COMP 145 UNC-Chapel Hill
Maintenance Manual

The Kalman Filter On-line Learning Tool

Greg Welch, client

May 1, 2001

1 Document Change History

- March 8, 2001

  Annotated Outline

- March 25, 2001

  Matlab code (appendix)

- April 14-30, 2001

  Filled in annotated outline

Contents

1 Document Change History **1**

2 Introduction **3**
- 2.1 Statement of purpose . . . 3
- 2.2 Document conventions . . . 3
- 2.3 Other relevant documents . . . 3

3 System Overview **4**
- 3.1 Project modules . . . 4
- 3.2 High-level view of the kftool module . . . 5
- 3.3 Low-level view of the kftool module . . . 6

4 Procedures **7**
- 4.1 Getting the source . . . 7
- 4.2 Editing the source . . . 7
- 4.3 Building the project . . . 8
- 4.4 Testing the system . . . 8
- 4.5 Updating the documentation . . . 8

A Appendix: The User Manual **10**
- A.1 Quick Start . . . 10
- A.2 Descriptions of the parameter settings and control buttons . . . 10

B Appendix: Development System Requirements and Information **12**

C Appendix: Contact Information **13**

D Appendix: Matlab Code for the Kalman Filter **14**
- D.1 `dosim.m` . . . 14
- D.2 `gui.m` . . . 19
- D.3 `guiCallback.m` . . . 19
- D.4 `plotSim.m` . . . 20
- D.5 `truth.m` . . . 22

2 Introduction

2.1 Statement of purpose

The purpose of this document is to provide instructions and guidance for the client (or technical staff appointed by the client) to use when extending/modifying *The Kalman Filter On-line Learning Tool*. This manual does not attempt to anticipate every concern that may arise during the development of later versions of *The Kalman Filter On-line Learning Tool*. Instead, it gives guidelines, information and pointers that proved useful during the initial development of the tool.

The Kalman filter is a set of mathematical equations that provides an efficient computational estimate of the state of a process (*e.g.*, the position and orientation of an airplane) given a time-varying sequence of noisy measurements (*e.g.*, air speed, pressure, temperature, engine thrust). The filter is a popular mathematical estimator due to its efficiency and robustness. (For more information on the Kalman filter, see `http://www.cs.unc.edu/~welch/kalman/index.html`.) *The Kalman Filter On-line Learning Tool* is a web-based tool to help develop the intuition and insight of novice users regarding the behavior of the Kalman filter. Users have the ability to change various input parameters and then see how the Kalman filter responds for a given set of noisy measurements.

It is assumed that the reader of this document has a general understanding of how *The Kalman Filter On-line Learning Tool* works. In particular, it is strongly advised that the reader thoroughly review the User Manual (see Appendix A) and experiment with the tool before reading the rest of this document. It is also advised that the reader have experience with the Java™ programming language.

2.2 Document conventions

- Command line instructions and URLs appear in `this font`.
- CVS (see §4.1) project modules appear in this font.
- *The Kalman Filter On-line Learning Tool* always refers to the application to which this manual pertains.

2.3 Other relevant documents

- The client, Greg Welch, provides Matlab™ source code that executes the same simulation as *The Kalman Filter On-line Learning Tool* (see Appendix D).
- Detailed documentation of the project packages, classes and interfaces is provided in "Java™ API Documentation for *The Kalman Filter On-line Learning Tool*". The Implementation Manual is also helpful for navigating the source code. Both are referenced in the remainder of this document and are included in *The Kalman Filter On-line Learning Tool* project package.
- The "Dynamic and Measurement Models" document, written by the client Greg Welch, provides a description of the simulation performed by *The Kalman Filter On-line Learning Tool*. It is referenced in the remainder of this document and is included in *The Kalman Filter On-line Learning Tool* project package.

3 System Overview

3.1 Project modules

All project files are stored in a CVS repository (see §4.1), and belong to one of the following project modules.

kftool. This module contains the source code for `The Kalman Filter On-line Learning Tool` application and applet. Sections 3.2 and 3.3 are devoted to the discussion of this module. (Note: The application version of `The Kalman Filter On-line Learning Tool` includes an option for saving tab-delimited simulation data to file. The applet version does not include this option because Java™'s security features forbid applets from accessing the local file system. Otherwise, the two versions of the tool are the same.)

ptplot3.1p1. This module contains source code for the plot software used in this project, and is included for archival purposes.

Jama. The module contains source code for the Java™ matrix package used in this project, and is included for archival purposes.

matlab_code. This module contains source code for a Matlab™ implementation of the Kalman filter simulation (see Appendix D).

manuals. This module contains files for the Maintenance Manual, Implementation Manual and "Java™ API Documentation for `The Kalman Filter On-line Learning Tool`".

latex_doclet. This module contains software for generating the "Java™ API Documentation for `The Kalman Filter On-line Learning Tool`" document, a Java™ doclet. The software reads Javadoc™ comments embedded in source code and conveniently produces a LaTeX document, which displays the information in an organized manner. Refer to Section 4.5 for instructions on creating/updating Javadoc™ comments and generating the Java™ doclet.

test_data. This module contains simulation data generated by the application version of `The Kalman Filter On-line Learning Tool` for every permutation of the input parameters. These data files may be used to test the correctness of the tool (in the case that it has been modified). Refer to Section 4.4 for testing guidelines.

jlfgr. This module contains a collection of toolbar button graphics. The graphics have been designed specifically for use with the Java™ look and feel. They conform to the Java™ look and feel Design Guidelines. The step window's button icons use look and feel graphics. (For more information see `http://java.sun.com/products/jlf/dg/index.htm`.)

web. This module contains the files necessary for running `The Kalman Filter On-line Learning Tool` applet. The files should be copied to any space desiring to host the applet. For more information, see the `README` file in the web module.

3.2 High-level view of the kftool module

The kftool module is largest and most important module of the project. It contains source code for *The Kalman Filter On-line Learning Tool* application and applet. The module is divided into six parts (see Figure 1), each performing a distinct function.

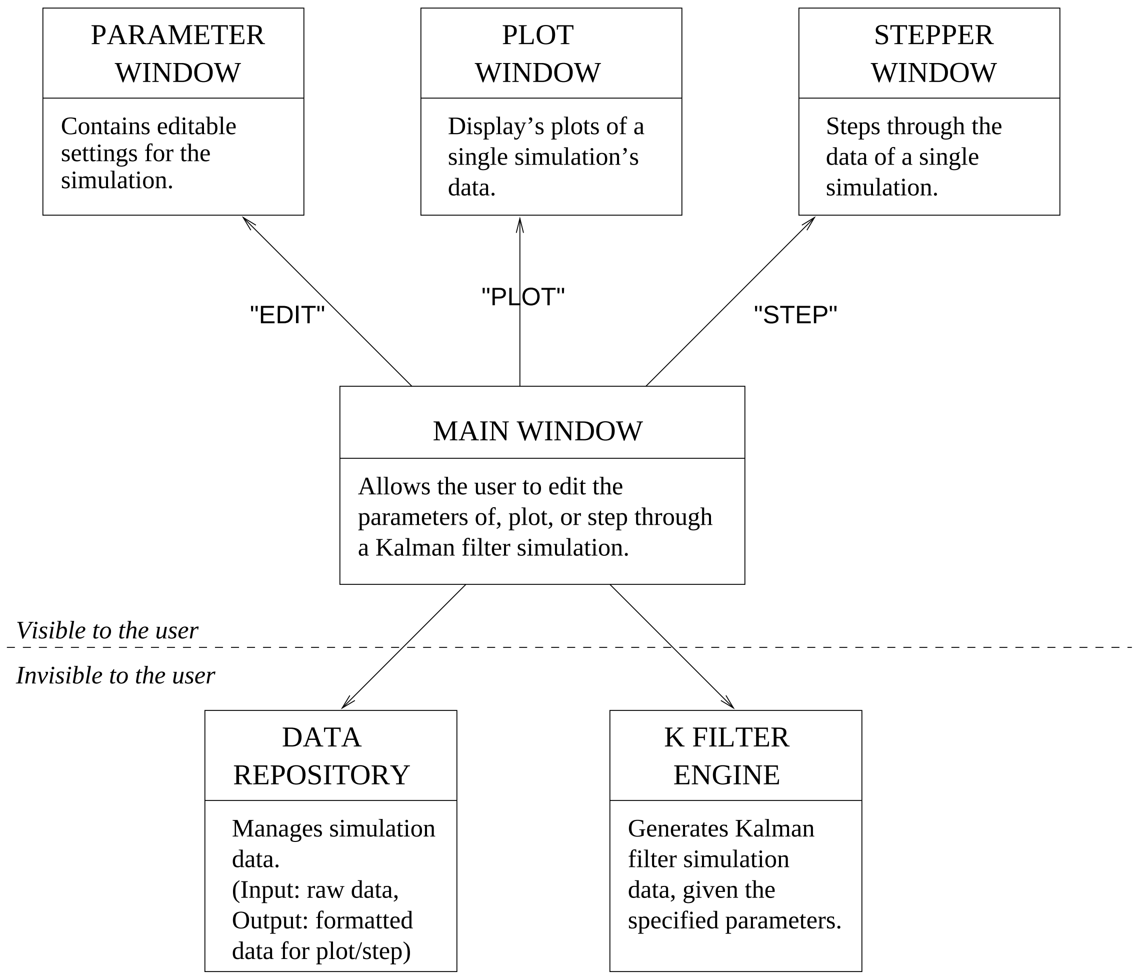
PARAMETER WINDOW
Contains editable settings for the simulation.
PLOT WINDOW
Display's plots of a single simulation's data.
STEPPER WINDOW
Steps through the data of a single simulation.
"EDIT"
"PLOT"
"STEP"
MAIN WINDOW
Allows the user to edit the parameters of, plot, or step through a Kalman filter simulation.
Visible to the user
Invisible to the user
DATA REPOSITORY
Manages simulation data. (Input: raw data, Output: formatted data for plot/step)
K FILTER ENGINE
Generates Kalman filter simulation data, given the specified parameters.

Figure 1: High-level view of the kftool project module.

Main window. Primarily, the main window serves to provide the user with choices that govern the execution of *The Kalman Filter On-line Learning Tool*. The user can choose to “Edit”, “Plot”, or “Step” through a Kalman filter simulation. The “Edit” selection invokes the parameter window, the “Plot” selection invokes the plot window, and the “Step” selection invokes the stepper window. The user is also given a “Help” option, which displays web pages with instructions and useful information.

Secondarily, the main window acts as the main controller for the system. It passes parameters from the parameter window to the Kalman filter engine and collects simulation data for passing to the data repository. The main window also passes formatted simulation data from the data repository to the plot window and the step window. In the application version of the tool, the main window also allows the user to save (tab-delimited) simulation data to file, receiving such data from the data repository.

Parameter window. The parameter window gives the user an opportunity to modify the default parameters of the Kalman filter simulation. The user may select among several options for the actual dynamics, the modeled dynamics, the measurement type and the measurement noise. (For more information about the purpose of these parameters, see the "Dynamic and Measurement Models" document.)

Plot window. The plot window displays three plots of simulation data to the user. Plot 1 displays the values of truth and estimate for each time-step. Plot 2 displays the covariance values for each time-step. Plot 3 displays the residual values for each time-step.

Step window. The step window allows the user to step through the actual simulation data for a given time-step.

Kalman filter engine. The Kalman filter engine performs the simulation according to the parameters passed from the main window (which are either the default parameters or the parameters specified by the user in the parameter window). The Kalman filter engine generates raw simulation data and passes it to the main window, which then passes it to the data repository.

Data repository. The data repository receives raw simulation data from the main window, and formats this data in the following three ways.

1. Arrays of size # of time-steps (target: plot window).
2. Single elements corresponding to a time-step (target: step window).
3. String of tab-delimited values for each time-step (target: file)

3.3 Low-level view of the kftool module

For a more detailed look at the kftool project module, refer to "Java^{TM} API Documentation for `The Kalman Filter On-line Learning Tool`" and the Implementation Manual.

4 Procedures

4.1 Getting the source

`The Kalman Filter On-line Learning Tool` source code is in a CVS (Concurrent Versions System, v1.11) repository. CVS is a version control system that allows old versions of program files to be systematically stored with corresponding explanations of when, why, and by whom the files were modified. (For more information on CVS, see `http://www.cvshome.org/docs/manual/cvs.html`.)

To get a copy of the source code for any module(s),

1. Set the `CVSROOT` environment variable to the designated project space (*i.e.*, `setenv CVSROOT /afs/cs.unc.edu/project/tracker/kftutor`).
2. Go to your local workspace where you desire to put the source code (*e.g.*, `cd /afs/cs.unc.edu/home/parker/kfcode`).
3. Checkout the module(s) (*e.g.*, to checkout the kftool module, `cvs checkout kftool`).

Once you have checked out source code to your local workspace, it is easy to bring it up-to-date with the files currently in the repository by doing the following,

1. Go to your local workspace that you desire to update (*e.g.*, `cd /afs/cs.unc.edu/home/parker/kfcode`).
2. Update the module(s) (*e.g.*, `cvs update kftool`).

4.2 Editing the source

To modify an existing source code file for any module(s),

1. Checkout the source code (see §4.1).
2. Modify the file(s) in your local workspace as desired.
3. Rebuild the project (see §4.3).
4. Checkin the modified file(s) (*i.e.*, `cvs commit` *`filename`*).

To create a new source code file for any module(s),

1. Checkout the source code (see §4.1).
2. Create the file(s) in your local workspace as desired.
3. Rebuild the project, modifying the build procedures as appropriate (see §4.3).
4. Add and checkin the new file(s) (*i.e.*,
```
cvs add filename
cvs commit filename
```
).

**Note: CVS allows you to provide a *message* anytime you add or commit a file. The message is simply a comment added to the change log, in which you can give some reason for modifying the code. Using this feature is recommended to get the most out of version control. To learn more about *messaging*, type `man cvs` or see `http://www.cvshome.org/docs/manual/cvs.html`.

4.3 Building the project

The Makefile in the kftool module is provided to facilitate the process of building an executable for *`The Kalman Filter On-line Learning Tool`*. To use the Makefile do the following,

1. Read and follow the directions in `kf_tool/Makefile`.
2. Build the desired target
(*e.g.*, `make all`).

4.4 Testing the system

Whenever *`The Kalman Filter On-line Learning Tool`* source code is modified, testing its output for correctness is recommended. The following two methods of testing are suggested.

- The test_data module contains data files for every permutation of the input parameters to the Kalman filter simulation. The tab-delimited data output saved to file by the application version of *`The Kalman Filter On-line Learning Tool`* should be identical to a data file in the test_data module (according to the simulation parameters used as input).

  The random number generator used will produce identical results for a given set of input parameters, so long as the seed stays the same. (For more information, see the `java.util.Random` class.) However, if the random number generator and/or its seed are modified, comparison against these data files will not be a good test. In fact, if the random number generator is modified, replacing the data files of the test_data module with new data files (believed to be correct for the modified random number generator) is recommended to facilitate testing of subsequent versions of *`The Kalman Filter On-line Learning Tool`*.

- Comparison with output of the client's Matlab™ code (see Appendix D) is also a good way to test the accuracy of the simulation data and plots. Note that output the Matlab™ implementation of the Kalman filter simulation will differ slightly from the Java™ implementation (since a different random number generator is used, and the source code is not identical). However, the Matlab™ output of data and plots should certainly be similar to that of *`The Kalman Filter On-line Learning Tool`*.

4.5 Updating the documentation

Updates to the documentation are recommended, as *`The Kalman Filter On-line Learning Tool`* source code is modified. Of course, these procedures are optional and have no effect on the functionality of the tool.

The Implementation Manual and Maintenance Manual should be updated as necessary (*i.e.*, when significant changes are made to the source code). Both documents are written in LaTeX, and the `.tex` files are located in the manuals module. (For more information on LaTeX, type `man latex`.)

The "Java™ API Documentation for `The Kalman Filter On-line Learning Tool`" document should be regenerated whenever changes to the source code affect its documentation (*e.g.*, deletion/addition of project classes, deletion/addition of class methods, deletion/addition of method parameters, *etc.*). The software for generating the Java™ doclet is located in the latex_doclet module. The following two steps are suggestions for updating the "Java™ API Documentation for `The Kalman Filter On-line Learning Tool`" document.

1. Add/delete/edit Javadoc™ comments in the source code. The general form of a Javadoc™ comment is the following.

   ```
   /**
    * This is the description part of a doc comment
    *
    * @tag    Comment for the tag
    */
   ```

   (For more information on how to write Javadoc™ comments,
   see `http://java.sun.com/j2se/javadoc/writingdoccomments`.)

2. Generate the Java™ doclet using the following suggested steps.

   (a) Go to your local workspace, where you have previously checked out the source code
   (*e.g.*, `cd /afs/cs.unc.edu/home/parker/kfcode`).

   (b) Make certain the latex_doclet module has been checked out
   (*e.g.*, `cvs checkout latex_doclet`).

   (c) Go to the kftool module
   (*i.e.*, `cd kftool`).

   (d) Use the Makefile to execute the Java™ doclet software
   (*i.e.*, `make latex_docs`).

   (e) A file `doclet.tex` will be written to the `latex_doc` subdirectory of `kftool`, so go to that subdirectory
   (*i.e.*, `cd latex_doc`).

   (f) Send `doclet.tex` to the LaTeX interpreter to produce the postscript document, `doclet.ps`
   (*i.e.*, `latex doclet.tex`
   `dvips -o doclet.ps doclet`).

A Appendix: The User Manual

The *User Manual* is provided to give the reader some context for how the `The Kalman Filter On-line Learning Tool` is to be used, as well as, to document a set of instructions (should they ever be needed).

A.1 Quick Start

1. Go to `http://www.cs.unc.edu/~parker/comp145/kalman.html`.
2. Click on *Use* `The Kalman Filter On-line Learning Tool`.
3. Leave the controls in their default settings.
4. Click on the *Plot* button.
5. View the plots.
6. Close the plot window.

A.2 Descriptions of the parameter settings and control buttons

Actual Dynamics The actual dynamics determine how the water level will actually behave in the simulation. The default is that the water will remain at a constant level. The other options include filling and sloshing. When filling is selected the actual level of the water tank will steadily increase over the time interval during which the simulation is run. When sloshing is selected the actual level of the water tank will have a sinusoidal increase and decrease (*i.e.*, like waves in water). Note that these two options, filling and sloshing, are orthogonal to each other in the sense that it is possible to select (i) only filling, (ii) only sloshing, (iii) both filling and sloshing, or (iv) neither filling nor sloshing (the default).

Modeled Dynamics The modeled dynamics setting determines the analytic function that the Kalman filter will use to model the level of water in the tank. In other words, the Kalman filter "thinks" that the actual behavior of the water is described by the estimate setting. Naturally, the Kalman filter will give the best estimate of the state of the water level when the modeling function matches the truth. It is intended however, that the user experiment with the behavior of the filter when the modeled function does *not* match that of the truth. In such case, note the relationship between the residual plot and discrepancy between the truth and model functions.

Measurement Type The measurement type feature allows the user to select whether the device used to measure the water level in the tank is (i) the height of a float on top of the water, or (ii) the angle of a mechanical arm whose end sits on top of the water. Mathematically this difference is significant because in the linear float case, the state of the process (the actual level of the water) is a linear function of the measurements received. In the angular case however, the state of the process is a more sophisticated sine function with respect to the measurements received. The Kalman filter equations must take this difference into account, and are more complex in the angular case. It is recommended that the user thoroughly understand the linear case before running simulations in the angular case.

Measurement Noise The measurement noise determines the accuracy of the measurements to which the filter has access. Real systems using the Kalman filter will have different qualities of measurement as determined by the precision of the measurement instruments available and features of the environment in which the system operates. The `The Kalman Filter On-line Learning Tool` allows the user to simulate these differences by inserting random noise in levels commensurate with the simulated environment/system.

Edit The edit button in the main window allows the user to select the desired actual dynamics, modeled dynamics, measurement type, and measurement noise.

Plot The plot button performs the simulation for a fixed amount of time. At each instant the actual level of the water is determined based on the user's choice of actual dynamics. Noisy measurements are then available to the filter based on the measurement noise and measurement type selected by the user. The Kalman filter then uses the modeling function selected by the user as a part of the Kalman filter equations to determine the filter's estimate of the state of the water. These quantities are then plotted in a separate window. If some part of the plot is of particular interest, the mouse may be used to draw a rectangle around that region, causing it to be shown in more detail.

Help The help button allows the user to see the help features built into `The Kalman Filter On-line Learning Tool`.

Step After developing a general feel for how the Kalman filter behaves with different combinations of truth and estimate functions, the stepping mode is designed to allow users to study exactly how this behavior is achieved via the actual Kalman filter equations. The step function shows, for a selected time step, the decimal values that determine the true state, predicted state, corrected state, actual measurement, predicted measurement, Kalman gain, predicted covariance, and corrected covariance. These values are displayed for two adjacent time steps, so that the user can see how the equations compute these quantities at points of interest in the simulation. The buttons within the step window allow different time instances to be selected at which the behavior of the filter is most interesting.

B Appendix: Development System Requirements and Information

The following are requirements for extensions/modifications to the project.

- Java™ Development Kit: JDK 1.3.0 for Linux™ (or Sun™) from Sun™
  (includes Java™ 2 RTE and Java Hotspot™ Client VM)
- Java™ Plug-in 1.3 allows a browser to view web pages using Java™ 2 RTE:
  (available at `http://java.sun.com/products/plugin`,
  for more information see `http://java.sun.com/products/plugin/1.3/overview.html`)

The following is information about tools used in the initial project development, provided in case they can be useful for further development of the project.

- Integrated Development Environment: Forte™ for Java™ 2.0 from Sun™
  (available at `http://www.sun.com/forte/ffj`)
- Plotting Tool: Ptplot 3.1
  (available at `http://ptolemy.eecs.berkeley.edu/java/ptplot3.1/ptolemy/plot/doc`)
- Java Matrix Package: Jama 1.0.1
  (available at `http://math.nist.gov/javanumerics/jama`)
- Version Control Software: CVS 1.10.7
  (for more information see `http://www.cvshome.org/docs/manual/cvs.html` or type `man cvs`)
- LaTeX2$_E$ Generating Java™ doclet
  (available at `http://www.c2-tech.com/java/TexDoclet`)

This optional information proved to be helpful during the initial project development.

- Java™ 2 API Specification
  (available at `http://java.sun.com/products/jdk/1.2/docs/api/index.html`)
- Java™ Look and Feel Design Guidelines
  (available at `http://java.sun.com/products/jlf/dg/index.htm`)

C Appendix: Contact Information

The following students completed `The Kalman Filter On-line Learning Tool` as a course project for COMP145 at the University of North Carolina at Chapel Hill.

- **TEAM 18**

  DIRECTOR Christopher Riley, `cjriley@cs.unc.edu`

  PRODUCER Thomas Bodenheimer, `bodenhei@cs.unc.edu`

  ADMINISTRATIVE LEADER Erin Parker, `parker@cs.unc.edu`

  LIBRARIAN John Carpenter, `carpente@cs.unc.edu`

The client for the project was Greg Welch, a professor in the Department of Computer Science at UNC-CH, `welch@cs.unc.edu`.

D Appendix: Matlab Code for the Kalman Filter

The following is Matlab™ source code (written by the client, Greg Welch) for the Kalman filter simulation performed by *The Kalman Filter On-line Learning Tool*. The code may be checked out of the CVS repository (see §4.1), as module matlab_code. To run the simulation, execute `gui.m` in Matlab™. (**Note that the Matlab™ source code is not a direct translation of the Java™ source code of *The Kalman Filter On-line Learning Tool*, but both perform the same Kalman filter simulation.)

D.1 dosim.m

```
function S = dosim(T, cda, cdm, cm, sr_scale, sq_scale, stq_C, stq_F, stq_S)
%
% Run sim of level sensors
%
% Input:
%  T: truth struct with fields...
%     L: actual levels struct w/ fields
%         L.C (constant)
%         L.F (filling)
%         L.S (sloshing)
%         L.FS (filling and sloshing)
%         L.S0 (slosh around 0)
%     m: measured values struct
%         level/linear fields m.l.C, m.l.F, m.l.S, m.l.FS
%         angle/nonlinear fields m.a.C, m.a.F, m.a.S, m.a.FS
%    ts: time steps [second]
%    ...: a bunch of other parameters that need to come from "truth"
%  cda: char specifying ACTUAL dynamics ('C','F','S','FS')
%  cdm: char specifying MODELED dynamics ('C','F','S','FS')
%  cm: char specifying the measurement type ('L', 'A')
%  sr_scale: meas noise factor (to play) -- {0.01, 0.1, 1, 10, 100}
%  sq_scale: process noise factor (to play) -- {0.01, 0.1, 1, 10, 100}
%

% misc globals
global d2r r2d;
d2r = pi/180.0; % constant to convert [degree] to [radian]
r2d = 180.0/pi; % constant to convert [radian] to [degree]

% temporal parameters
stime = T.stime; % length of sim [second]
mrate = T.mrate; % measurement rate [1/second]
dt = 1/mrate;

% water level limits
L_min = T.L_min; % where start filling, etc. [meter]
L_max = T.L_max; % full [meter]

% Measurement device parameters
global db df ka kl;
db = T.db; % Base for angular sensor, just above the max water [meter]
df = T.df; % Angular sensor arm w/ float, long enough to hit bottom on empty [meter]
ka = T.ka; % Angular/non-linear float scale constant
kl = T.kl; % Level/linear float parameters scale constant

% Sloshing (sinusoidal) parameters
sf = T.sf; % slosh/sin frequency [1/second]
sp = T.sp; % slosh phase [degree]

% measurement noise magnitudes
srl_m = sr_scale*T.srl_m; % stdev of level/linear sensor noise [meter]
sra_d = sr_scale*T.sra_d; % stdev of angule/non-linear sensor noise [degree]
```

```
%
% Set up filter parameters
%
% Four possibilities
% 1. Constant level (C)
% 2. Filling (F)
% 3. Sloshing (S)
% 4. Filling + Sloshing (FS)
%

if (nargin == 6) % normal situation
  stq_C(1) = 5.9609e-05; % Tuned Constant w/ linear meas
  stq_C(2) = 3.5936e-05; % Tuned Constant w/ angular meas
  sq_C = mean(stq_C); % avg of the two

  stq_F(1) = 1.9638; % Tuned Filling w/ linear meas
  stq_F(2) = 2.027; % Tuned Filling w/ angular meas
  sq_F = mean(stq_F); % avg of the two

  stq_S(1) = 5.9609e-05; % Tuned Sloshing w/ linear meas
  stq_S(2) = 5.9609e-05; % Tuned Sloshing w/ angular meas
  sq_S = mean(stq_S); % avg of the two

  % special tune case for filling and sloshing
  if (cdm == 'FS')
    stq_F(1) = 0.0028009; % Tuned Filling w/ linear meas
    stq_F(2) = 0.0021271; % Tuned Filling w/ angular meas
    sq_F = mean(stq_F); % avg of the two

    stq_S(1) = 0.0014462; % Tuned Sloshing w/ linear meas
    stq_S(2) = 0.0010254; % Tuned Sloshing w/ angular meas
    sq_S = mean(stq_S); % avg of the two
  end
elseif (nargin == 9) % tuning the filters
  sq_C = stq_C;
  sq_F = stq_F;
  sq_S = stq_S;
else
  error('Incorrect number of input arguments for dosim.m.')
end

% Measurement model
if (cm == 'L')
  mtype = 1;
elseif (cm == 'A')
  mtype = 2;
else
  error('Unkown measurement model.');
end

% Set up the dynamic model parameters based on the input selector cdm
switch cdm
  case 'C' % Constant level
    % State dimension
    sd = 1;

    % Dynamic/process model
    % see Section 1.2.1 in document "models"
    A = zeros(1,1,T.nmeas);
    A(1,1,:) = 1;
    qc = (sq_scale*sq_C)^2; % scale qc then square for variance
    Q = zeros(1,1,T.nmeas);
    Q(1,1,:) = qc*dt;

    % Set aside space for results
    S.Xp = zeros(sd,T.nmeas); % predicted state (X)
    S.Xc = zeros(sd,T.nmeas); % corrected
    S.Pp = zeros(sd,sd,T.nmeas); % predicted covariance (P)
    S.Pc = zeros(sd,sd,T.nmeas); % corrected
    S.Zp = zeros(1,T.nmeas); % predicted measurement (Z)
    S.K  = zeros(sd,T.nmeas); % Kalman gain

    % Initialize filter
    L_init = L_max/2.0; % initial guess
    S.Xp(1,1) = L_init;
    S.Xc(1,1) = L_init;
    S.Pp(1:1,1:1,1) = L_max^2;
    S.Pc(1:1,1:1,1) = L_max^2;

  case 'F' % Filling
    % State dimension
    sd = 2;

    % Dynamic/process model
    % see Section 1.2.2 and equations (4) and (5) in document "models"
    A = zeros(sd,sd,T.nmeas);
    for m=1:T.nmeas
      A(1,1,m) = 1;
      A(1,2,m) = dt;
    end
    Q = zeros(sd,sd,T.nmeas);
    qc = (sq_scale*sq_F)^2; % scale qc then square for variance
    for m=1:T.nmeas
      Q(1,1,m) = qc*dt^3/3.0;
      Q(1,2,m) = qc*dt^2/2.0;
      Q(2,1,m) = Q(1,2,m);
      Q(2,2,m) = qc*dt;
    end

    % Set aside space for results
    S.Xp = zeros(sd,T.nmeas); % predicted state (X)
    S.Xc = zeros(sd,T.nmeas); % corrected
    S.Pp = zeros(sd,sd,T.nmeas); % predicted covariance (P)
    S.Pc = zeros(sd,sd,T.nmeas); % corrected
    S.Zp = zeros(1,T.nmeas); % predicted measurement (Z)
    S.K  = zeros(sd,T.nmeas); % Kalman gain

    % Initialize filter
    L_init = L_max/2.0; % initial guess
    S.Xp(1,1) = L_init;
    S.Xc(1,1) = L_init;
    S.Pp(:,:,1) = [L_max^2,0;0,(L_max/stime)^2];
    S.Pc(:,:,1) = [L_max^2,0;0,(L_max/stime)^2];

  case 'S' % Sloshing
    % State dimension
    sd = 2;

    % Dynamic/process model
    % see Section 1.2.3 and equations (6) and (7) in document "models"
    A = zeros(sd,sd,T.nmeas);
    w  = 2*pi*sf; % omega
    for m=1:T.nmeas
      A(1,1,m) = 1;
      A(2,2,m) = 1;
      A(1,2,m) = w*cos(w*T.ts(m)+sp*d2r);
    end
    Q  = zeros(sd,sd,T.nmeas);
    qc = (sq_scale*sq_S)^2; % scale qc then square for variance
    for m=1:T.nmeas
      Q(1,1,m) = qc*w^2*cos(w*T.ts(m))^2*dt*(3*T.ts(m)^2+3*T.ts(m)*dt+dt^2)/3.0;
      Q(1,2,m) = w*cos(w*T.ts(m))*qc*dt*(2*T.ts(m) + dt)/2.0;
      Q(2,1,m) = Q(1,2,m);
      Q(2,2,m) = qc*dt;
    end

    % Set aside space for results
    S.Xp = zeros(sd,T.nmeas); % predicted state (X)
    S.Xc = zeros(sd,T.nmeas); % corrected
    S.Pp = zeros(sd,sd,T.nmeas); % predicted covariance (P)
    S.Pc = zeros(sd,sd,T.nmeas); % corrected
    S.Zp = zeros(1,T.nmeas); % predicted measurement (Z)
    S.K  = zeros(sd,T.nmeas); % Kalman gain

    % Initialize filter
    L_init = L_max/2.0; % initial guess
    S.Xp(1,1) = L_init;
    S.Xc(1,1) = L_init;
    S.Xp(2,1) = T.sm; % temp - cheat
    S.Xc(2,1) = T.sm;
    S.Pp(:,:,1) = [L_max^2,0;0,(L_max/2)^2];
    S.Pc(:,:,1) = [L_max^2,0;0,(L_max/2)^2];

  case 'FS' % Filling and Sloshing
    % State dimension
    sd = 3;

    % Dynamic/process model
    % see Section 1.2.4 and equations (8) and (9) in document "models"
    A = zeros(sd,sd,T.nmeas);
    w  = 2*pi*sf; % omega
    for m=1:T.nmeas
      A(1,1,m) = 1;
      A(2,2,m) = 1;
      A(3,3,m) = 1;
      A(1,2,m) = dt;
      A(1,2,m) = w*cos(w*T.ts(m)+sp*d2r);
    end
    Q  = zeros(sd,sd,T.nmeas);
    qcs = (sq_scale*sq_S)^2; % scale qc then square for variance
    qcf = (sq_scale*sq_F)^2; % scale qc then square for variance
    for m=1:T.nmeas
      Q(1,1,m) = dt*(dt^2+3*T.ts(m)^2+3*T.ts(m)*dt)*
                 (qcf+w^2*cos(w*T.ts(m))^2*qcs)/3.0;
      Q(1,2,m) = qcf*dt*(2*T.ts(m) + dt)/2.0;
      Q(2,1,m) = Q(1,2,m);
      Q(1,3,m) = w*cos(w*T.ts(m))*qcs*dt*(2*T.ts(m)+dt)/2.0;
      Q(3,1,m) = Q(1,3,m);
      Q(2,2,m) = qcf*dt;
      Q(3,3,m) = qcs*dt;
    end

    % Set aside space for results
    S.Xp = zeros(sd,T.nmeas); % predicted state (X)
    S.Xc = zeros(sd,T.nmeas); % corrected
    S.Pp = zeros(sd,sd,T.nmeas); % predicted covariance (P)
    S.Pc = zeros(sd,sd,T.nmeas); % corrected
    S.Zp = zeros(1,T.nmeas); % predicted measurement (Z)
    S.K  = zeros(sd,T.nmeas); % Kalman gain

    % Initialize filter
    L_init = L_max/2.0; % initial guess
    S.Xp(1,1) = L_init;
    S.Xc(1,1) = L_init;
    S.Xp(3,1) = T.sm; % temp - cheat
    S.Xc(3,1) = T.sm;
    S.Pp(:,:,1) = [L_max^2,0,0; 0,(L_max/stime)^2,0; 0,0,(L_max/2)^2];
    S.Pc(:,:,1) = [L_max^2,0,0; 0,(L_max/stime)^2,0; 0,0,(L_max/2)^2];

  otherwise
```

```
    error('Unknown dynamic model type.');
end

% Choose the set of measurements based on the input selector cda
% (actual dynamics)
switch cda
  case 'C' % Constant level
    % Measurement model
    if (mtype == 1)
      Za = T.m.l.C; % actual measurements
    else
      Za = T.m.a.C; % actual measurements
    end
  case 'F' % Filling level
    % Measurement model
    if (mtype == 1)
      Za = T.m.l.F; % actual measurements
    else
      Za = T.m.a.F; % actual measurements
    end
  case 'S' % Sloshing level
    % Measurement model
    if (mtype == 1)
      Za = T.m.l.S; % actual measurements
    else
      Za = T.m.a.S; % actual measurements
    end
  case 'FS' % Filling and Sloshing level
    % Measurement model
    if (mtype == 1)
      Za = T.m.l.FS; % actual measurements
    else
      Za = T.m.a.FS; % actual measurements
    end
  otherwise
    error('Unknown dynamic model type.');
end

% Measurement noise model
if (mtype == 1)
  % see Section 2.1 in document "models"
  R = (srl_m * kl)^2;
else
  % see Section 2.2 in document "models"
  R = (sra_d * ka)^2;
end



% Loop through all measurements
for k = 2:T.nmeas

  % predict
  S.Xp(:,k)   = A(:,:,k)*S.Xc(:,k-1);
  S.Pp(:,:,k) = A(:,:,k)*S.Pc(:,:,k-1)*A(:,:,k)' + Q(:,:,k);

  % measurement prediction and Jacobian
  S.Zp(k) = meas(S.Xp(:,k),mtype,sd);
  H       = measJacobian(S.Xp(:,k),mtype,sd);

  % correct
  S.K(:,k)    = S.Pp(:,:,k)*H'*inv(H*S.Pp(:,:,k)*H' + R); % Kalman gain
  S.Xc(:,k)   = S.Xp(:,k) + S.K(:,k)*(Za(k)-S.Zp(k));
  S.Pc(:,:,k) = (eye(sd) - S.K(:,k)*H)*S.Pp(:,:,k);

end
```

```
return



% local subroutine for measurement model
function Zp = meas(Xp,mtype,sd)
  global db df ka r2d;

  if (mtype == 1)
    % Linear measurement model
    % see Section 2.1 in document "models"
    H = measJacobian(Xp,mtype,sd);
    Zp = H*Xp;
  else
    % Angular measurement model
    % see Section 2.2 and equation (13) in document "models"
    as = (db-Xp(1))/df;
    if (as > 1)
      as = 1;
    elseif (as < -1)
      as = -1;
    end
    Zp = r2d*ka*asin(as);
  end
return

% local subroutine for measurement model Jacobian
function H = measJacobian(Xp,mtype,sd)
  global db df ka kl r2d;


  if (mtype == 1)
    % Linear measurement model
    % see Section 2.1 in document "models"
    H = zeros(1,sd);
    H(1,1) = kl;
  else
    % Angular measurement model
    % see Section 2.2 and equation (14) in document "models"
    H = zeros(1,sd);
    as2 = ((db-Xp(1))/df)ˆ2;
    H(1,1) = -r2d*ka/(df*sqrt(abs(1-as2)));
  end
return
```

D.2 gui.m

```
da = uicontrol('Style','Popup','String',...
       'Constant|Fill|Slosh|Fill & Slosh','Position',[20 320 100 50]);

dm = uicontrol('Style','Popup','String',...
       'Constant|Fill|Slosh|Fill & Slosh','Position',[120 320 100 50]);

mm = uicontrol('Style','Popup','String',...
       'Linear|Angular','Position',[220 320 100 50]);

go = uicontrol('Style','Pushbutton','Position',...
    [20 150 100 70], 'Callback','guiCallback','String','Go');
```

D.3 guiCallback.m

```
vda = get(da,'Value');
vdm = get(dm,'Value');
vmm = get(mm,'Value');
```

```
switch vda
  case 1
    cda = ’C’;
  case 2
    cda = ’F’;
  case 3
    cda = ’S’;
  case 4
    cda = ’FS’;
  otherwise
    error(’Unknown actual dynamic model type.’);
end

switch vdm
  case 1
    cdm = ’C’;
  case 2
    cdm = ’F’;
  case 3
    cdm = ’S’;
  case 4
    cdm = ’FS’;
  otherwise
    error(’Unknown measured dynamic model type.’);
end

switch vmm
  case 1
    cm = ’L’;
  case 2
    cm = ’A’;
  otherwise
    error(’Unknown measurement model type.’);
end

meas_scale = 100;
process_scale = 1;
T = truth(0);
S = dosim(T,cda,cdm,cm,meas_scale,process_scale);
plotSim(T,S,cda,cdm,cm);
```

D.4 plotSim.m

```
function plotSim(T,S,cda,cdm,cm)

scrsz = get(0,’ScreenSize’);
h = figure(2)
set(h,’Position’,[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)]);

% Level
pane = 1;
h = subplot(3,1,pane);
switch cda
  case ’C’
    plot(T.ts,S.Xc(1,:),T.ts,T.L.C)
  case ’F’
    plot(T.ts,S.Xc(1,:),T.ts,T.L.F)
  case ’S’
    plot(T.ts,S.Xc(1,:),T.ts,T.L.S)
  case ’FS’
    plot(T.ts,S.Xc(1,:),T.ts,T.L.FS)
  otherwise
    error(’Unknown dynamic model type.’);
end
min_x = 0;
max_x = T.stime;
```

```
min_y = 0;
max_y = T.L_max;
axis([min_x max_x min_y max_y]);
xlabel('t [second]');
ylabel('L [meter]');
title(sprintf('Actual (%s) vs. Estimated (%s)',cda,cdm));

% Covariance
pane = 2;
h = subplot(3,1,pane);
semilogy(T.ts,sqrt(squeeze(S.Pc(1,1,:))));
min_x = 0;
max_x = T.stime;
min_y = 0;
max_y = T.L_max;
axis([min_x max_x min_y max_y]);
xlabel('t [second]');
ylabel('sqrt(P) [meter]');
title(sprintf('Process deviation',cda,cdm));

% Measurement residuals
pane = 3;
h = subplot(3,1,pane);
switch cda
  case 'C'
    if (cm == 'L')
      plot(T.ts,T.m.l.C-S.Zp)
    else
      plot(T.ts,T.m.a.C-S.Zp)
    end
  case 'F'
    if (cm == 'L')
      plot(T.ts,T.m.l.F-S.Zp)
    else
      plot(T.ts,T.m.a.F-S.Zp)
    end
  case 'S'
    if (cm == 'L')
      plot(T.ts,T.m.l.S-S.Zp)
    else
      plot(T.ts,T.m.a.S-S.Zp)
    end
  case 'FS'
    if (cm == 'L')
      plot(T.ts,T.m.l.FS-S.Zp)
    else
      plot(T.ts,T.m.a.FS-S.Zp)
    end
  otherwise
    error('Unknown dynamic model type.');
end
min_x = 0;
max_x = T.stime;
min_y = 0;
max_y = T.L_max;
%axis([min_x max_x min_y max_y]);
xlabel('t [second]');
if (cm == 'L')
  ylabel('dL [meter]');
else
  ylabel('Angle [degree]');
end
title(sprintf('Measurement Residual (%s)',cm));

return
```

D.5 truth.m

```
function T = truth(seed);
%
% Generate truth signals
%
% Input:
%   seed: random number seed (allows it to generate repeatable results)
%
% Output:
%  T: truth struct with fields...
%     L: actual levels struct w/ fields
%         L.C (constant)
%         L.F (filling)
%        L.S (sloshing)
%         L.FS (filling and sloshing)
%         L.S0 (slosh around 0)
%     m: measured values struct
%         level/linear fields m.l.C, m.l.F, m.l.S, m.l.FS
%         angle/nonlinear fields m.a.C, m.a.F, m.a.S, m.a.FS
%    ts: time steps [second]
%

% misc globals
d2r = pi/180.0; % constant to convert [degree] to [radian]
r2d = 180.0/pi; % constant to convert [radian] to [degree]

% seed random number generator
randn('state',seed);

% temporal parameters
T.stime = 5; % length of sim [second]
T.mrate = 50; % measurement rate [1/second]
T.nmeas = T.stime*T.mrate + 1;

% water level limits
T.L_min = 0.3; % where start filling, etc. [meter]
T.L_max = 1.0; % full [meter]

% Angular/non-linear float parameters
T.db = T.L_max + 0.02; % base for angular sensor, just above the max water [meter]
T.df = 1.25*T.db; % angular sensor arm w/ float, long enough to hit bottom on empty [meter]
T.ka = 1.0;

% Level/linear float parameters
T.kl = 1.0;

% measurement noise magnitudes
T.srl_m = 0.01*T.L_max; % stdev of level/linear sensor noise [meter]
T.sra_d = 0.01*90; % stdev of angule/non-linear sensor noise [degree]

% Filling parameters
T.delay = 1; % delay to start of filling [second]
frate = (T.L_max - T.L_min) / (T.stime - T.delay + 1/T.mrate);
% fill rate [meter/second]
fmeas = frate/T.mrate; % amount filled per measurement [meter/meas]
mf = T.delay*T.mrate; % meas of first fill motion

% Sloshing (sinusoidal) parameters
T.sf = 10/T.stime; % slosh/sin frequency [1/second]
T.sp = 0; % slosh phase [degree]
T.sm = 0.05; % slosh magnitude [meter]

%
% Generate signals
%
% Four possibilities
% 1. Constant level (C)
% 2. Filling (F)
% 3. Sloshing (S)
% 4. Filling + Sloshing (FS)
%

% sample times
T.ts = 0:1/T.mrate:T.stime;

% noise signals for each sensor type
n.l = T.srl_m*randn(1,T.nmeas); % level/linear noise
n.a = T.sra_d*randn(1,T.nmeas); % angle/non-linear noise

%
% actual/true levels (generate same number/resolution as measurements)
%

% 1. Constant level (C)
% see equation (1) in document "models"
L.C = repmat(T.L_min,1,T.nmeas); % constant signal
m.l.C = T.kl*L.C + n.l; % level measured = signal + level/linear noise
m.a.C = T.ka*r2d*asin((T.db-L.C)/T.df) + n.a;
% angle measured = signal + angle/non-linear noise

% 2. Filling (F)
% see equation (2) in document "models"
L.F = zeros(1,T.nmeas);
L.F(1:mf-1) = T.L_min;
L.F(mf:T.nmeas) = T.L_min:fmeas:T.L_max;
m.l.F = T.kl*L.F + n.l; % level measured = signal + level/linear noise
m.a.F = T.ka*r2d*asin((T.db-L.F)/T.df) + n.a;
% angle measured = signal + angle/non-linear noise

% 3. Sloshing (S)
% see equation (3) in document "models"
L.S0 = T.sm*sin(2*pi*T.ts*T.sf + T.sp*d2r); % around 0
L.S = T.L_min + L.S0;
m.l.S = T.kl*L.S + n.l; % level measured = signal + level/linear noise
m.a.S = T.ka*r2d*asin((T.db-L.S)/T.df) + n.a;
% angle measured = signal + angle/non-linear noise

% 4. Filling + Sloshing (FS)
% see equations (2) and (3) in document "models"
L.FS = L.F + L.S0; % sum of filling and sloshing (slosh around 0 not min)
m.l.FS = T.kl*L.FS + n.l; % level measured = signal + level/linear noise
m.a.FS = T.ka*r2d*asin((T.db-L.FS)/T.df) + n.a;
% angle measured = signal + angle/non-linear noise

% put true Level and measurement information in struct
T.L  = L;
T.m  = m;

return
```

Index

actual dynamics
  in parameter window, 6
  setting, 10
adding a file, 7
applet version, 4
application version, 4

checking in a file, 7
checking out a module, 7
control buttons
  Edit, 11
  Help, 11
  Plot, 11
  Step, 11
CVS, 7, 12
  comment procedures, 8
  CVSROOT, 7

data (tab-delimited)
  saving to file, 4
data repository, 6

IDE
  initial project development, 12
Implementation Manual
  updating, 8

Jama module, 4
Java
  API, 12
  design guidelines, 12
Java doclet, 12
  updating, 8–9
Javadoc comments, 9
JDK
  requirements, 12
jlfgr module, 4

Kalman filter
  what is the, 3
Kalman filter engine, 6
kftool module, 4

latex_doclet module, 4

main window, 5
Maintenance Manual
  updating, 8
manuals module, 4
Matlab code, 8, 14
matlab_code module, 4
matrix package, *see* Jama
  initial project development, 12
measurement noise
  in parameter window, 6
  setting, 10
measurement type
  in parameter window, 6
  setting, 10
modeled dynamics
  in parameter window, 6
  setting, 10

parameter window, 6
plot window, 6
plotting tool, *see* ptplot3.1p1
  initial project development, 12
plug-in
  requirements, 12
project build procedures, 8
ptplot3.1p1 module, 4

step window, 6

test_data module, 4, 8
testing the tool for correctness, 4, 8
The Kalman Filter On-line Learning Tool
  what is the, 3
truth, *see* actual dynamics

web browser
  requirements, 12
web module, 4