Prognostics Algorithm Library: User Manual

Matthew Daigle
NASA Ames Research Center, Moffett Field, CA 94035
matthew.j.daigle@nasa.gov

August 18, 2016

1 Introduction

Prognostics is a technical problem dealing with predicting the future state of a system of interest. *Model-based prognostics* is a framework to solve this problem based on the use of dynamic system models. The *Prognostics Algorithm Library* is a Matlab toolbox with prognostics algorithms to support this framework. It relies on the *Prognostics Model Library*, which defines a `PrognosticsModel` class. The Prognostics Algorithm Library defines a set of classes implementing state estimation and prediction algorithms. Examples are also provided showing how to use the various algorithms for example models.

The model-based prognostics paradigm is summarized in Fig. 1. There is a system being monitored, and one develops a model describing how the system evolves in time in response to its inputs. That model is used as the basis for state estimation, performed by an *observer*, and state prediction, performed by a *predictor*. This *Prognostics Model Library* toolbox addresses the problem of defining models for this purpose. This toolbox provides observer and predictor algorithms, for use with these models. A complete prognoser solution (encompassing the entire model-based prognostics box) is also provided through a class defined for this purpose.

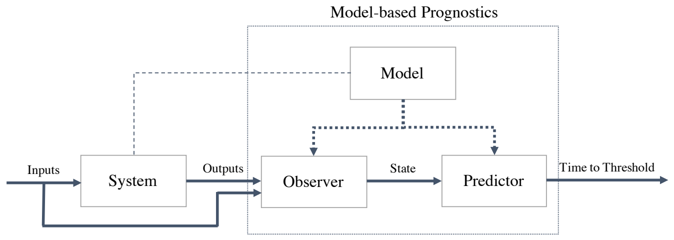

Figure 1: Model-based prognostics conceptual architecture.

1.1 Installation

The toolbox is packaged as `PrognosticsAlgorithmLibrary.mltbx`. To install the toolbox, open Matlab, double-click on the file, and click the install button. The toolbox files will be added to Matlab's user toolboxes directory and will make its path accessible to Matlab. The toolbox works with Matlab R2016a, but has been known to work down to R2012a.

1.2 Documentation

The toolbox is fully documented within Matlab, and can be accessed through Matlab's `help` and `doc` functions on any toolbox file, for example:

```
>> help Observers.ParticleFilter
>> doc Observers.ParticleFilter
```

Thus, the purpose of this user manual is to explain the theory behind the framework (Section 2), explain what the toolbox provides (Section 3), and demonstrate the toolbox through detailed examples (Section 4).

2 Background

For a given system model, the prognostics problem is focused on determining how the system will evolve in time, and when the system state will reach some region of interest. In defining a model for prognostics, one must define a state equation, an output equation, an input equation, and a threshold equation.

In the following, the system state is denoted by the vector $\mathbf{x}$, the inputs by the vector $\mathbf{u}$, and the outputs by the vector $\mathbf{z}$. The state equation defines how the state is updated from time t to $t+\Delta t$, where Δt is the sampling time:

$$\mathbf{x}(t+\Delta t) = \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t), \mathbf{v}(t)), \tag{1}$$

where $\mathbf{v}(t)$ is the process noise, and $\mathbf{f}$ is the state update function. The output equation defines how the system outputs are computed from the states and inputs:

$$\mathbf{z})(t) = \mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t), \mathbf{n}(t)), \tag{2}$$

where $\mathbf{n}(t)$ is the sensor noise, and $\mathbf{h}$ is the output function.

For prognostics, one is interested in predicting when a region of the state space is reached, and so this region must be defined. This is achieved through a *threshold equation*, $\mathbf{g}(t)$, that returns a Boolean indicating whether the system is in the given space or not. In general, this is expressed as a function of time, the states, and the inputs:

$$b = \mathbf{g}(t, \mathbf{x}(t), \mathbf{u}(t)), \tag{3}$$

where b is the Boolean output (true/false).

In order to predict how the state evolves in time, the system inputs for all future time points must also be known. This is defined through the input equation, which computes what the system inputs are for a given time and a set of *input parameters*:

$$\mathbf{u}(t) = \mathbf{i}(t, \mathbf{p}), \tag{4}$$

where $\mathbf{i}$ is the input function and $\mathbf{p}$ are the input parameters. The input parameters are defined as a set of numbers that are used to parameterize what the system inputs are at a given time. For example, consider a battery model, where the input to the model is the power required by the load. This power depends on the system usage and could take one of a number of profiles. The input parameters could be used to define such profiles, e.g., by specifying magnitudes and durations for different load segments.

The prognostics problem is then defined as, given some initial system state, with some specified uncertainty, the future inputs to the system, with some specified uncertainty, and a description of the process noise, to determine any combination of the following:

- what is the probability of the threshold being reached within a given future time,
- when the system will enter a state-input space such that the threshold equation evaluates to true, and
- what is the state of the system when the threshold is reached.

In general, these are computed through a stocastic simulation. However, in order to perform a prediction, one must first determine what the system state is at the desired time of prediction. Typically, this is solved through a state estimation algorithm, such as the Kalman filter, extended Kalman filter, unscented Kalman filter, or particle filter. These algorithms determine the state of a system at a given time, given the history of known inputs and measured (noisy) outputs to the system for a given system model. The result is an estimate of the system state with some description of its uncertainty, e.g., a mean vector and covariance matrix, or a set of weighted samples.

In addition, one must also determine what the future input to the system will be, and its expected uncertainty. Often, this is done through the input parameters. Uncertainty in the input parameters is represented (e.g., by mean and covariance), and this indirectly defines the uncertainty in the future input trajectory.

3 Toolbox Contents

The toolbox provides two packages:

- `+Observers`: contains an `Observer` interface class, and subclasses implementing the Kalman filter, extended Kalman filter, unscented Kalman filter, and particle filter algorithms.

- `+Prognosis`: contains a `Predictor` class, implementing a sample-based, model-based prediction algorithm, and a `Prognoser` class, which implements a complete prognosis solution (an observer and a predictor).

The contents of the `Observers` package is as follows:

- `Observer.m`: an interface class for defining an observer. Classes for this interface must implement `initialize`, `estimate`, and `getStateEstimate` methods.
- `KalmanFilter.m`: a class implementing the Kalman filter algorithm, given a model as represented by state-space matrices, and process and sensor noise covariance matrices.
- `ExtendedFilter.m`: a class implementing the Kalman filter algorithm, given a model as represented by state and output equations, state and output Jacobian functions, and process and sensor noise covariance matrices.
- `UnscentedKalmanFilter.m`: a class implementing the unscented Kalman filter algorithm, given a model as represented by state and output equations, and process and sensor noise covariance matrices.
- `ParticleFilter.m`: a class implementing the particle filter algorithm, given a model as represented by state and output equations, and process and sensor noise variance vectors. It implements the sampling-importance-resampling type of particle filter, and assumes Gaussian noise distributions.
- `meanCovSampler.m` A function that generates samples from a distribution defined by a mean vector and covariance matrix.
- `computeSigmaPoints.m` A function that computes sigma points for a given mean vector, covariance matrix, and sigma point selection algorithm parameters.
- `wmean.m` A function computing the mean of a set of weighted samples.
- `wcov.m` A function computing the covariance of a set of weighted samples.

The contents of the `Prognosis` package is as follows:

- `Predictor`: a class implementing a sample-based prediction algorithm, given a `PrognosticsModel` object, and functions that sample the initial state, input parameters, and process noise. The number of samples and prediction horizon may be specified. It computes the probability of reaching the threshold in the horizon, the time to reach the horizon, and the system states and inputs at the time of reaching the threshold.

- `Prognoser`: a class implementing a model-based prognoser, given an `Observer`, a `PrognosticsModel`, and functions for sampling the input parameters and process noise.

In addition, several examples are provided:

- `testKalmanFilter.m`: A function demonstrating how to construct a Kalman filter object and estimate a model's state.
- `testUnscentedKalmanFilter.m`: A function demonstrating how to construct an unscented Kalman filter object and estimate a model's state, using the lithium-ion battery model as an example.
- `testParticleFilter.m`: A function demonstrating how to construct a particle filter object and estimate a model's state, using the lithium-ion battery model as an example.
- `testPredictor.m`: A function demonstrating how to construct a predictor object and set up sampling functions, using the lthium-ion battery model as an example.
- `testPrognoser.m`: A function demonstrating how to construct a prognoser object, using an unscented Kalman filter, and set up sampling functions, using the lthium-ion battery model as an example.

4 Example

As an example, a prognoser for the lithium-ion battery model is constructed, as shown below.

```
function testPrognoser
% testPrognoser   Test Prognoser class for Battery model
%
%   Copyright (c) 2016 United States Government as represented by
%   the Administrator of the National Aeronautics and Space
%   Administration. All Rights Reserved.

% Create battery model
battery = Battery.Create;

% Set variable input profile
loads = [8; 10*60; 4; 5*60; 12; 15*60; 5; 20*60; 10; 10*60];

% Set up noise covariance matrices
Q = diag(battery.V);
R = diag(battery.N);

% Create UKF
UKF = Observers.UnscentedKalmanFilter(@battery.stateEqn,...
    @battery.outputEqn,Q,R,'symmetric',3-8,1);

% Create sample generator for input equation parameters
% For each of the 10 load segments, sample from a uniform
% distribution with the mean given in the loads vector and
% the range [-1,+1] W for load and [-60,+60] s for the
% durations.
gains = ones(10,1);
gains(2:2:end) = 60;
inputParameterSampler = @(N) repmat(loads,1,N) + ...
    repmat(gains,1,N).*(rand(10,N)-0.5);

% Create Prognoser
horizon = 5000;
numSamples = 100;
prognoser = Prognosis.Prognoser('model',battery,'observer',UKF,...
    'horizon',horizon,'numSamples',numSamples,...
    'stateSampler',@Observers.meanCovSampler,...
    'inputParameterSampler',inputParameterSampler,...
    'processNoiseSampler',@battery.generateProcessNoise);

% Get initial state for battery simulation
t0 = 0;
[x0,u0,z0] = battery.getDefaultInitialization(t0,loads);

% Update/initialize prognoser based on initial data
prognoser.update(t0,u0,z0);

% Set up output data matrices
dt = 1;
T = t0:dt:3100;
X = zeros(length(x0),length(T));
Z = zeros(length(z0),length(T));
XEst = X;
ZEst = Z;
X(:,1) = x0;
Z(:,1) = z0;
XEst(:,1) = UKF.x;
ZEst(:,1) = UKF.z;
EODMean = [];
EODMax = [];
EODMin = [];
predictionTimes = [];

% Initialize simulation
x = x0;
u = u0;
z = z0;

% Simulate battery and run prognoser
for i=2:length(T)
    % Update state from T(i-1) to T(i)
    x = battery.stateEqn(T(i-1),x,u,...
                         battery.generateProcessNoise(),dt);
    % Get inputs for time T(i)
    u = battery.inputEqn(T(i),loads);
    % Compute outputs for time T(i)
    z = battery.outputEqn(T(i),x,u,battery.generateSensorNoise());

    % Update step for prognoser
    prognoser.update(T(i),u,z);

    % Predict once per minute
    if mod(i-1,60)==0
        prognoser.predict();

        % Print some status
        fprintf('Time: %g s\n',T(i));
        battery.printOutputs(z);
        fprintf('    EOD: %g s\n',mean(...
            prognoser.predictor.predictions.thresholdTimes));

        % Save some prediction data
        predictionTimes(end+1) = T(i);
        EODMean(end+1) = mean(...
            prognoser.predictor.predictions.thresholdTimes);
        EODMin(end+1) = min(...
            prognoser.predictor.predictions.thresholdTimes);
        EODMax(end+1) = max(...
            prognoser.predictor.predictions.thresholdTimes);
    end

    % Save data
    X(:,i) = x;
    Z(:,i) = z;
    XEst(:,i) = UKF.x;
    ZEst(:,i) = UKF.z;
end

% Plot output estimates
figure;
subplot(2,1,1);
plot(T,Z(1,:),'.',T,ZEst(1,:),'--');
title('Temperature Estimates');
xlabel('Time (s)')
ylabel('Temperature (deg C)');
subplot(2,1,2);
plot(T,Z(2,:),'.',T,ZEst(2,:),'--');
title('Voltage Estimates');
xlabel('Time (s)');
ylabel('Voltage (V)');
legend('Measured','Estimated');

% Compute actual end of discharge time, giving the exact
% loading parameters
battery.inputEqnHandle = @(P,t)Battery.InputEqn(P,t,loads);
T = battery.simulateToThreshold();
trueEOD = T(end);

% Plot prediction results
figure;
plot(predictionTimes,EODMean,'o',predictionTimes,EODMin,'o',...
    predictionTimes,EODMax,'o',predictionTimes,...
    trueEOD*ones(size(predictionTimes)),'o');
legend('Predicted EOD Mean','Predicted EOD Max',...
    'Predicted EOD Min','True EOD');
axis tight;
```

First, a battery model, from the `Prognostics Model Library`, is constructed.

A variable loading profile is then set. For the battery, the input parameters consist of pairs of load magnitude and durations, i.e., the battery will see first a load of 8 W for 10 minutes, followed by a load of 4 W for 5 minutes, etc. The process and sensor noise covariance matrices are then set, based on the default process and sensor noise variance vectors within the battery model object. The unscented Kalman filter is then created, using the battery model's state and output equations, the constructed covariance matrices, and parameters for the sigma points (symmetric unscented transform, with free parameter $\kappa = -5$, and no scaling ($\alpha = 1$)).

In order to construct the prognoser, one must define also the input parameter sampling function. Here, a function is constructed to draw from uniform distributions the magnitude and duration of each load segment. The distributions are centered on the true values (as defined in the `loads` variable), with a width of 2 W for the magnitude and 2 minutes for the duration. The input parameter sampler will generate different input parameter vectors for the given number of samples, using the built-in `rand` function (which generates samples from a uniform distribution between 0 and 1).

The prognoser is then constructed, with a prediction horizon of 5000 s, 100 samples for prediction, the battery model, the constructed unscented Kalman filter, the provided state sample generator (`Observers.meanCovSampler`), the constructed input parameter sampler, and the default process noise sampling function.

The battery simulation is then started, feeding the generated noisy simulated sensor data into the prognoser, calling the `update` method, which updates the internal state estimate (the data gets passed into the internal observer object). Once every 60 s, the `predict` method is called, and the prediction results extracted from the prognoser and saved.

The results are shown in Figs. 2 and 3. The former figure shows the comparison of measured sensor data and estimated outputs. Clearly, the unscented Kalman filter can track the battery state easily. The end-of-discharge predictions are shown in the latter figure. Because the battery state is accurately known, and the future input uncertainty is centered on the true future input trajectory, the mean predictions are very accurate. The minimum and maximum predicted values over the 100 samples are also shown. As the true end-of-discharge is approached, the resulting prediction uncertainty continues to decrease, as expected.

5 Notices

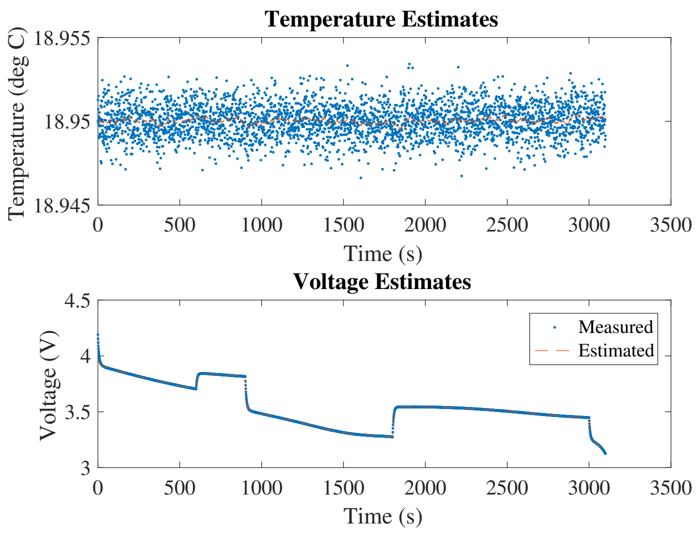

Figure 2: Battery output estimates for the unscented Kalman filter.

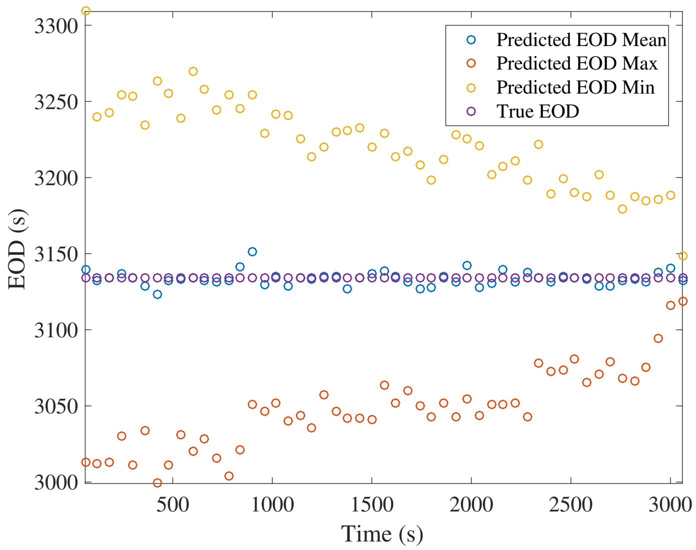

Figure 3: Battery end-of-discharge (EOD) predictions for a variable loading profile.

5.1 Disclaimers

No Warranty: THE SUBJECT SOFTWARE IS PROVIDED"AS IS" WITHOUT ANY WARRANTY OF ANY KIND, EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY