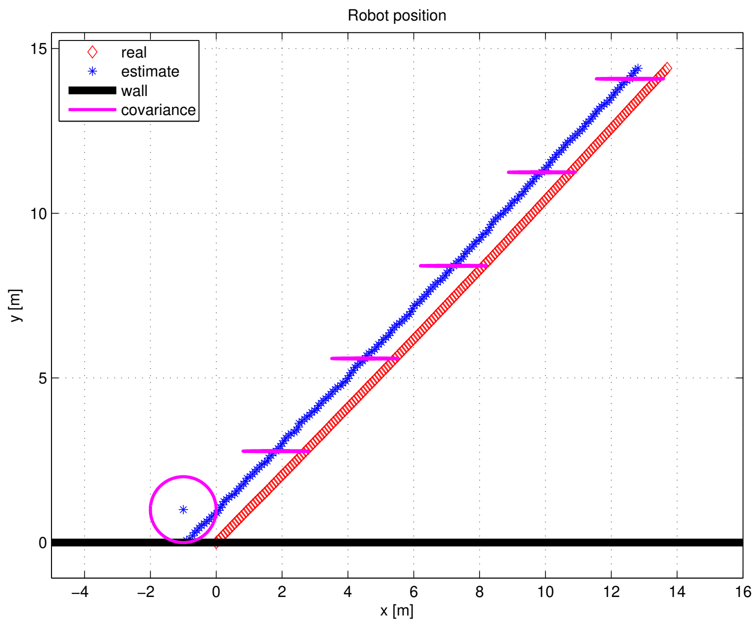

Figure 2.5: Result of estimation with measurement information

2.5 A particle filter with non linear system model and non linear measurement model in only 10 minutes extra

After completing the tutorial of the extended Kalman filter, this section will guide you step-by-step through the implementation of a particle filter using an non linear system model and a non linear measurement model. This tutorial assumes you have completed the previous tutorial.

In this third example we use a particle filter to estimate the position and the orientation of a mobile robot. The mobile robot drives around in an open space with one wall. Again, using a distance sensor, the mobile robot measures its distance to this wall.

2.5.1 Preparing the .cpp file

In the folder

```
BFL/examples/nonlinear_particle/
```

you find the source file called

```
test_nonlinear_particle.cpp
```

The beginning of the file contains the same C++ overhead as the first and second tutorial except for the fact we have to include the proper filter and the nonlinear system model and measurement model we are going to construct in this section:

```
#include <filter/bootstrapfilter.h>
#include "nonlinearSystemPdf.h"
#include "nonlinearMeasurementPdf.h"
```

2.5.2 The nonlinear system model

For this particle filter, we could directly use the non linear system model that was presented in the previous example with the extended Kalman filter. However, we wanted to make this system model more general, and show how to build a general non linear system model that is not limited to Gaussian additional noise.

The noise on the system model is still considered Gaussian (just because this is easy, but you're not limited by Gaussian noise like in the previous tutorial examples with Kalman filters). we create a Gaussian distribution, which is defined by a mean (Mu) and a covariance (Cov). This Gaussian distribution represents the uncertainty on the predicted position of the mobile robot. The mean is chosen to be $MU\_SYSTEM\_NOISE\_X$, $MU\_SYSTEM\_NOISE\_Y$ and $MU\_SYSTEM\_NOISE\_THETA$ for the x, y-position and angle respectively :

```
ColumnVector sys_noise_Mu(STATE_SIZE);
sys_noise_Mu(1) = MU_SYSTEM_NOISE_X;
sys_noise_Mu(2) = MU_SYSTEM_NOISE_Y;
sys_noise_Mu(3) = MU_SYSTEM_NOISE_THETA;
```

and the covariance is chosen as a diagonal matrix, with 0.01^2 as the σ^2 boundary:

```
SymmetricMatrix sys_noise_Cov(STATE_SIZE);
sys_noise_Cov = 0.0;
sys_noise_Cov(1,1) = SIGMA_SYSTEM_NOISE_X;
sys_noise_Cov(1,2) = 0.0;
sys_noise_Cov(1,3) = 0.0;
sys_noise_Cov(2,1) = 0.0;
sys_noise_Cov(2,2) = SIGMA_SYSTEM_NOISE_Y;
sys_noise_Cov(2,3) = 0.0;
sys_noise_Cov(3,1) = 0.0;
sys_noise_Cov(3,2) = 0.0;
sys_noise_Cov(3,3) = SIGMA_SYSTEM_NOISE_THETA;
```

The mean and covariance together define a three dimensional Gaussian distribution:

```
Gaussian system_Uncertainty(sys_noise_Mu, sys_noise_Cov);
```

Now we have to create a general non linear conditional probability density function (pdf) which represents the probability of the predicted position given the current position of the mobile robot. BFL does not provide a class for such a non linear conditional pdf, and therefore we have to implement it ourselves. We give the class we'll create a specific name for this example: NonlinearSystemPdf. BFL provides a general class of a conditional pdf from which we can inherit the functionality: ConditionalPdf. This is the most general conditional pdf.

To generate our own class we start with the implementation of the header file: *nonlinearSystemPdf.h.*

We start the class implementation with the inclusion of the header of the class we inherit from:

```
#include <pdf/conditionalpdf.h>
```

The class will be created in the BFL namespace and obviously needs a constructor and destructor. The constructor expects a Gaussian representing the system noise.

```
namespace BFL
{
 class NonLinearSystemPdf:
   public ConditionalPdf<MatrixWrapper::ColumnVector, MatrixWrapper::ColumnVector>
 {
  public:
   NonlinearSystemPdf( const Gaussian& additiveNoise);
   virtual ~NonlinearSystemPdf();
 };
}
```

When we look at the documentation of ConditionalPdf, we notice that there are many unimplemented methods. However, for a system model of a particle filter, we only need to implement one of these functions:

- bool SampleFrom (...);

This function allows the particle filter to sample from the system distribution. To implement this function, we add to following declaration to the public part of the header file:

```
virtual bool SampleFrom (Sample<MatrixWrapper::ColumnVector>& one_sample, ...)
```

The header file is ready, and we can now switch to the cpp file for the implementation: *nonlinearSystemPdf.cpp.*
The conditional arguments of the conditional pdf, the state and the input (velocity), are private variables of the conditional pdf.
Now, we switch to the implementation of SampleFrom(...). To calculate the expected value of the conditional pdf, the current state and input are necessary:

```
ColumnVector state = ConditionalArgumentGet(0);
ColumnVector vel  = ConditionalArgumentGet(1);
```

The expected value is then calculated and returned by:

```
state(1) += cos(state(3)) * vel(1);
state(2) += sin(state(3)) * vel(1);
state(3) += vel(2);
```

Then we take one sample from the additive Gaussian noise:

```
Sample<ColumnVector> noise;
_additiveNoise.SampleFrom(noise, method, args);
```

And finally store the result in the output variable of this function:

```
one_sample.ValueSet(state + noise.ValueGet());
```

The implementation of our own system model is terminated. We can now create an instance of the non linear conditional probability density function.

```
NonlinearSystemPdf sys_pdf(system_Uncertainty);
```

Finally we create the system model from the system pdf:

```
SystemModel<ColumnVector> sys_model(&sys_pdf);
```

Remark: The code

```
Sample<ColumnVector> noise;
```

causes an allocation every time SampleFrom is called and therefore breaks realtime execution. For realtime use noise should be allocated during construction.

2.5.3 The nonlinear measurement model

For this particle filter, we could directly use the measurement model that was presented in the previous example with the extended Kalman filter. However, we wanted to make this measurement model more general, and show how to build a general non linear measurement model that is not limited to Gaussian measurement noise.

The measurement noise of this measurement model is still considered Gaussian (just because this is easy, but you're not limited by Gaussian noise like in the previous tutorial examples with Kalman filters). we create a Gaussian distribution, which is defined by a mean (Mu) and a covariance (Cov). This Gaussian distribution represents the uncertainty on the distance measurement of the mobile robot distance sensor. The mean is chosen to be MU_MEAS_NOISE:

```
  ColumnVector meas_noise_Mu(MEAS_SIZE);
  meas_noise_Mu(1) = MU_MEAS_NOISE;
```

and the covariance is chosen as a diagonal matrix, with $SIGMA\_MEAS\_NOISE$ as the σ^2 boundary:

```
  SymmetricMatrix meas_noise_Cov(MEAS_SIZE);
  meas_noise_Cov(1,1) = SIGMA_MEAS_NOISE;
```

The mean and covariance together define a three dimensional Gaussian distribution:

```
  Gaussian measurement_Uncertainty(meas_noise_Mu, meas_noise_Cov);
```

Now we have to create a general non linear conditional probability density function (pdf) which represents the probability of the measured distance, given the current position of the mobile robot. BFL does not provide a class for such a non linear conditional pdf, and therefore we have to implement it ourselves. We give the class we'll create a specific name for this example: NonlinearMeasurementPdf. BFL provides a general class of a conditional pdf from which we can inherit the functionality: ConditionalPdf. This is the most general conditional pdf.

To generate our own class we start with the implementation of the header file: *nonlinearMeasurementPdf.h.*

We start the class implementation with the inclusion of the header of the class we inherit from:

```
  #include <pdf/conditionalpdf.h>
```

The class will be created in the BFL namespace and obviously needs a constructor and destructor. The constructor expects a Gaussian representing the measurement noise.

```
  namespace BFL
  {
   class NonLinearMeasurementPdf:
     public ConditionalPdf<MatrixWrapper::ColumnVector, MatrixWrapper::ColumnVector>
   {
    public:
     NonlinearMeasurementPdf( const Gaussian& additiveNoise);
     virtual ~NonlinearMeasurementPdf();
   };
  }
```

When we look at the documentation of ConditionalPdf, we notice that there are many unimplemented methods. However, for a measurement model of a particle filter, we only need to implement one of these functions:

- Probability ProbabilityGet(meas);

This function allows the particle filter to calculate the probability of a sensor measurement. To implement this function, we add to following declaration to the public part of the header file:

```
  virtual Probability ProbabilityGet(const MatrixWrapper::ColumnVector& measurement) const;
```

The header file is ready, and we can now switch to the cpp file for the implementation: *nonlinearMeasurementPdf.cpp*. The conditional arguments of the conditional pdf, the state and the input (velocity), are private variables of the conditional pdf.
Now, we switch to the implementation of ProbabilityGet(). To calculate the expected value of the conditional pdf, the current state and input are necessary:

```
ColumnVector state = ConditionalArgumentGet(0);
ColumnVector vel  = ConditionalArgumentGet(1);
```

The expected value is then calculated and returned by:

```
ColumnVector expected_measurement(1);
expected_measurement(1) = 2 * state(2);
```

We return the probability of this expected value (this is the probability of the difference between the expected and the real measurement according to the additive measurement noise):

```
 return _measNoise.ProbabilityGet(expected_measurement-measurement);
```

The implementation of our own measurement model is terminated. We can now create an instance of the non linear conditional probability density function.

```
NonlinearMeasurementPdf meas_pdf(measurement_Uncertainty);
```

Finally we create the system model from the system pdf:

```
MeasurementModel<ColumnVector,ColumnVector> meas_model(&meas_pdf);
```

2.5.4 Discrete Prior Distribution

To use a particle filter, we need a discrete prior distribution which is represented by samples. In this case we already have a continuous prior distribution of the previous examples and can therefore use it to create the discrete distribution. For convenience there is a constant in *mobile_robot_wall_cts.h* representing the number of samples:

```
#define NUM_SAMPLES  2000
```

We obtain the prior samples by sampling from the previous created continuous prior distribution and store this samples in a vector:

```
vector<Sample<ColumnVector> > prior_samples(NUM_SAMPLES);
prior.SampleFrom(prior_samples,NUM_SAMPLES,CHOLESKY,NULL);
```

With these samples we create a discrete prior distribution, a monte carlo pdf:

```
MCPdf<ColumnVector> prior_discr(NUM_SAMPLES,3);
prior_discr.ListOfSamplesSet(prior_samples);
```

2.5.5 Construction of the filter

In this example we want to use a bootstrap filter to estimate the unknown position of the mobile robot. This bootstrap filter can be created by:

```
BootstrapFilter<ColumnVector,ColumnVector> filter(&prior_discr, 0, NUM_SAMPLES/4.0)
```

The arguments that have to be provided to the BootstrapFilter are:

- The discrete prior distribution
- The resample period if static resampling is desired (otherwise use 0)
- The resample treshold if dynamic resampling is desired (otherwise use 0)

2.5.6 Results

A movie of the moving particles can be obtained from: `http://people.mech.kuleuven.be/~tdelaet/bfl_vis/deadreckoning_bootstrap_10000.avi` for the movie without using measurement information and `http://people.mech.kuleuven.be/~tdelaet/bfl_vis/meas_bootstrap_10000.avi` for the moving with measurement information.

2.6 Conclusion

As previously shown, a lot of filters can be tested on the same problem with only a minor effort: a huge advantage of BFL! If you want to know more about the possibilities to switch between filters, please check out our test_compare_filters.cpp. This file shows how easily switching between filters can be done for the mobile robot example. You find the source file in the subdirectory compare_filters.