Nonlinear Observers based on the Functional Mockup Interface with Applications to Electric Vehicles

Jonathan Brembeck, Martin Otter, Dirk Zimmer
German Aerospace Center (DLR) Oberpfaffenhofen, Institute of Robotics and Mechatronics
Münchner Strasse 20, D-82234 Wessling, Germany
jonathan.brembeck@dlr.de, martin.otter@dlr.de, dirk.zimmer@dlr.de

Abstract

At DLR, an innovative electric vehicle is being developed that requires advanced, nonlinear control systems for proper functioning. One central aspect is the use of nonlinear observers for several modules. A generic concept was developed and implemented in a prototype to automatically generate a nonlinear observer model in Modelica, given a continuous (usually nonlinear) Modelica model of the physical system to be observed. The approach is based on the Functional Mockup Interface (FMI), by exporting the model in FMI format and importing it again in a form that enables the application of different observer designs, like EKF and UKF nonlinear Kalman Filters. The approach is demonstrated at hand of an observer for the nonlinear battery model of the electric vehicle of DLR.

Keywords: FMI, FMU, Kalman Filter, EKF, UKF

1 Introduction

The ROboMObil (Figure 1, [Bre11]), a research platform for future electro mobility is developed at the DLR Institute for Robotics and Mechatronics. Its fully centralized control architecture enables highly innovative control strategies. For most of these methods, a good knowledge of all actuator states is required. Unfortunately, many of them cannot be measured directly and therefore have to be estimated. In [Eng10] , a concept for one of the ROboMObil actuators was developed to implement recursive estimator algorithms in Modelica manually based on the Functional Mockup Interface [FMI10] , [FMI11]. This approach is enhanced in this paper such that, at least in principle, every Modelica model can be automatically utilized in a nonlinear observer. In the following sections, the utilized recursive state estimation algorithms are summarized, the implementation in Modelica is outlined, and a universal Phyton based [Phy10] FMI importer is presented. Finally, experimental results with the Lithium-Ion cells of the ROboMObil in combination with this new observer framework are demonstrated.

Figure 1: ROboMObil test drive

2 Recursive state estimation

In this chapter, the principle ideas of recursive state estimation are summarized, and its (historical) development leading to the Kalman Filter is outlined. In the second part, this algorithm is extended to nonlinear systems and finally the latest developments are sketched. Further background information, alternative formulations, and recent developments are provided in the standard book [Sim06] that is also the starting point for the following explanations.

2.1 Principles

At first, we consider an estimation of a *constant signal* on the basis of several noisy measurements. This *Weighted Least Squares Estimation* problem is well-known in system identification tasks (see, e.g., [Lju98]). Through the weighted formulation, the user can assign different levels of confidence to certain measurements (or observations). This feature is crucial for tuning Kalman Filters. The corresponding minimization problem is formulated as follows:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} = \begin{bmatrix} H_{11} & \dots & H_{1n} \\ \vdots & \ddots & \vdots \\ H_{kn} & \cdots & H_{kn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} v_1 \\ \vdots \\ v_k \end{bmatrix} \qquad (1)$$
$$E(v_i^2) = \sigma_i^2 \quad (i = 1, \dots, k)$$
$$x \in \mathbb{R}^n, y \in \mathbb{R}^k, v \in \mathbb{R}^k$$

The unknown vector x is constant and consists of n elements, y is a k-element noisy measurement vector and usually $k \gg n$. Each element of y - y_i - is a linear combination (H_{k*}) with the unknown vector *x* and the variance of the measurement noise of the i-th measurement v_i. The noise of each measurement is zero-mean and independent from each other, therefore the measurement covariance matrix is

$$R = (vv^T) = diag(\sigma_1^2, \dots, \sigma_k^2) \qquad (2)$$

The residual

$$\epsilon_y = \underbrace{(Hx + v)}_{=y} - H\hat{x} \qquad (3)$$

is the difference of all measured values y with the (unknown) *x*-vector minus the estimated vector $\hat{y}$ that is computed from the estimated vector $\hat{x}$. The goal is to compute the estimated vector $\hat{x}$ such that the weighted residual is as small as possible, i.e., to minimize the cost function *J*:

$$J = \frac{\epsilon_{y1}^2}{\sigma_1^2} + \dots + \frac{\epsilon_{yk}^2}{\sigma_k^2} \qquad (4)$$

To minimize J, it is useful to compute the partial derivative with respect to the estimated $\hat{x}$ vector and set it to zero. In this way, an optimal solution for $\hat{x}$ can be calculated:

$$\frac{\partial J}{\partial \hat{x}} = 2 \cdot (-y^T R^{-1} H + \hat{x}^T H^T R^{-1} H) = 0 \qquad (5)$$
$$\hat{x} = (H^T R^{-1} H)^{-1} H^T R^{-1} y$$

(5) requires that R is nonsingular and H has full rank. This is the "textbook" version of the algorithm. It is inefficient and numerically not reliable.

Alternatively, (4) can be formulated as:

$$J = \begin{bmatrix} \frac{\epsilon_{y1}}{\sigma_1} & \dots & \frac{\epsilon_{yk}}{\sigma_k} \end{bmatrix} \cdot \begin{bmatrix} \frac{\epsilon_{y1}}{\sigma_1} \\ \vdots \\ \frac{\epsilon_{yk}}{\sigma_k} \end{bmatrix} \qquad (6)$$

To solve the following standard linear least squares problem that minimizes the Euclidian norm of the weighted residue vector:

$$\begin{aligned} &\min_{\hat{x}} \left\| \begin{bmatrix} \frac{\epsilon_{y1}}{\sigma_1} & \dots & \frac{\epsilon_{yk}}{\sigma_k} \end{bmatrix} \right\|^2 \\ &= \min_{\hat{x}} \|W(y - H\hat{x})\|^2 \\ &= \min_{\hat{x}} \|WH\hat{x} - Wy\|^2 \\ &= \min_{\hat{x}} \|A\hat{x} - b\|^2 \\ &W = diag(1/\sigma_1, \dots, 1/\sigma_k) \end{aligned} \qquad (7)$$

This minimization problem has a unique solution, if $A=WH$ has *full rank*. If A is *rank deficient*, an infinite number of solutions $\hat{x}$ exists. The usual approach is to select from the infinite number of solutions the unique one that additionally minimizes the norm of the solution vector: $\|\hat{x}\|^2 \rightarrow min$. Given $A = WH$ and $b = Wy$, this solution vector can be computed with the Modelica function Modelica.Math.Matrices.leastSquares(...) from the Modelica Standard Library which is a direct interface to the LAPACK function DGELSX [Lap99].

This function uses a QR decomposition of A with column pivoting together with a right multiplication of an orthogonal matrix *Z* to arrive at:

$$\min_{\hat{x}} \left\| [Q_1 \quad Q_2] \begin{bmatrix} U & 0 \\ 0 & 0 \end{bmatrix} ZP\hat{x} - b \right\|^2 \qquad (8)$$

where *Q* and *Z* are orthogonal matrices, *P* is a permutation matrix, *U* is a regular, upper triangular matrix and the dimension of the quadratic matrix *U* is identical to the rank of *A*. Since the norm of a vector is invariant against orthogonal transformations, this equation can be transformed to:

$$\min_{\hat{x}} \left\| \begin{bmatrix} U & 0 \\ 0 & 0 \end{bmatrix} ZP\hat{x} - \begin{bmatrix} Q_1^T b \\ Q_2^T b \end{bmatrix} \right\|^2 \qquad (9)$$

This is equivalent to

$$\min_{\hat{x}} \left\| \begin{bmatrix} U \\ 0 \end{bmatrix} \bar{\hat{x}}_1 - \begin{bmatrix} Q_1^T b \\ Q_2^T b \end{bmatrix} \right\|^2, \overline{\hat{x}} = ZP\hat{x} \qquad (10)$$

from which the solution can be directly computed as (taking into account $b = Wy$):

$$\hat{x} = PZ^T U^{-1} Q_1^T W y \qquad (11)$$

In the following, only textbook versions of algorithms will be shown, such as (5). Their implementation is, however, performed in an efficient and numerically reliable way, such as (11), where matrices R and H can be rank deficient.

The sketched approach, both (5) and (11), can be used for *offline estimation* with a predetermined number of measurements *k*.

In real-time applications, new measurements arrive in each sample period to improve the estimation. Using (11) would require a complete recalculation with $O(k^3)$-flops. One approach could be to use a moving horizon and to forget the older measurements (still

costly). Another option is to reformulate the problem into a recursive form that is updated at every sample instant with the new measurements. A linear recursive estimator can be written in the following representation:

$$\begin{aligned} y_k &= H_k x + v_k \\ \hat{x}_k &= \hat{x}_{k-1} + K_k \cdot (y_k - H_k \hat{x}_{k-1}) \end{aligned} \quad (12)$$

We compute $\hat{x}_k$ based on the estimation from the last time step $\hat{x}_{k-1}$ and the information from the new measurement y_k. K_k is the estimator gain vector that weights the correction term $y_k - H_k \hat{x}_{k-1}$. Hence, we have to compute an optimal K_k in a recursive way. To this end, it is necessary to formulate another cost function that minimizes the covariance in a recursive way.

$$\frac{\partial J_k}{\partial K_k} = \frac{\partial Tr P_k}{\partial K_k} = 0 \quad (13)$$

$$P_k = (I - K_k H_k) P_{k-1} (I - K_k H_K)^T + K_k R_k K_k^T \quad (14)$$

$$K_k = P_{k-1} H_k^T \cdot (H_k P_{k-1} H_k^T + R_k)^{-1} \quad (15)$$

This results in a recursive formula to update the estimation of the unknown, but constant, vector x in every sample with the latest measurements, based only on the estimation from the last sample. Table 1 summarizes the whole algorithm.

Table 1: Recursive weighted least squares algorithm

Initialization

$$\hat{x}_0 = E(x)$$
$$P_0 = E[(x - \hat{x}_0)(x - \hat{x}_0)]$$

For $k = 1,2, \ldots$

$$y_k = H_k x + v_k$$
$$K_k = P_{k-1} H_k^T \cdot (H_k P_{k-1} H_k^T + R_k)^{-1}$$
$$\hat{x}_k = \hat{x}_{k-1} + K_k \cdot (y_k - H_k \hat{x}_{k-1})$$
$$P_k = (I - K_k H_k) P_{k-1} (I - K_k H_K)^T + K_k R_k K_k^T$$

For many real-time control problems, it is more interesting to estimate the system states rather than some constant parameters. Therefore the *linear Kalman Filter* was developed in the 60's. It enables to estimate the system states of a linear discrete-time model in a recursive way. The fundamental assumption is that the system and the output equations are disturbed by white Gaussian noise. Both of these noise processes are regarded as uncorrelated with zero mean. This results in the following equations:

$$\begin{aligned} x_k &= F_{k-1} x_{k-1} + G_{k-1} u_{k-1} + w_{k-1} \\ y_k &= H_k x_k + v_k \\ E(w_k w_j^T) &= Q_k \delta_{k-j} \\ E(v_k v_k^T) &= R_k \delta_{k-j} \\ E(w_k v_k^T) &= 0 \end{aligned} \quad (16)$$

At this point, we introduce the principle of every Kalman Filter derivation (compare Figure 2). Subsequent to filter initialization, the first step in every sample is the a-priori estimation of the mean (system states) and the covariance (a gauge for the confidence in them). This is called the prediction step and all of the equations that are related with it contain a "-" in the superscript.

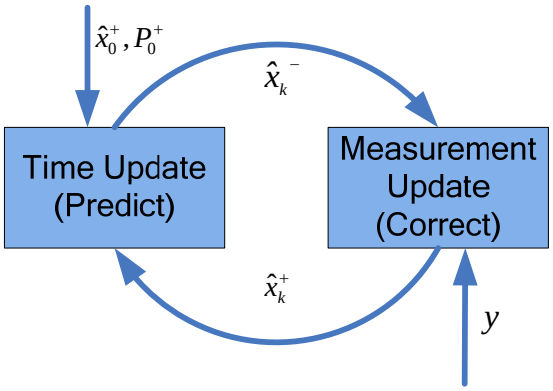

Figure 2: Principle of recursive Kalman filter.

This forms the basis for the calculation of the optimal Kalman gain that is used to correct the estimated state vector with the information from the actual measurements. Finally, the covariance matrix is updated. This is called the correction step. In the next sample, these values are used to restart again at the subsequent prediction step. The algorithm can be formulated as follows:

Table 2: Linear discrete Kalman Filter

Initialization

$$\hat{x}_0 = E(x_0)$$
$$P_0^+ = E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)]$$

For $k = 1,2, \ldots$

$$\hat{x}_k^- = F_{k-1} x_{k-1}^+ + G_{k-1} u_{k-1}$$
$$P_k^- = F_{k-1} P_{k-1}^+ F_{k-1}^T + Q$$
$$K_k = P_k^- H_k^T \cdot (H_k P_k^- H_k^T + R)^{-1}$$
$$\hat{x}_k^+ = \hat{x}_k^- + K_k \cdot (y_k - H_k x_k)$$
$$P_k^+ = (I - K_K \cdot H_k) \cdot P_k^-$$

To determine the relationship between the *Kalman Filter* and *recursive weighted lest squares*, we should have a closer look at Table 2. The matrix $Q(ww^T) = diag(\sigma_{w1}^2, \ldots, \sigma_{wn}^2)$ represents the covariance of the system states (w denotes the variance of the system states). Its entries represent the confidence in the a-priori estimation and can be tuned by the application engineer. Large values represent high uncertainty (probably due to an imprecise model), whereas small values indicate good trust. The second tuning matrix R represents the confidence in the actual measure-

ments. Its effect resembles our first estimation problem (eq. (1) to (5)). Furthermore, it can be shown that if x_k is a constant vector then $F_k = I, Q_k = 0$ and $u_k = 0$. In this case, the *Linear discrete Kalman Filter algorithm* (Table 2) reduces to the *recursive weighted least squares algorithm* (Table 1). This property is often exploited in the formulation of parameter estimation problems using *Kalman Filter algorithms*.

2.2 Nonlinear Kalman Filter Algorithms

So far, we have discussed estimation problems for linear discrete systems. This is generalized to nonlinear systems starting from a continuous-time representation in state space form:

$$\begin{aligned} \dot{x} &= f(x,u) \\ y &= g(x) \end{aligned} \tag{17}$$

In section 3, it is sketched how such a model description can be generated from a Modelica model for use in a *nonlinear Kalman Filter* using the Functional Mockup Interface. In this way, it is possible to formulate the synthesis models for the prediction step (see Figure 2) with Modelica, even in implicit representation, and shift all tedious tasks to the *Nonlinear Observer* framework. This avoids calculus mistakes and allows us to put the main focus on the design of the algorithms.

In *Table 3*, the widely used extension of the *discrete linear Kalman Filter* to the *discrete nonlinear Kalman Filter with additive noise* is presented. The dynamic system is represented as follows:

$$\begin{aligned} x_k &= f_{k-1}(x_{k-1}, u_{k-1}) + w_{k-1} \\ y_k &= h_k(x_k) + v_k \\ \mathrm{w_k} &\cong (0, \mathrm{Q_k}) \\ \mathrm{v_k} &\cong (0, \mathrm{R_k}) \end{aligned} \tag{18}$$

The algorithm is very similar to a purely linear one. To handle the nonlinearity, the system is linearized around the last estimation point using a Taylor Series Expansion up to the first term. This can be performed numerically by the use of a forward difference formula.

Table 3: Extended Kalman Filter Algorithm

Initialization

$$\hat{x}_0 = E(x_0)$$
$$P_0^+ = E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)]$$

For $k = 1,2,\ldots$

$$\hat{x}_k^- = f_{k-1}(\hat{x}_{k-1}^+, u_{k-1})$$
$$P_k^- = F_{k-1} P_{k-1}^+ F_{k-1}^T + Q$$
$$where\ F_{k-1} = \left.\frac{\partial f_{k-1}}{\partial x}\right|_{\hat{x}_{k-1}^+}$$
$$K_k = P_k^- H_k^T \cdot \left(H_k P_k^- H_k^T + R\right)^{-1}$$
$$where\ H_k = \left.\frac{\partial h_k}{\partial x}\right|_{\hat{x}_k^-}$$
$$\hat{x}_k^+ = \hat{x}_k^- + K_k \cdot \left(y_k - h_k(\hat{x}_k^-)\right)$$
$$P_k^+ = (I - K_K \cdot H_k) \cdot P_k^-$$

Since we have a nonlinear continuous-time system representation, we have to linearize and discretize our system at every sample instant. Discretization means to integrate the system in the prediction step from the last sample instant to the new one, e.g. with the *Trapezoidal* or the *Runge-Kutta 4* integration method. The transition matrix F_{k-1} is calculated by an analytic derivation of the system state Jacobian. An alternative is the numerical calculation with, e.g., a forward difference formula:

For $i = 1,2,\ldots,n$

$$J_{\hat{x}_{k-1}^+}^{[:,i]} = \frac{f(\hat{x}_{k-1}^+ + h \cdot E(:,i), u) - f(\hat{x}_{k-1}^+, u)}{h} \tag{19}$$

The transition matrix can be computed with function Modelica.Math.Matrices.exp from the Modelica Standard Library resulting in:

$$F_{k-1} = e^{\left(J_{\hat{x}_{k-1}^+} \cdot T_s\right)} \tag{20}$$

The same procedure is necessary to calculate the output Jacobian H_k. Using this method, it is possible to use a nonlinear continuous-time system within the *discrete nonlinear Kalman Filter* algorithm.

The discussed EKF algorithm is widley used in many applications. However, it often gives unsatisfactory results or even does not converge if the system nonlinearities are severe because the linearization causes a propagation of the mean and covariance that is only valid up to the first order. The following section sketches the principles of the *Unscented Kalman Filter* (UKF) and its advantages in nonlinear state estimation.

2.3 Unscented Kalman Filter

In order to achieve higher accuracy, the UKF calculates the means and covariances from disturbed state vectors, called sigma points, by using the nonlinear system description. As one side effect, the Jacobians of f(x) and h(x) are no longer needed. See [Mer04] for more detailed information. The structure of the equation set, containing prediction and update, is similar to the EKF. However, the calculation of the covariances requires to integrate the nonlinear system $2n + 1$ times from the last to the actual time instant and is therefore computationally costly. The symmetry of all the involved matrices is fully exploited to reduce computational costs. An additional reduction of computational effort is achieved with the Square Root UKF (SR-UKF).

2.4 Square Root Unscented Kalman

The equations of the SR-UKF are identical to the UKF, but the structure is utilized during the evaluation: Although the covariance matrix P_k and the predicted covariance matrix P_k^- are uniquely defined by their Ckolesky factors $\sqrt{P_k}$ and $\sqrt{P_k^-}$ respectively, with UKF the covariance matrices are calculated at each step. Furthermore, the sigma points X_k can be computed with the Cholesky factor $\sqrt{P_k}$, and the updated sigma points of the measurement update with the Cholesky factor $\sqrt{P_k^-}$ without using the covariance matrices. Moreover, the gain matrix K_k is determined as solution of the linear equation system

$$K_k \cdot P_{y_k y_k} = P_{x_k y_k} \qquad (21)$$

that can be more efficiently solved by utilizing again the Cholesky factorization. In the SR-UKF implementation, the Cholesky factors are propagated directly and the refactorization of the covariance matrices is avoided [Mer01b] .

The EKF, UKF, and SR-UKF algorithms are implemented as Modelica functions using LAPACK for core numerical computations. Implementation details of the numerical algorithms will be provided in an upcoming publication by Marcus Baur.

3 Nonlinear Observers in Modelica

In this section a prototype implementation is sketched for applying the nonlinear observers from the previous section to Modelica models. The goal is to start from a given (continuous, usually nonlinear) Modelica model and provide automatically a nonlinear observer for this model in form of a sampled data system.

This task cannot be performed directly, because Modelica has no means to discretize a continuous model and to solve this discretized model with a user-defined method (= integration + update of the next state according to the observer equations).

Note, it is insufficient to simply integrate the nonlinear models from the last to the new sample instant (which could be achieved by using the "mapping" annotation introduced in Modelica 3.1). Instead, the extended Kalman filter additionally requires linearizing the model around the sample time and using it together with the solution of the integration to compute a new estimation of the state that is utilized in the next step. On the other hand, the unscented Kalman filter requires integrating the model several times with disturbed states from the last to the new sample instant.

To summarize, there is no way to describe a nonlinear observer completely in Modelica and it is also very unlikely that the Modelica language is extended so that this becomes possible.

The basic approach is to export the Modelica model in the FMI-format (see section 3.1), import it again in Modelica and during import call the FMI-functions in such a way that the model is discretized and utilized in a nonlinear observer algorithm.

3.1 Functional Mockup Interface

The Functional Mock-up Interface (FMI) for Model Exchange [FMI10] , [FM11] was developed in the MODELISAR project to standardize the exchange of dynamic models between tools. This interface is supported already by Dymola, SimulationX, JModelica.org, Silver and Simulink[1]. Other tools are planning to support it as well.

The goal of the FMI is to describe input/output blocks of dynamic systems defined by differential, algebraic and discrete equations and to provide an interface to evaluate these equations as needed in different simulation environments, as well as in embedded control systems, with explicit or implicit integrators and fixed or variable step-size. Some details of the type of systems that can be handled are shown in Figure 3 (from [FMI10]).

[1] Dymola 7.4 can export Simulink models in FMI-format via Realtime-Workshop of MathWorks.

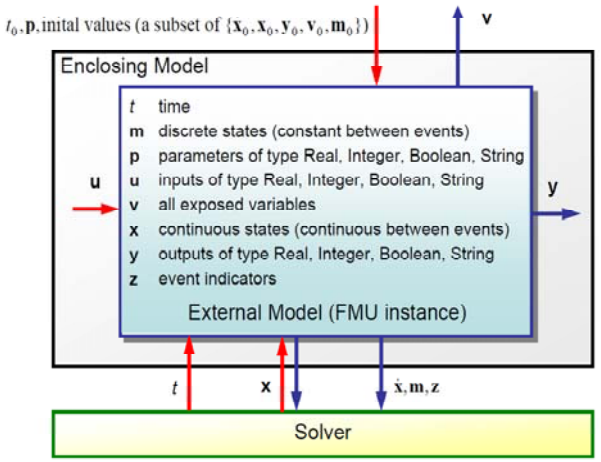

Figure 3: FMI for model exchange.

The interface consists of (a) a small set of standardized "C-functions" to evaluate the model equations and (b) an XML-file that contains all information that is not needed during execution, such as the variable definitions. Every variable has a handle (a 32 bit Integer) that is used to identify the variable in the C-function calls. The source and/or object code of the C-functions, as well as the XML-file and optionally other files, are stored in a zip-file with the extension ".fmu" for "Functional Mockup Unit".

In order to implement nonlinear observers for Modelica models, the corresponding model has to be exported by one of the tools in FMI format. In a subsequent step, it has to be imported again. Unfortunately, a standard FMU-import as supported by Dymola and other tools cannot be used, because these interfaces import a model as continuous model, if it was exported as continuous model. For this reason, a new FMU-import method was implemented (see section 3.3). From the Modelica perspective, it was necessary to use the new feature of "functions as input argument to functions", as introduced in the Modelica Language 3.2. This feature is currently only supported in Dymola 7.5 Beta. So we used this Dymola version for the prototype implementation.

3.2 FMU Definition in Modelica

The key point is that all FMI-functions of an imported FMU need to be available for design methods in Modelica. This is achieved in the following way:

1. A FMU (so a model exported by a Modelica tool) is mapped to a replaceable package consisting of (a) an external object that holds the "internal memory" of the model, (b) external functions that call the FMU functions, and (c) a Modelica model to instantiate and initialize the external object optionally defining new values for the parameters. This is a similar approach as used for media from the Modelica.Media package.
2. Design functions, such as computing the new estimated state of a model, are implemented in a model independent way. This is achieved by providing functions (as input arguments) that compute the needed information from a model. Concrete implementations of these functions are provided for FMUs.

Here is a more detailed sketch of this approach:

Package PartialFmiFunctions defines the interfaces to all FMI functions:

```
partial package PartialFmiFunctions
  constant Integer nx=1 "# of states";
  constant Integer nu=1 "# of inputs";
  constant Integer ny=1 "# of outputs";
  constant Integer id_u[nu]"Input handles";
  constant Integer id_y[ny]"Output handles";

  replaceable partial class FmiInstance
    extends ExternalObject;
    replaceable partial function constructor
      input String       instanceName;
      input Boolean      loggingOn;
      output FmiInstance fmi;
    end constructor;
    replaceable partial function destructor
      input FmiInstance fmi;
    end destructor;
  end FmiInstance;

  replaceable partial function fmiSetTime
    input  FmiInstance fmi
    input  Real ti;
    input  Real preAvail;
    output Real postAvail = preAvail;
  end fmiSetTime;

  replaceable partial function
                      fmiSetContinuousStates
    input  FmiInstance fmi;
    input  Real x[:];
    input  Real preAvail;
    output Real postAvail= preAvail;
  end fmiSetContinuousStates;

  ...
end PartialFmiFunctions;
```

It is important that the dimensions of the input, output and state vectors, as well as the vector of handles for the input variables (id_u) and for the output variables (id_y) are available in the package as constants, since they are needed later by the specialized functions for the design models.
Importing an FMU means to *generate* a FMU specific Modelica *package* of the form (below: <MODEL> is the name of the FMU):

```
package <MODEL>_fmu
  model Model
    // Define parameters of the FMU
    // Define inputs, outputs of the FMU
    // initialize FMU
```

```
    parameter String name = "<MODEL>"
    Functions.FmiInstance fmi=
            Functions.FmiInstance(name);
    ...
  end Model;

  package Functions
    extends PartialFmiFunctions(
      nx=4,
      nu=1,
      ny=2,
      id_u={352321536},
      id_y={335544320,335544321});

    redeclare class FmiInstance
      extends ExternalObject;
      function constructor
        input String instanceName;
        input Boolean loggingOn;
        output FmiInstance fmi;
        external"C" fmi = <MODEL_init>
           (instanceName, loggingOn);
      end constructor;
      function destructor
        input FmiInstance fmi;
        external"C" <MODEL_close>(fmi);
      end destructor;
    end FmiInstance;

    redeclare function extends fmiSetTime
      external"C"
       <MODEL_ fmiSetTime>(fmi, ti);
    end fmiSetTime;
    …
  end Functions;
end <MODEL>_fmu;
```

The imported FMU is now available as a package that contains a model to initialize the FMU and a set of functions to operate on the initialized FMU.
Up to this stage, the code is completely independent from the design that shall be carried out, and the generated FMU package can be utilized for all kinds of design tasks. For every specific design, like an UKF observer, a model has to be implemented that has the following basic structure:

```
model UKF_FMI "Unscented Kalman filter"
  import C =
         Modelica_LinearSystems2.Controller;
  import I = Modelica.Blocks.Interfaces;
  extends C.Interfaces.PartialDiscreteBlock
      (initType = C.Types.Init.InitialState);

  replaceable package FmiFunctions =
                        PartialFmiFunctions;
  constant Integer nx = FmiFunctions.nx;
  constant Integer ny = FmiFunctions.ny;
  constant Integer nu = FmiFunctions.nu;
  parameter Real Q[nx,nx]=identity(nx);
  parameter Real G[nx, nx];
  parameter Real R[ny, ny];
  parameter Real P_init[nx,nx];
  parameter Real x_init[nx] "Initial states";

  input FmiFunctions.FmiInstance fmi;
  I.RealInput  u[nu]         "Input u";
  I.RealInput  y_measure[ny] "Measured y";
  I.RealOutput x_est[nx]     "Estimated x";
  I.RealOutput y_est[ny]     "Estimated y";
  Real time_;
  Real P[nx,nx] "Error covariance matrix";
  Real K[nx,ny] "Kalman filter gain matrix";
  ...
protected
  outer C.SampleClock sampleClock ;
initial algorithm
  x_est :=FmiFunctions.fmiGetContinuousStates
                                    (fmi,nx,1);
  P     := P_init;
  time_ := 0;
algorithm
  when sampleTrigger then
    (x_est,y_est,P,K) := UKF(
          function fFMI(fmi=fmi),
          function hFMI(fmi=fmi),
          pre(x_est),pre(u),y_measure, ...);
    time_ :=time_ + sampleClock.sampleTime;
    FmiFunctions.fmiSetTime(fmi,time_,1);
    FmiFunctions.fmiCompletedStep(fmi,3);
  end when;
end UKF_FMI;
```

The UKF_FMI design model uses the PartialFmiFunctions as replaceable package to get access to the FMU functions of the model (in the same way as a medium is used in a fluid model), as well as an instance of the external object in this package (FmiInstance) to hold the internal memory of the FMU.
All data that the user has to provide for this design method is provided via parameters and input signals. The central code consists basically of a periodically evaluated when-clause where in every sample interval the UKF design function is called. This design function, here: UKF(...), is generic and does not depend on FMI. In case of the UKF, the design function requires two functions as inputs: fFMI(..) and hFMI(..). In model UKF_FMI above, these (generic) functions will internally call FMI functions, and therefore the handle to the FMU external object is provided as additional argument via a “function partial application”.
Function “fFMI” integrates the FMU over one sample period, whereas “hFMI” computes the output signals at the new sample time. For example, fFMI is implemented as:

```
function fFMI
  input FmiFunctions.FmiInstance fmi;
  input Real u[:] "Input at instant k";
  input Real x[:] "State at instant k";
  input Modelica.SIunits.Time Ts;
  output Real x_new[size(x, 1)]
                  "Predicted x at k+1";
algorithm
   FmiFunctions.fmiSetReal
          (fmi, FmiFunctions.id_u, u, 1);
   x_new := RkFix4(fmi,Ts,x);
end fFMI;
```

With “fmiSetReal”; the input values are set and with function “RkFix4” the FMU is integrated from the previous to the next sample instant using a Runge-Kutta method of order 4 with a fixed step size. The

design function “UKF” finally is an implementation of the algorithm sketched in section 2 using LAPACK [Lap99] for its numerical part.
All pieces can now be assembled together. Assume for example, that a crane model is exported as FMU and that the importer of section 3.3 generated the package “**Crane_fmu**” according to “<MODEL_fmu>” from above. Then the code for an UKF observer for this model has basically the following structure:

```
model CraneObserver
  // FMU instance
  Crane_fmu.Model CraneFMU(...);

  // Unscented Kalman Filter
  UKF_FMI UKF(
    fmi = CraneFMU.fmi,
    redeclare package FmiFunctions =
                    Crane_fmu.Functions,
    ...)

  // Connect input and measurement signals
  // to model UKF
end CraneObserver;
```

In the first statement an instance of the FMU model is generated. In the second statement, the model of the unscented Kalman filter is used and the FMU instance as well as the FMU functions are provided as arguments, besides Kalman specific settings.

3.3 FMU import using Python

To support the reimport of a FMU into a Modelica model in the specific form of section 3.2, a tool box has been developed in Python 3 [Phy10] . It consists of a library of Python classes and a set of scripts representing the end-user applications. Using this tool-box, a developer can easily create its own re-import functionality for FMUs, specially tailored to fit his or her set of demands. The result of the final Python script is Modelica package <MODEL>_fmu from the last section representing the imported FMU. The equired input consists in the XML-file that is extracted from the FMU zip-file, optionally additional text-input by the user, and most important a template file, see Figure 4. This template file consists of a Modelica model file that contains mark-up elements to be replaced by the Python Script.
The template file for FMUs for nonlinear observers resembles the structure of package “<MODEL>_fmu” sketched in section 3.2.
Using the Python tool-box, FMUs can be re-imported into Modelica in a very flexible way suiting a broad set of potential future applications.

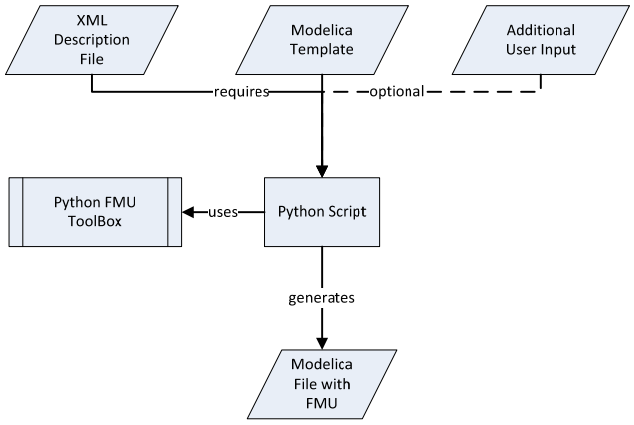

Figure 4: Processing Scheme for the Python-based FMU Re-import

4 Example SOC estimation

Subsequently, the observer framework is demonstrated in an application from the development of the ROboMObil. The battery model introduced in [Bre11b] is used as the synthesis model for the FMU-Export. The observer scheme is shown in Figure 5.

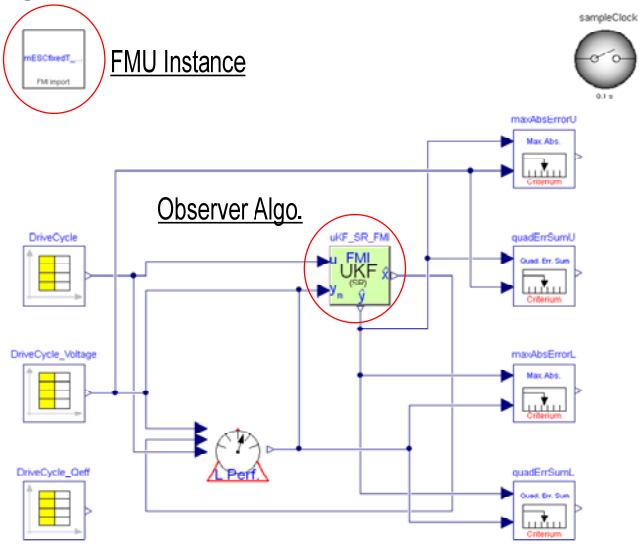

Figure 5: FMU based observer setup

In the top left corner of the model a FMU instance block is placed. The free parameters of the imported model can be tuned here before simulation. So it is possible to modify system parameters, i.e. due to changed conditions in the experiment, without the necessity of repeating the importing procedure. With these parameters and the system equations, the FMU instance calculates the initial states of the prediction model and instantiates the FMI object.

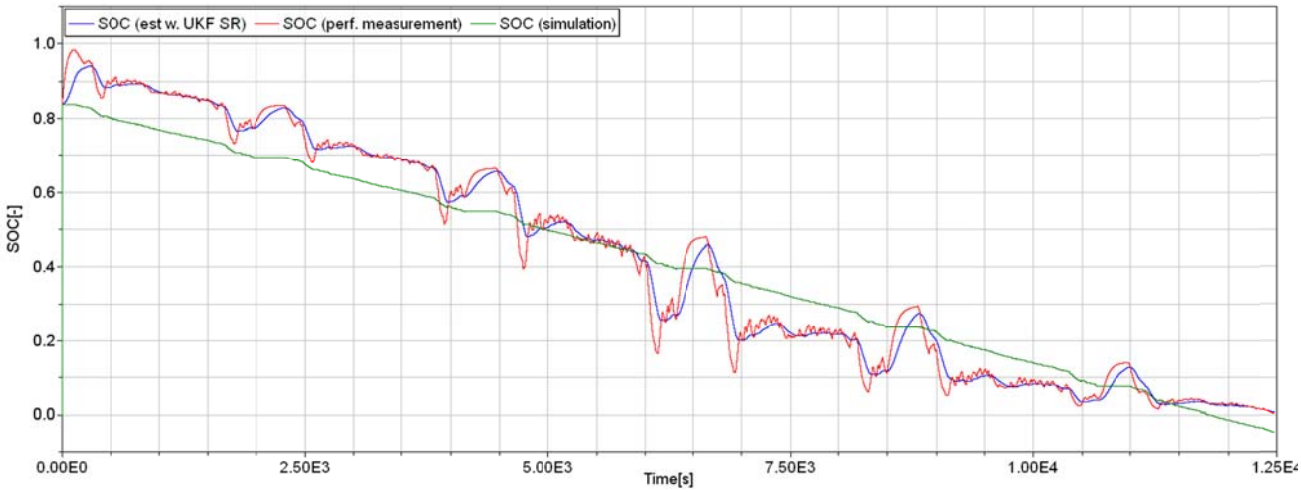

Figure 6: Comparison of observer vs. model based SOC characteristics

The pointer to this instance is passed to the observer algorithm. This can be done via the parameter dialog. In our case we have choosen an ***U**nscented**K**alman-**F**ilter* using ***S**quare**R**oot* matrix calculation. Moreover the user has to tune the free filter parameters like the covariance matrix or the sigma point spread. This has to be performed individually for every application, manually or by offline optimization methods ([Bre11b]).

In this example we like to estimate the ***S**tate**O**f**C**harge* of a LiIon battery cell. The input u of the battery model is the measured current, while the model output vector y_m is the cell voltage and the noisy SOC which is calculated via the method of perfect measurements (c.f. Figure 5, component with description text "L Perf.", where L is the state variable for the SOC).

For experiment data we use a FTP75 driving cycle which is simulated with the ROboMObil energetic model([Eng10]). The calculated electric power demand of the actuators is converted to the current demand of one cell. This is used as current demand to the single cell test bench (Figure 7). The voltage at the cell terminals, the surface temperature and the effective current flow are recorded during this test. Finally they are used as input and measurement data of the experiment setup (Figure 5 bottom left).

In Figure 6 we have presented the experiment results and benefits of using a model based recursive observer in real-time applications. The red curve shows the SOC characteristic calculated via the perfect measurement.

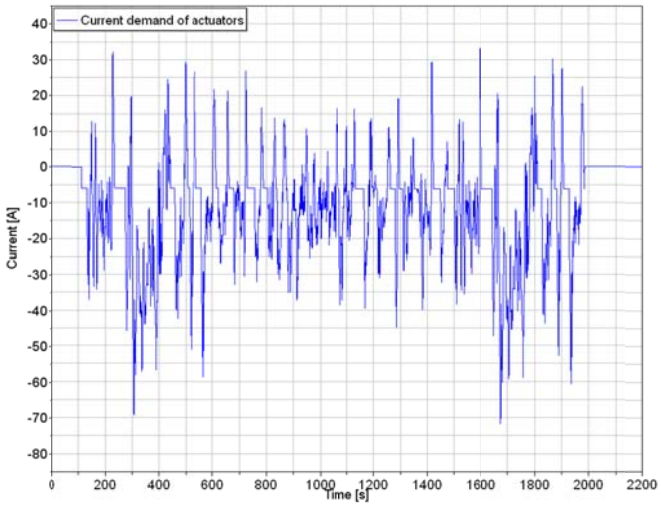

Figure 7: Current demand from ROboMObil in FTP75 drive cycle

It is, despite of signal pre-filtering, very erratic and noisy. This characteristic is qualitatively correct, especially in comparison to the green curve, which represents the output of a pure model simulation without observer correction. The pure simulation causes a SOC that is less than zero at the end of the simulation which is physically impossible (c.f. Figure 7, bottom right). In car applications, this would mean that the SOC display would show incorrect information. In this case it would not be possible to drive on, although the battery is not exhausted yet. Through our estimation algorithm, we get a better and smoother estimation of the SOC that converges to zero (blue curve) at the end. Due to the efficient code provided by the FMI interface, this test runs with a real time factor greater than 100 on standard desktop systems. Thereby, it is possible to implement this observer on embedded or rapid prototyping controllers within the ROboMObil.

5 Conclusions and future work

We have demonstrated a way to develop a framework for generic observer design. The algorithm part is completely separated from the synthesis model. This could be achieved by the use of the FMI reimport mimic and the new possibilities of Modelica 3.2 to pass functions as arguments to functions. The presented example of a battery state estimation and its results make us confident that this framework can be used for many control system tasks in the future, especially in the ROboMObil project. Furthermore, the estimation algorithms will be extended to handle constraints in a recursive way, [Sim09] [Kan08] and to take "out of sequence measurements" into account, [Lar98] [Mer04] .

6 Acknowledgement

The financial support of DLR by BMBF (Förderkennzeichen: 01IS08002) for this work within the ITEA2 project MODELISAR (http://www.itea2.org/public/project_leaflets/MODELISAR_profile_oct-08.pdf) is highly appreciated.

References

[And04] André M. (2004): **The ARTEMIS European driving cycles for measuring car pollutant emissions**. Science of The Total Environment, 334-335, pp. 73-84.

[Bre11] Brembeck J., Ho L. M., Schaub A., Satzger C., Hirzinger P. G. (2011): **ROMO – the robotic electric vehicle**. IAVSD, Aug. 14-19, accepted for publication.

[Bre11b] Brembeck J., Wielgos S. (2011): **A real time capable battery model (mESC) for electro mobility applications using optimal estimation methods**. Modelica'2011 Conference, March 20-22.

[Eng10] Engst C. (2010): **Object-Oriented Modelling and Real-Time Simulation of an Electric Vehicle in Modelica**. Master Thesis, Technische Universität München, Lehrstuhl für elektrische Antriebssysteme und Leistungselektronik. Supervisors: J. Brembeck, M. Otter, R. Kennel.

[FMI10] **The Functional Mock-up Interface for Model Exchange,Version 1.0** (2010). ITEA2 MODELISAR Project. Download: www.functional-mockup-interface.org/fmi.html

[FMI11] Blochwitz T., Otter M., Arnold M., Bausch C, Clauß C., Elmqvist H., Junghanns A., Mauss J., Monteiro M., Neidhold T., Neumerkel D., Olsson H., Peetz J.-V., Wolf S. (2011): **The Functional Mockup Interface for Tool independent Exchange of Simulation Models.** Modelica'2011 Conference, March 20-22.

[Kan08] Kandepu R., Imsland L., Foss B. (2008): **Constrained state estimation using the Unscented Kalman Filter**. Proceedings of the 16th Mediterranean Conference on Control and Automation, pp. 1453-1458, June 25-27.

[Lap99] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Sorensen D. (1999): **Lapack Users' Guide**. Third Edition, SIAM. Download: http://www.netlib.org/lapack

[Lar98] Larsen T. D., Andersen N. A., Ravn O., Poulsen N. K. (1998): **Incorporation of time delayed measurements in a discrete-time Kalman filter**. Proceedings of the 37th IEEE Conference on Decision and Control, vol. 4, pp. 3972-3977, Dec. 16-18.

[Lju98] Ljung, L. (1998): **System Identification: Theory for the User**. Prentice Hall, 2nd Edition.

[Mer01] Merwe R. V., Wan E. (2001): **The square-root unscented Kalman filter for state and parameter-estimation**. Proceedings of the International Conference on Acoustics, Speech, and Signal Processingvol. 6, pp. 3461-3464.

[Mer04] Merwe R. V., Wan E., Julier S. (2004): **Sigma-Point Kalman Filters for Nonlinear Estimation and Sensor-Fusion: Applications to Integrated Navigation**. Proceedings of the AIAA Guidance Navigation & Control Conference, Providence, RI, Aug 2004. Download: http://www.csee.ogi.edu/~rudmerwe/pubs/pdf/vanderMerwe_GNC2004.pdf

[Mod10] Modelica Association (2010): **Modelica – A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, Version 3.2**. March 24, 2010. Download: https://www.modelica.org/documents/ModelicaSpec32.pdf.

[Phy10] **Phython 3.1.2- Documentation (2010)**: Download: http://docs.python.org/py3k/

[Sim06] Simon D. (2006): **Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches**. John Wiley & Sons Inc..

[Sim09] Simon D. (2010): **Kalman filtering with state constraints: a survey of linear and nonlinear algorithms**. IET Control Theory & Applications, vol. 4, no. 8, pp. 1303-1318, Aug..