Trajectory & Sensor Simulation Toolkit

by: Michael Stanley

Overview

The Trajectory & Sensor Simulation Toolkit (TSim) is a collection of Matlab functions created to facilitate standardized methods of verification and characterization of sensor fusion algorithms in Matlab.

Features

- 100% native Matlab implementation. No other software is required to use this toolkit.
- Object-oriented design of trajectories and sensors.
- Create 6 Degree of Freedom description of physical motion plus translation
  - X, Y, Z position coordinates
  - quaternion orientation
- Global position can be calculated from piecewise linear representations of any of:
  - position (X, Y & Z)
  - velocity (X, Y & Z)
  - acceleration (X, Y & Z)
- Global orientation can be calculated from any of:
  - quaternion orientation (p0-p3)
  - angular velocity (X, Y & Z)
  - angular acceleration (X, Y & Z)
- Linear and spline interpolation supported for all but quaternion inputs, which is supported by SLERP.
- Position & Orientation information is automatically converted to sensor frames of reference for accelerometer, magnetometer, and gyro
- Easily expanded to include noise and other effects of physical sensors
- Extensive set of plotting and output functions, including animations for CompositeTrajectories.

Contents

Overview 1
Detailed Table of Contents 3
Introduction 7
References 9
Structural Overview 10
Trajectories 11
Specifying the Environment 48
IdealSensorPod 51
ExampleSensorPod 56
Utility Functions 69
MoCap Interfaces 75
Installation 75
Use Guidelines 76
Useful Quaternions 77
SLERP 77
Revision History 78

© Freescale Semiconductor, Inc., 2015. All rights reserved.

Freescale Trajectory Simulation Library for Matlab (TSim)

FREESCALE END-USER SOFTWARE LICENSE AGREEMENT

© 2004-2015 Freescale Semiconductor, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Freescale Semiconductor, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL FREESCALE SEMICONDUCTOR, INC. BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Detailed Table of Contents

Overview 1

Detailed Table of Contents 3

Introduction 7

References 9

Structural Overview 10

Trajectories 11

- Overview 11
- Trajectory Constructors 13
  - PositionTrajectory 13
  - AttitudeTrajectory 15
  - CompositeTrajectory 16
- Trajectory Initialization 18
  - set_acceleration 18
  - set_velocity 21
  - set_position 24
  - set_aa 27
  - set_av 30
  - quaternion_initialization 33
- Computing trajectories 36
  - compute 36
  - Example – using Spline interpolation 37
  - Example – using a low pass filter 39
- Plotting Trajectories 42
- Trajectory animations 44
  - Syntax 44
  - Description 44
  - Definitions 44
- Retiming and rotating trajectories 47
  - Syntax 47
  - Description 47

Definitions 47
Specifying the Environment 48
Syntax 48
Description 48
Definitions 48
IdealSensorPod 51
Constructor 51
Syntax 51
Description 51
Definitions 51
Plotting Sensor Outputs 54
Syntax 54
Description 54
Saving data files for use by other applications 54
Syntax 54
Description 55
Definitions 55
ExampleSensorPod 56
Standard sensor pod method definitions 56
Syntax 56
Description 57
Definitions 57
Example 57
3-Axis Sensor Model 59
Details 60
Bulk Model 60
Drift Model 60
Parameter Extraction via Allan Variance 62
Derivation of Modified Rate Random Walk Noise Model 64
Environmental Scale Factor Model 67
Quantization Model 68
1-Axis Sensor Model 68

Utility Functions 69
LPF (low pass filter) 69
Syntax 69
Description 69
Definitions 69
Example 69
standard_air_pressure 70
Syntax 70
Description 70
Definitions 70
Example 70
Rotations 71
Syntax 71
Description 71
Definitions 71
Example 71
Quaternion to/from rotation matrix conversions 72
Syntax 72
Description 72
Definitions 72
Example 72
sinusoid 73
Syntax 73
Description 73
Definitions 73
Examples 73
MoCap Interfaces 75
Installation 75
Licensing ..... **Error! Bookmark not defined.**
Useful Quaternions 77
SLERP 77
Revision History 78

Introduction

Modern consumer products include a variety of sensors used for motion capture. These may include:

- linear accelerometers
- angular rate sensors (gyros)
- magnetometers
- air pressure sensors

The figure below illustrates a basic sensor fusion system which includes the elements listed above. The magnetometer, gyro and accelerometer ideally have X, Y & Z axes which follow a "right hand rule" (RHR[1]). But this is not guaranteed. Also, assembly constraints may make it impossible for multiple sensors on a PCB to be aligned to a common frame of reference. In the general case, some Frame of Reference mapping must occur before sensor outputs are consumed.

After trimming mapped sensor values to account for board/environmental effects, the outputs are typically merged via sensor fusion algorithms to compute orientation, inclination, linear acceleration, etc. Developers of these types of algorithms have a unique problem: How to quantify performance of the fusion algorithms? Repeatable and unambiguous methods for doing so are lacking. Experimental verification is fine for "sanity tests", but these are not normally repeatable in a controlled fashion. For example, ambient magnetic interference in most office buildings changes from foot to foot within the space. Robots can be used for repeatable gyro/accelerometer testing, but they distort the magnetic field as well.

[1] Common RHR systems include NED (X=North, Y=East, Z=Down) or ENU (X=East, Y=North, Z=Up).

A method is needed to create data sets that can be used in simulations to determine algorithm performance. One such system is IMUSim [1] by Alex Young and Martin Ling at the School of Informatics, University of Edinburgh. IMUSim provides tools for the specification of position/orientation information, and subsequent translation of this information to simulated sensor outputs. It is implemented in Python and C, and released under an open source license. Unfortunately, the author has found it extremely difficult to replicate the Python/C environment in a reliable fashion.

That shortcoming led the author to develop similar capabilities within a 100% native Matlab environment. The remainder of this document details the architecture, implementation and use of these Matlab capabilities, which will henceforth be referred to as "TSim" (for "Trajectory Simulation").

References

1. D. Young, M. J. Ling and D. K. Arvind. "IMUSim: A Simulation Environment for Inertial Sensing Algorithm Design and Evaluation", in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2011)*, pp. 199-210. ACM, 2011.
2. Quaternions and Rotation Sequences, Jack B. Kuipers, Princeton University Press, 1999
3. Quaternion Math & Rotation Operators, Michael Stanley, Internal Freescale presentation, last revised 9 Dec 2011.
4. "U.S. Standard Atmosphere: 1976", NASA
5. Characterization of Various IMU Error Sources and the Effect on Navigation Performance, Warren S. Flenniken, IV, John H. Wall & David M. Bevly, Auburn University
6. Master's Thesis: Modeling Inertial Measurement Units and Analyzing the Effect of Their Errors in Navigation Applications, Warren S. Flenniken, IV., Auburn University, 2005
7. IEEE Standard 1431-2004: IEEE Standard Specification Format Guide and Test Procedure for Coriolis Vibratory Gyros
8. Annex C of IEEE Standard 952-1997: An overview of the Allan variance method of IFOG noise analysis.

Structural Overview

At the highest level, TSIM models the process as a series of interacting objects:

Environment: specifies the environment in which a set of movements is to be simulated. This includes such things as: ambient magnetic field, local altitude, temperature, frame of reference (ENU or NED), etc.

Composite Trajectory: specifies the following as a function of time: position, velocity, acceleration, orientation, angular velocity, angular acceleration

Ideal Sensor Pod: relates Environment and Composite Trajectory to one another, and generates ideal simulated values for magnetometer, accelerometer, gyro and pressure sensors.

Physical Sensor Pod: Corrupt idealized sensor readings in any number of fashions. These may include addition of Gaussian noise, offset, temperature effects, etc.

The desired output is a data set that includes not only simulated sensor values, but also reference "TRUE" values for position, orientation, etc. These can then be used to help evaluate sensor fusion algorithm performance.

These interactions are illustrated in the figure that follows. Shaded elements represent TSim class objects.

Trajectories

Overview

Trajectories are implemented as Matlab class objects. At the most fundamental level, we have an AbstractTrajectory. This object type is never used directly. It defines common components for the PositionTrajectory and AttitudeTrajectory classes. These two classes operate in similar fashions.

PositionTrajectory objects are used to compute linear acceleration, velocity and position as functions of time. The user can supply any one of those three data views in terms of X/Y/Z versus time, and TSIM will calculate the other two from it.

AttitudeTrajectory objects are used to compute orientation, angular velocity and angular acceleration as functions of time. Orientations are stored as quaternion values, requiring 4 parameters for each time point. Angular velocity and acceleration are simple X/Y/Z triples for each time point. Again, supply one of the three, and the other two can be calculated from it.

The CompositeTrajectory class inherits from both PositionTrajectory and AttitudeTrajectory. It contains all the information required to model translation and rotation in time for a given object. Raw input for both position and attitude information is in the form of simple time and data arrays. In many cases, only a few points are necessary to fully specify a sequence, as TSIM can interpolate between supplied points and resample as necessary.

In most cases, users will instantiate CompositeTrajectory objects, as both position and attitude are required to fully specify a sequence. It is expected that AttitudeTrajectory and PositionTrajectory will not be used (directly) nearly as often.

Trajectory Constructors

PositionTrajectory

Syntax:

```
[traj] = PositionTrajectory(name)
```

Description

Class constructor for the PositionTrajectory class. This is not normally called directly, as it occurs naturally from within the CompositeTrajectory class. However it CAN be used directly if the user has no need of orientation data.

The PositionTrajectory class has the following public members:

RAWPT Matlab timeseries object which contains a user supplied time/data sequence. The data values may represent acceleration, velocity or position triples, depending upon which initialize function is called.

P Matlab timeseries for time/position (X, Y & Z in meters)

V Matlab timeseries for linear velocity vs time (X, Y & Z vector components)

A Matlab timeseries for linear acceleration vs time (X, Y & Z vector components)

initial_velocity

When velocity is determined via integration of acceleration, an initial value is required. This value is initialized to [0; 0; 0] by the object constructor. This value must be changed prior to calling compute() in order to have any effect.

initial_position

When position is determined via integration of velocity, an initial value is required. This value is initialized to [0; 0; 0] by the object constructor. This value must be changed prior to calling compute() in order to have any effect.

Definitions

name = Matlab string used as trajectory identifier in reports and plots

Examples

```
traj = PositionTrajectory('Trajectory1')
```

See Also

AttitudeTrajectory, CompositeTrajectory, Matlab help on “timeseries”

AttitudeTrajectory

Syntax:

```
[traj] = AttitudeTrajectory(name)
```

Description

Class constructor for the AttitudeTrajectory class. This is not normally called directly, as it occurs naturally from within the CompositeTrajectory class. However it CAN be used directly if the user has no need of orientation data.

The AttitudeTrajectory class has the following public members:

RAWAT Matlab timeseries object which contains a user supplied time/data sequence. The data values may represent angular acceleration, angular velocity or quaternions, depending upon which initialize function is called.

O Matlab timeseries for time and quaternions ($w=q_0$, $X=q_1$, $Y=q_2$, $Z=q_3$)

AV Matlab timeseries for angular velocity versus time (X, Y & Z vector components)

AA Matlab timeseries for angular acceleration versus time (X, Y & Z vector components)

initial_orientation

When orientation is determined via integration of angular rates, an initial orientation is required. This value defaults to [1; 0; 0; 0], which corresponds to a zero rotation (unity) rotation. This value must be changed prior to calling compute() in order to have any effect.

initial_angular_velocity

When angular velocity is determined via integration of angular acceleration, an initial velocity is required. This value defaults to [0; 0; 0]. This value must be changed prior to calling compute() in order to have any effect.

Definitions

name = Matlab string used as trajectory identifier in reports and plots

Examples

```
traj = AttitudeTrajectory('Trajectory1')
```

See Also

PositionTrajectory, CompositeTrajectory, , Matlab help on “timeseries”

CompositeTrajectory

Syntax:

```
[traj] = CompositeTrajectory(name)
```

Description

Class constructor for the CompositeTrajectory class. This class is derived from both AttitudeTrajectory and PositionTrajectory classes, and therefore has the public members of both:

RAWPT Matlab timeseries object which contains a user supplied time/data sequence. The data values may represent acceleration, velocity or position triples, depending upon which initialize function is called.

P Matlab timeseries for time/position (X, Y & Z in meters)

V Matlab timeseries for linear velocity vs time (X, Y & Z vector components)

A Matlab timeseries for linear acceleration vs time (X, Y & Z vector components)

initial_velocity

When velocity is determined via integration of acceleration, an initial value is required. This value is initialized to [0; 0; 0] by the object constructor. This value must be changed prior to calling compute() in order to have any effect.

initial_position

When position is determined via integration of velocity, an initial value is required. This value is initialized to [0; 0; 0] by the object constructor. This value must be changed prior to calling compute() in order to have any effect.

RAWAT Matlab timeseries object which contains a user supplied time/data sequence. The data values may represent angular acceleration, angular velocity or quaternions, depending upon which initialize function is called.

O Matlab timeseries for time and quaternions ($w=q_0$, $X=q_1$, $Y=q_2$, $Z=q_3$)

AV Matlab timeseries for angular velocity versus time (X, Y & Z vector components)

AA Matlab timeseries for angular acceleration versus time (X, Y & Z vector components)

initial_orientation

When orientation is determined via integration of angular rates, an initial orientation is required. This value defaults to [1; 0; 0; 0], which corresponds to a

zero rotation (unity) rotation. This value must be changed prior to calling compute() in order to have any effect.

initial_angular_velocity

When angular velocity is determined via integration of angular acceleration, an initial velocity is required. This value defaults to [0; 0; 0]. This value must be changed prior to calling compute() in order to have any effect.

Definitions

name = Matlab string used as trajectory identifier in reports and plots

Examples

```
traj = CompositeTrajectory('Trajectory1')
```

See Also

PositionTrajectory, AttitudeTrajectory, , Matlab help on “timeseries”

Trajectory Initialization

set_acceleration

Syntax:

```
[traj] = traj.set_acceleration(intp, time, data)
```

Description

Use this function to initialize a trajectory using linear acceleration data sequence. Internally, this method will use time/data to initialize the traj.RAWPT timeseries object.

This method can be called for both PositionTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the P, V & A members of the class object.

Definitions

intp	Interpolation method: 'linear' or 'spline'. This is used during the compute() phase to specify how the RAWPT timeseries will be interpolated to fill the A timeseries.
time	Nx1 time vector
data	Nx3 set of X,Y,Z acceleration values in m/sec^2
traj	pointer to updated trajectory object

Example

```
path(path, '../tool'); % set this to point to the TSIM
% tool sub-directory
close all;
clc;
traj = PositionTrajectory('test1');
time = [0; 5; 10; 15]';
data = [0, 0, 0; 1, 0, 0; 1, 0, 0; 0, 0, 0];
traj = traj.set_acceleration('linear', time, data );
traj = traj.compute(0.05, [], []);
traj.plot_pt_all();     % plot all available
% plots for PositionTrajectory
```

Notice that this example is almost identical in form to those for initializing velocity and position. Pick the initialization mechanism that makes it easiest for you to achieve the desired final trajectory. Compare the figures for each example to see the effects of using the same time sequence for position versus velocity versus acceleration initialization.

Raw Acceleration Input

Calculated Position

Calculated Velocity

Interpolated Acceleration

3D Trajectory

Results obtained via acceleration initialization (Example_set_acceleration.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

set_velocity

Syntax:

```
[traj] = traj.set_velocity(intp, time, data)
```

Description

Use this function to initialize a trajectory using linear velocity data sequence. Internally, this method will use time/data to initialize the traj.RAWPT timeseries object.

This method can be called for both PositionTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the P, V & A members of the class object.

Definitions

intp	Interpolation method: 'linear' or 'spline'. This is used during the compute() phase to specify how the RAWPT timeseries will be interpolated to fill the V timeseries.
time	Nx1 time vector
data	Nx3 set of X,Y,Z velocity values in m/sec
traj	pointer to updated trajectory object

Example#1

```
path(path, '../tool'); % set this to point to the TSIM
% tool sub-directory
close all;
clc;
traj = PositionTrajectory('test1');
time = [0; 5; 10; 15]';
data = [0, 0, 0; 1, 0, 0; 1, 0, 0; 0, 0, 0];
traj = traj.set_velocity('linear', time, data );
traj = traj.compute(0.05, [], []);
traj.plot_pt_all();     % plot all available
% plots for PositionTrajectory
```

Notice that this example is almost identical in form to those for initializing acceleration and position. Pick the initialization mechanism that makes it easiest for you to achieve the desired final trajectory. Compare the figures for each example to see the effects of using the same time sequence for position versus velocity versus acceleration initialization.

Raw Velocity Input

Calculated Position

Calculated Velocity

Calculated Acceleration

3D Trajectory

Results obtained via velocity initialization (Example_ set_velocity.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

set_position

Syntax:

```
[traj] = traj.set_position(intp, time, data)
```

Description

Use this function to initialize a trajectory using linear position data sequence. Internally, this method will use time/data to initialize the traj.RAWPT timeseries object.

This method can be called for both PositionTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the P, V & A members of the class object.

Definitions

intp	Interpolation method: 'linear' or 'spline'. This is used during the compute() phase to specify how the RAWPT timeseries will be interpolated to fill the P timeseries.
time	Nx1 time vector
data	Nx3 set of X,Y,Z position values in m
traj	pointer to updated trajectory object

Example#2

```
path(path, '../tool'); % set this to point to the TSIM
                         % tool sub-directory
close all;
clc;
traj = PositionTrajectory('test1');
time = [0; 20; 40; 60]';
data = [0, 0, 0; 1, 0, 0; 1, 0, 0; 0, 0, 0];
traj = traj.set_position('linear', time, data );
traj = traj.compute(0.05, [], []);
traj.plot_pt_all(); % plot all available plots
                    % for PositionTrajectory
```

Notice that this example is almost identical in form to those for initializing velocity and acceleration. Pick the initialization mechanism that makes it easiest for you to achieve the desired final trajectory. Compare the figures for each example to see the effects of using the same time sequence for position versus velocity versus acceleration initialization.

Raw Position Input

Interpolated Position

Calculated Velocity

Calculated Acceleration

3D Trajectory

Results obtained via position initialization (Example_set_position.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

set_aa

Syntax:

```
[traj] = traj.set_aa(intp, time, data)
```

Description

Use this function to initialize a trajectory using angular acceleration data sequence. Internally, this method will use time/data to initialize the traj.RAWAT timeseries object.

This method can be called for both AttitudeTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the O, AV & AA members of the class object.

Definitions

intp	Interpolation method: 'linear' or 'spline'. This is used during the compute() phase to specify how the RAWPT timeseries will be interpolated to fill the AA timeseries.
time	Nx1 time vector
data	Nx3 set of X,Y,Z angular acceleration values in radians/sec^2
traj	pointer to updated trajectory object

Example

```
r = 2*pi();
% angular acceleration initialization
%        X      Y      Z
data = [0.0,  0.0,   0.0;
        r  ,  0.0,   0.0;
        r  ,  0.0,   0.0;
        0.0,  0.0,   0.0  ];
time = [0; 1; 5; 6];
t = AttitudeTrajectory('Traj1');
t = t.set_aa('linear', time, data);
t = t.compute(0.05, 0.01, [], []);
n=sqrt(1/3);
t.plot_at_all([n; n; n]);
```

Notice that this example is almost identical in form to that for initializing angular velocity. Pick the initialization mechanism that makes it easiest for you to achieve the desired final trajectory.

Raw Angular Acceleration Input

Orientation Displayed as Exponential Maps

Rotated Vector vs Time

Orientation Quaternion Components

Angular Velocity

Angular Acceleratoin

Results obtained via angular acceleration initialization (Example_set_aa.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

set_av

Syntax:

```
[traj] = traj.av(intp, time, data)
```

Description

Use this function to initialize a trajectory using angular velocity data sequence. Internally, this method will use time/data to initialize the traj.RAWAT timeseries object.

This method can be called for both AttitudeTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the O, AV & AA members of the class object.

Definitions

intp	Interpolation method: 'linear' or 'spline'. This is used during the compute() phase to specify how the RAWPT timeseries will be interpolated to fill the V timeseries.
time	Nx1 time vector
data	Nx3 set of X,Y,Z velocity values in radians/sec
traj	pointer to updated trajectory object

Example

```
r = 2*pi();
% angular acceleration initialization
%        X      Y      Z
data = [0.0,   0.0,    0.0;
        r  ,   0.0,    0.0;
        r  ,   0.0,    0.0;
        0.0,   0.0,    0.0  ];
time = [0; 1; 5; 6];
t = AttitudeTrajectory('Traj1');
t = t.set_av('linear', time, data);
t = t.compute(0.05, 0.01, [], []);
n=sqrt(1/3);
t.plot_at_all([n; n; n]);
```

Notice that this example is almost identical in form to that for initializing angular acceleration. Pick the initialization mechanism that makes it easiest for you to achieve the desired final trajectory.

Raw Angular Velocity Input

Orientation Displayed as Exponential Maps

Rotated Vector vs Time

Orientation Quaternion Components

Angular Velocity

Angular Acceleratoin

Results obtained via angular velocity initialization (Example_set_av.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

quaternion_initialization

Syntax:

```
[traj] = traj.quaternion_initialization(time, data)
```

Description

Use this function to initialize a trajectory using an orientation (represented by quaternions) data sequence. Internally, this method will use time/data to initialize the traj.RAWAT timeseries object.

This method can be called for both AttitudeTrajectory and CompositeTrajectory objects.

The compute() method will be used to specify how this data is used to populate the O, AV & AA members of the class object.

Definitions

time Nx1 time vector

data Nx4 set of quaternion values. Quaternions are organized as:

$q = q_0 + \mathbf{q} = \cos\theta + \mathbf{u}\sin\theta = [q_0, q_1, q_2, q_3]$

which represents a rotation of 2θ about the axis of rotation = $q = [q_1, q_2, q_3]$. Rotations are relative to the native RHR frame of reference (NED or ENU) as specified in the Environment object.

traj pointer to updated trajectory object

Example

```
num = sqrt(0.5);
%       Time  q0    q1     q2    q3
% Quaternion initialization
data = [1.0,  0.0,   0.0,  0.0;  % identity
        num,  num,   0.0,  0.0;  % 90 about X
        0.0,  1.0,   0.0,  0.0;  % 180 degrees about X
        num,  -num,  0.0,  0.0;  % -90 about X
        -1.0, 0.0    0.0,  0.0;  % back to identity
        num,  0.0,   num,  0.0;  % 90 about Y
        0.0,  0.0,   1.0,  0.0;  % 180 degrees about y
        num,  0.0,   -num, 0.0;  % -90 about Y
        -1.0, 0.0    0.0,  0.0;  % back to identity
        num,  0.0,   0.0,  num;  % 90 about Z
        0.0,  0.0,   0.0,  1.0;  % 180 degrees about Z
        num,  0.0,   0.0,  -num; % -90 about Z
        1.0,  0.0,   0.0,  0.0
]; % back to starting point

    time = 0:12;
    Vin = [1; 0; 0];  % Used for ploting purposes
    t = AttitudeTrajectory('Traj1');
    t = t.quaternion_initialization(time, data);
    t = t.compute(0.1, 0.01, [], []);
    t.plot_at_all(Vin);
```

Raw Input Data

Rotated Vector over Time

Orientations displayed as exponential maps

Orientation Quaternion Components

Angular Velocity

Angular Acceleration

Results obtained via quaternion initialization (Example_quaternion_initialization.m)

See Also

PositionTrajectory, AttitudeTrajectory, compute, Matlab help on “timeseries”

Computing trajectories

compute

Syntax:

```
[traj] = traj.compute(inc2, filter_numerator,
         filter_denominator)

[traj] = traj.compute(inc1, inc2, filter_numerator,
         filter_denominator)
```

There are two forms of the compute method. The first is applicable only to objects of type PositionTrajectory. The second applies to objects of types AttitudeTrajectory and CompositeTrajectory. Parameter definitions are consistent for the two forms of the function.

Description

The compute() method will be used to specify how initialization data in RAWPT (position) and RAWAT (attitude) is used to populate other members of the class object.

Definitions

inc1 This is the time increment using during SLERP interpolation of orientation quaternions supplied by the traj.quaternion_initialization method.

inc2 Time increment used for interpolation/retiming of acceleration, velocity, position, angular acceleration and angular velocity initiated sequences.

In the case of quaternion initialization, inc2 can be set to "0", in which case only SLERP is performed. If inc2 is greater than zero, then an additional retiming using cubic splines is performed using inc2 as the time increment. Quaternions are renormalized as a final step, regardless of the value of inc2.

In the case of other forms of initialization, setting inc2 to zero effectively skips retiming/interpolation. This might be used when importing high frequency data (from MoCap systems for instance) which is already at a high enough sample rate.

filter_numerator

row vector containing coefficients for the numerator of a filter transfer function. The form is identical to that used by the Matlab "filter" function.

filter_denominator

row vector containing coefficients for the denominator of a filter transfer function. The form is identical to that used by the Matlab "filter" function.

filter numerator/denominators are optional. Pass empty arrays ("[]") for both if additional filtering is not desired during the computation phase. The filter operator is run only on retimed/interpolated input sequence. It is NOT run on values derived from that sequence via integration/differentiation.

As an example, if:

$$Y(Z) = \frac{n_1 + n_2 z^{-1} + n_3 z^{-2}}{d_1 + d_2 z^{-1} + d_3 z^{-2}} X(Z)$$

Then filter_numerator = [n_1, n_2, n_3] and filter_denominator = [d_1, d_2, d_3].

Example – using Spline interpolation

```
% Example using spline interpolation on use specified positions
path(path, '../tool');
close all;
clc;

TimeInc=0.01;

t = PositionTrajectory('Traj1');

% time   X      Y       Z
data = [0,    0.0,    0.0,   0.0;
    1,    0.0,    0.0,   0.0;
    2,    1.0,    0.0,   0.2;
    3,    1.0,    1.0,   0.4;
    4,    0.0,    1.0,   0.6;
    5,    0.0,    0.0,   0.8;
    6,    1.0,    0.0,   1.0;
    7,    1.0,    1.0,   1.2;
    8,    0.0,    1.0,   1.4;
    9,    0.0,    0.0,   1.6];

t = t.set_position('spline', data(:,1), data(:,2:4) );
t = t.compute(TimeInc, [], []);
t.plot_pt_all();
```

As can be seen by the 3D trajectory plot below, this simple example does a very nice job of computing a spiral trajectory.

Raw Position Input Parameters

Spline Interpolated Position Parameters

Velocity Parameters

Linear Acceleration Parameters

3D Trajectory

Results obtained using spline interpolation of position data (Example_compute1.m)

Example – using a low pass filter

This example illustrates the use of a low pass filter to smooth piece-wise linear inputs prior to integration/differentiation steps. This example uses the LPF function supplied as part of TSIM to calculate a low pass filter with a cutoff frequency of 1Hz, assuming a sample rate of 200Hz and 200 taps in the filter.

```
% Create trajectory based on linear interpolation followed by low
% pass filter.  Angular velocity is square wave, position is
% triangular.
path(path, '../tool');
close all;
clc;

r = 2*pi();
d=20;

% Adata for this test is angular velocity
%        X      Y      Z
Adata = [0.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    r  ,  0.0,   0.0;
    r  ,  0.0,   0.0;
    0.0,  0.0,   0.0;
    0.0,  0.0,   0.0];
Atime = [0; 1; 2; d-2; d-1; d];

% Pdata is position data
Pdata = [0.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    10.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    0.0,  0.0,   0.0];
Ptime = [0; 1; (d/2)+1; d-3; d-2; d-1; d];

% Compute parameters for a low pass filter
% Cutoff frequency=1Hz, sample frequency = 200Hz, 200 taps
% This filter takes several seconds to run, but does a nice job
% of ensuring that our waveforms look reasonable.
% Note that it DOES introduce phase delay (which we don't
% care about)
[ N, D ] = LPF( 1, 200, 200 );

t = CompositeTrajectory('Traj1');
t = t.set_av('linear', Atime, Adata);
t = t.set_position('linear', Ptime, Pdata);
ok = t.precheck(); % Optional
t = t.compute(0.05, 0.005, N, D);
```

```
num = sqrt(1/3);
t.plot_all([num; num; num]); % vector chosen for visualizaton
purposes only
```

Orientation displayed as exponential map

Vector rotated as function of orientation

Raw angular velocity input

Orientation quaternion values

Processed angular velocity

Angular acceleration data

Raw position data

Processed position data

Velocity data

Linear acceleration data

3D trajectory

Plotting Trajectories

TSIM is capable of providing many views of a given trajectory. Most of the resultant plots have already been seen in earlier sections of this user's guide. Each plot command is implemented as a class method for the appropriate type of trajectory object. These are specified in the table below.

Plot Command	**Position**	**Attitude**	**Composite**
plot_exponential_maps()		X	X
plot_rotation_sequence(Vin)		X	X
plot_raw_at_inputs()		X	X
plot_quaternion_values()		X	X
plot_AV_coords()		X	X
plot_AA_coords()		X	X
plot_raw_pt_inputs()	X		X
plot_position_coords()	X		X
plot_velocity_coords()	X		X
plot_acceleration_coords()	X		X
plot_3D_trajectory()	X		X
plot_orientation_and_position()			X

In addition to the above, we have the following "multiple" plot commands:

Plot Command	Function
plot_at_all(Vin)	plots ALL AttitudeTrajectory plots listed in the previous table
plot_pt_all()	plots ALL PositionTrajectory plots listed in the previous table
plot_all()	Callable only from CompositeTrajectory objects. Creates plots of ALL types.

Note that most of the plot commands require no parameters. The exceptions are plot_rotation_sequence, plot_orientation_and_position, and plot_at_all. These require an input vector

for visualization purposes. Any 3X1 vector can be used, but the following has been found to provide "nice" results:

```
num = sqrt(1/3);
Vin = [num; num; num];
```

In every case, the plot command is simply called as a trajectory member function: traj.plot_command();

Every plot command has 1 optional parameter at the end of the parameter list. This is a Matlab string which should contain a directory pathname. If supplied, JPEG dumps of the plots will be printed to a directory of that name (which will be created if it does not exist).

There is one type of plot which is ONLY available for composite trajectories: plot_orientation_and_position(). This function adds orientation information to the plot created by plot_3D_trajectory to show both orientation and position in the same plot. An example is shown below.

Trajectory animations

Syntax

traj.animate(inc, dirName);

traj.animate(inc);

Description

This command is supported for CompositeTrajectory data types only. It is used to create an animation which illustrates both translational and rotational motion of the trajectory over time.

Definitions

inc	animation increment – due to the large number of points in some simulations, the resulting animation may be painfully slow. Set this variable to "1" to animate every cycle, otherwise set to some positive variable "n" to animate using every "n" cycles
dirName	pathname to a directory in which an AVI file of the animation is to be stored. This parameter is optional.

Example

```
% Copyright (c) 2012, Freescale Semiconductor

path(path, '../tool');
close all;
clc;
outputDir = 'esp_example2_outputs';

% constant definitions
sample_rate = 100;                % sensor sample rate
ts = 1/sample_rate;

% Define the environment
env = Env(Env.ENU);

r = 5*pi();

% Adata for this test is angular velocity
%   X     Y      Z
Adata = [...
    0.0,  r,   0.0;
    0.0,  r,   0.0;
    0.0,  0.0, 0.0;
    r  ,  0.0, 0.0;
    r  ,  0.0, 0.0;
    0.0,  0.0, 0.0;
    0.0,  0.0    r;
```

```
    0.0,  0.0,   r];
Atime = [0; 1; 3; 4; 6; 7; 8; 9];

% Pdata is position data
% time   X     Y      Z
Pdata = [...
    0.0,   0.0,  0.0;
    0.0,   0.0,  0.0;
    1.0,   0.0,  0.2;
    1.0,   1.0,  0.4;
    0.0,   1.0,  0.6;
    0.0,   0.0,  0.8;
    1.0,   0.0,  1.0;
    1.0,   1.0,  1.2;
    0.0,   1.0,  1.4;
    0.0,   0.0,  1.6];
Ptime = 0:9;

% Compute parameters for a low pass filter
 [ N, D ] = LPF( 1, 200, 200 );

t = CompositeTrajectory('Traj1');
t = t.set_av('linear', Atime, Adata);
t = t.set_position('spline', Ptime, Pdata );
t = t.compute(0.01, 0.005, N, D);
%t.plot_all([1;1;1]);

%t.animate(5, 'animation_example1');
t.animate(5);
```

The result of running this animation is shown below. The figure contains two plots. The left plot shows the current position of the trajectory object on the traced trajectory. The right plot is an enlargement of the same graphic object shown on the left. Both rotate to reflect the orientation of the trajectory object at each point on the trajectory.

3D trajectory
2
1
0
-1
-2
Z
2
0
-2
Y
-2
0
2
X

Retiming and rotating trajectories

Syntax

```
traj = traj.retime_then_rotate_pt(inc, RM)
traj = traj.rotate_then_retime_pt(inc, RM)
traj = traj.retime_pt(inc)
traj = traj.rotate_pt(RM)
traj = traj.retime_then_rotate_at(inc, RM)
traj = traj.rotate_then_retime_at(inc, RM)
traj = traj.retime_at(inc)
traj = traj.rotate_at(RM)
traj = traj.retime_then_rotate(inc, RM);
traj = traj.rotate_then_retime(inc, RM);
traj = traj.rotate(RM);
traj = traj.retime(inc);
```

Description

This series of functions all do basically the same thing: retime and/or rotate a trajectory. The _pt functions operate only on PositionTrajectory objects. The _at functions operate only on AttitudeTrajectory objects. Those without either suffix operate on CompositeTrajectory objects.

The only difference between the retime_then_rotate and rotate_then_retime functions is the order of execution. The results are the same in both cases, but one may be more efficient than another, depending upon whether the resulting trajectory is longer or shorter than it started.

Definitions

inc	sample period to be used for retiming purposes
RM	3x3 rotation matrix used to perform rotation and/or frame of reference mapping

Specifying the Environment

Syntax

The Env class has no user-callable functions (those member functions that do exist are called only by TSIM itself). The only member function required for users is the constructor itself. There are two forms for the constructor:

```
env = Env(
     rf,
     AccelAtRest,
     altitude,
     magnetic_calculator,
     pressure_calculator,
     temperature_calculator
)

env = Env(rf)
```

where the parameters match those described below. Use the first (expanded) form if you wish to specify non-default values for gravity, altitude, magnetic field, pressure and temperature. The second, shorter, version of the command will set these as follows:

```
altitude = 400 meters
temperature = 25C
pressure = @standard_air_pressure
IF (rf equals Env.NED)
        magnetic_calculator = [24.34777; 0; 41.47411];
        AccelAtRest = [0; 0; -magG];
IF  (rf equals Env.Win8 or Env.Android or Env.ENU)
        magnetic_calculator = [0; 24.34777; -41.47411];
        AccelAtRest = [0; 0; Env.magG];
```

"magnetic_calculator" and "AccelAtRest" are not used during trajectory computation. They are only used when computing simulated sensor outputs.

Description

TSIM includes a device class ("Env") which is used to specify physical constants and environmental parameters.

Definitions

Physical constants defined within the Env class include:

Env.magG 9.80665 m/sec^2

Env.NED	enumerated value used to specify North-East-Down reference frame
Env.ENU	enumerated value used to specify East-North-Up reference frame
Env.Win8	enumerated value used to specify Windows 8 reference frame
Env.Android	enumerated value used to Android reference frame

Environmental parameters include:

rf	Reference Frame used for the trajectory simulation. This should be one of Env.NED, Env.ENU, Env.Win8 or Env.Android. If only the frame of reference is supplied upon initialization, other parameters will default as described elsewhere. Note that NED and Win8 are defined as gravity positive systems. Accelerometer and gravity outputs from simulated sensors will be inverted for these sensors. Trajectory outputs are internally computed as "acceleration positive".
AccelAtRest	**In TSIM, this variable <u>is defined</u> as the value that an accelerometer, which is aligned with the specified global frame of reference, returns when at rest** (linear acceleration is zero). When explicitly specifying this, use the value of the gravity vector consistent with the reference frame above. For ENU, Win8 and Android, this is [0; 0; Env.magG]. For NED, this is [0; 0; -Env.magG].
altitude	Geographic altitude relative to sea level at vertical = 0;
magnetic_calculator	This can be either: • 3X1 magnetic vector specifying the local magnetic field in the specified frame of reference. Magnetic values are in microTeslas. Alternately, specify: • a Matlab function pointer to a function of the the form `M = f(time, position, temperature)` time is in seconds, position in meters, and temperature in Celsius. The TSIM examples directory contains an example_mag_calculator.m file that illustrates the form of the function.
pressure_calculator	This can be either: • a constant value specifying the local air pressure in Pascals; OR

- a Matlab function pointer to a function of the

  ```
  [P] = f(time, position, temperature, altitude)
  ```

  time is in seconds, position in meters, temperature in Celsius and altitude in meters

temperature_calculator

This can be either:

- a constant value specifying the local temperature in Celsius; OR
- a Matlab function pointer to a function of the

  ```
  [P] = f(time, position)
  ```

  time is in seconds and position in meters.

By allowing the user to specify either constants or functions for magnetic vector, air pressure and temperature, TSIM gives the user the freedom to simulate non-ideal environments of almost any type.

IdealSensorPod

Constructor

Syntax

```
isp = IdealSensorPod(env, traj, samplePeriod);
```

Description

IdealSensorPod class objects link previously defined trajectory and environment objects. The class constructor performs the minimum set of translations required to general idealized sensor readings for:

- accelerometer
- magnetometer
- gyroscope
- temperature
- air pressure

Time sequences for each of these are computed by the IdealSensorPod constructor. The new time sequences are at time intervals specified by samplePeriod.

The reader might ask: "Why do we have the ability to resample again at this point, when the trajectory functions also include that capability?" The answer is that we may want to evaluate the effect of sample period on the accuracy of an algorithm being simulated. In that case, we would use a very fine resolution for the trajectory calculations, which are done once and stored. Then decimate down for a range of values using the feature here.

Definitions

env	an object of type "Env", used to specify the simulation environment
traj	an object of type "CompositeTrajectory", used to specify translational and rotational movement over time.
samplePeriod	The sample period desired for sensor output sequences.

Public class members

traj	pointer to the original trajectory class object
env	pointer to the environment class object
ts	sample period used to create the ISP time sequences

P	Matlab timeseries for time/position (X, Y & Z in meters)
V	Matlab timeseries for linear velocity vs time (X, Y & Z vector components)
A	Matlab timeseries for linear acceleration vs time (X, Y & Z vector components) *from the sensor's perspective*
G	Matlab timeseries for acceleration due to gravity only – *from the sensor's perspective*
O	Matlab timeseries for time and quaternions ($w=q_0$, $X=q_1$, $Y=q_2$, $Z=q_3$)
AV	Matlab timeseries for angular velocity versus time (X, Y & Z vector components) *from the sensor's perspective*
M	Matlab timeseries for magnetic measurements *from the sensor's perspective*
AP	Matlab timeseries for air pressure readings *by the sensor*
T	Matlab timeseries for temperature readings *by the sensor*

Example

```
function [ output_args ] = isp_example1( input_args )

path(path, '../tool');

% constant definitions
sample_rate = 200;                % sensor sample rate
ts = 1/sample_rate;

% Define the environment
env = Env(Env.ENU)

% Define our trajectory

traj = CompositeTrajectory('isp_example1');
data = [0, 0, 0; 0, 0, 0; 0, 0, Env.magG; 0, 0, -Env.magG];
time = [0; 1; 9; 22];
traj = traj.set_acceleration('linear', time, data );

% sinusoid3 is defined in the examples directory
[ Adata, Atime ] = sinusoid3( 0.01, 22, 1, 1 );
traj = traj.set_av('linear', Atime, 5*Adata);
traj = traj.compute(0.05, 0.005, [], []);

% Link sensor pod to the environment and trajectory
isp = IdealSensorPod(env, traj, ts);

% Plot resultant values
```

```
isp.plot_all();
% save data file
isp.create_data_file('isp_example1.dat', '-ascii')
end
```

Simulated accelerometer output

Temperature is a constant for this simulation

Simulated Gyro output

Simulated magnetometer output

Simulated air pressure

Plotting Sensor Outputs

Syntax

```
isp.plot_acceleration();
isp.plot_angular_velocity();
isp.plot_magnetic_field();
isp.plot_air_pressure();
isp.plot_temperature();
isp.plot_gravity();
isp.plot_all();
```

Description

There are five possible plot types that can be generated by IdealSensorPod class object. See the previous section for graphic examples of each. You can also plot all five via the plot_all() function.

Like the trajectory plot commands, each of these takes an optional parameter to specify an output directory path. If specified, JPEG dumps of each plot will be created and stored in that directory.

Saving data files for use by other applications

Syntax

```
isp.data_dump(dirName)
```

Description

Use this command for generation of a series of ASCII .dat files. Each of these will contain 1 variable over time. The output files are stored in directory dirName, which is created if it does not exist.

filename within dirName	
TRUE_time.dat	time
magnetometer.dat	X, Y & Z magnetic sensor readings in microTeslas
accelerometer.dat	X, Y & Z accelerometer readings in gravities
gyro.dat	X, Y & Z gyroscope readings in radians/sec
TRUE_air_pressure.dat	air pressure readings in pascals
TRUE_temperature.dat	temperature readings in Celsius
TRUE_quaternion.dat	TRUE Orientation quaternion values [q_0, q_1, q_2, q_3]
TRUE_rotations.dat	TRUE Rotation matrix entries over time
TRUE_position.dat	X, Y & Z TRUE position coordinates in meters
TRUE_velocity.dat	X, Y & Z TRUE velocity values in meters/sec
TRUE_gravity.dat	X, Y & Z TRUE gravity values (in g's) from the sensor's perspective.

The dump file also places a copy of the testbench (assumed to be 1 level up in the calling stack from this function) into the output directory, along with a copy of load_sensor_dataset(), which can be used to reload this data into a set of Matlab variables.

```
[time, mag, acc, gyro, true_AP, true_temp, true_quaternions,
true_rotations, true_positions, true_velocity, true_gravity] =
load_sensor_dataset(dirName)
```

Definitions

dirName	pathname to output directory

ExampleSensorPod

The ExampleSensorPod class is derived from the IdealSensorPod AND PhysicalSensorPodInterface classes. It adds functions for "distorting" sensor outputs away from ideal values. "Distortion" can occur in-place. This means that the existing time series data inherited from the IdealSensorPod class is updated to include effects of noise, quantization, etc. Alternately the class supplies functions which can be used to obtain "sensor readings" in an iterative fashion. These functions do NOT distort the underlying time series. This style of programming may be useful when performing sensitivity analysis, where model input parameters are varied over time.

ExampleSensorPod should be viewed as a template which illustrates one possible collection of sensors. The user will want to create similar class definitions for their particular sensor cluster.

Any physical sensor class should implement the set of functions defined in PhysicalSensorPodInterface, which is defined as:

```
classdef ExampleSensorPodInterface

    properties
    end

    methods (Abstract)
      % These methods should be implemented
      % for physical sensors
         [sz] = num_points(psp)
         [time] = get_time(psp)
         [psp] = initialize_models(psp)
         [a, m, av, t, ap] = get_samples(psp, i)
         [a] = get_acc_sample(psp, i)
         [m] = get_mag_sample(psp, i)
         [g] = get_gyro_sample(psp, i)
         [t] = get_temperature_sample(psp, i)
         [ap] = get_air_pressure_sample(psp, i)
         [psp] = corrupt(psp)
    end
end
```

Standard sensor pod method definitions

Syntax

```
[sz] = psp.num_points();
[time] = psp.get_time();
[psp] = psp.initialize_models();

[a, m, av, t, ap] = psp.get_samples(i);
[a] = psp.get_acc_sample(i);
```

[m] = psp.get_mag_sample(I;)
[g] = psp.get_gyro_sample(i);
[t] = psp.get_temperature_sample(i);
[ap] = psp.get_air_pressure_sample(I;)
[psp] = psp.corrupt();

Description

num_points() returns the number of sample times in a given dataset. get_time() returns the Nx1 array of time values for the sequence. initialize_models() should contain all code necessary to initialize sensor models prior to performing computations. It must be called prior to any of the "get" functions. get_samples() returns a full set of sensor values for a point in time. The "get_<sensorType>_sample()" functions return data for a single simple of a single sensor of the appropriate type. The corrupt() function can be used to "corrupt" the timeseries objects defined by the IdealSensorPod class in such a manner that they now represent physical sensor readings, rather than ideal. This is especially handy if you want to use the IdealSensorPod plot commands.

Definitions

psp	pointer to physical sensor pod object
sz	the number of time points available in the physical sensor pod psp.
time	Nx1 array of time values corresponding to the set of physical sensor samples
i	an index in the range of 1:sz, representing a set of samples at a point in time defined by time(i)
a	3x1 accelerometer sample at time index i
m	3x1 accelerometer sample at time index i
av	3x1 gyro sample at time index i
t	temperature sensor sample at time index i
ap	barometer sample at time index i

Example

This example illustrates the case where sensor readings can be extracted from the sensor cluster iteratively over time.

```
% Copyright (c) 2012, Freescale Semiconductor

path(path, '../tool');
close all;
clc;
outputDir = 'esp_example2_outputs';

% constant definitions
sample_rate = 100;                % sensor sample rate
ts = 1/sample_rate;
```

```
% Define the environment
env = Env(Env.ENU);

r = 2*pi()/20;
d=9;

% Adata for this test is angular velocity
%        X     Y      Z
Adata = [0.0,  0.0,   0.0;
    0.0,  0.0,   0.0;
    r  ,  0.0,   0.0;
    r  ,  0.0,   0.0;
    0.0,  0.0,   0.0;
    0.0,  0.0,   0.0];
Atime = [0; 1; 2; d-2; d-1; d];

% Pdata is position data
% time   X     Y      Z
data = [0,    0.0,   0.0,  0.0;
    1,    0.0,   0.0,  0.0;
    2,    1.0,   0.0,  0.2;
    3,    1.0,   1.0,  0.4;
    4,    0.0,   1.0,  0.6;
    5,    0.0,   0.0,  0.8;
    6,    1.0,   0.0,  1.0;
    7,    1.0,   1.0,  1.2;
    8,    0.0,   1.0,  1.4;
    9,    0.0,   0.0,  1.6];

% Compute parameters for a low pass filter
% Cutoff frequency=1Hz
% frequency = 200Hz, 200 taps
% This filter takes several seconds to run, but
% does a nice job of
% ensuring that our waveforms look reasonable.
% Note that it DOES introduce phase delay
% (which we don't care about)
[ N, D ] = LPF( 1, 200, 200 );

t = CompositeTrajectory('Traj1');
t = t.set_av('linear', Atime, Adata);
t = t.set_position('spline', data(:,1), data(:,2:4) );
t = t.compute(0.01, 0.005, N, D);
t.plot_orientation_and_position([1;1;1], outputDir)

esp = ExampleSensorPod(env, t, ts);
esp = esp.initialize_models();
time = esp.get_time();

for i=1:esp.num_points()
    acc(i,:) = esp.get_acc_sample(i);
    mag(i,:) = esp.get_mag_sample(i);
```

```
    gyro(i,:) = esp.get_gyro_sample(i);
    ap(i,1) = esp.get_air_pressure_sample(i);
    temp(i,:) = esp.get_temperature_sample(i);
end
figure; plot(time, acc); title('Accelerometer'); legend('X', 'Y',
'Z');
figure; plot(time, mag); title('Magnetometer'); legend('X', 'Y',
'Z');
figure; plot(time, gyro); title('Gyro'); legend('X', 'Y', 'Z');
figure; plot(time, ap); title('Barometer');
figure; plot(time, temp); title('Thermometer');
```

3-Axis Sensor Model

TSim includes example sensor models for both 3-axis and single-axis sensor types. The two models are very similar. Users can reference these models in their physical sensor pod implementation, or they can create their own.

"Standard TSim Sensor models" for algorithm verification will have several components:

1. bulk model, which covers:
    a. axis misalignment / rotation errors
    b. axis non-orthogonality
    c. non-environmental scale factor errors
    d. non-linearity (not included in current model)
    e. scale factor asymmetry (not included in current model)
2. drift model
3. environmental scale factor variations
4. quantization model

This is an *abstract model* based on standard datasheet parameters. A model based on physics of the device can provide improved noise performance when used for system design.

The model borrows heavily from references shown in the next section.

Details

IEEE Standard 1431 defines digital gyro outputs to be modeled via the following equation:

$$S_0V_d = [I+D][1+10^{-6}\varepsilon_k]^{-1}$$

where:

- S_0 = nominal scale factor (example: (degrees/hour)/LSB)
- V_d = digital output of the sensor
- I = Inertial input
- D = Drift
- ε_k = scale factor error in ppm

This equation forms the basis for drift and environmental scale factor components of the model.

Bulk Model

The bulk model is a simple linear transformation:

$$B = G\,S + O$$

$$\begin{bmatrix} I_x \\ I_y \\ I_z \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} + \begin{bmatrix} o_x \\ o_y \\ o_z \end{bmatrix}$$

where:

- B = output of the bulk model
- G = 3x3 gain matrix which combines the effects of:
  - scaling
  - rotation / misalignment
  - axis non-orthogonality
- S = 3X1 <u>ideal</u> sensor output
- O = 3x1 offset values. This may be shown as zero rate offset in the sensor datasheet.

The model as described here does NOT include the following effects:

- non-linearity
- scale factor asymmetry

Drift Model

Graphically, a subset of the IEEE 1431 drift model looks something like:

where:

Var	IEEE 1431-2004 definition	Our interpretation	Find it
D	Drift Rate = $D_F + D_{RN} + D_{RB} + D_{RK} + D_{RR} + D_{RM} + D_{RQ}$		
D_F	Bias	zero-rate offset or quadrature error for gyros	This is redundant with the offset term in the bulk model, and will not be implemented.
D_{RN}	Random drift rate attributable to Angle Random Walk, where N is the coefficient	Gaussian noise in the sensor output.	Allan Chart – N Output noise density from datasheet X sqrt(bandwidth)[2]
D_{RB}	Random drift rate attributable to Bias Instability, where B is the coefficient		Allan Chart - B
D_{RK}	Random drift rate attributable to Rate Random Walk, where K is the coefficient	Gaussian noise in the derivative of the sensor output. Current plan is to model this as a 1st order Markov process (leaky integrator). See derivation that follows.	Allan Chart - K
D_{RR}	Random drift attributable to Ramp, where R is the		Allan Chart - R

[2] datasheet noise parameters are often expressed in units per square root(Hz).

	coefficient		
$\mathbf{D_{RM}}$	Random drift attributable to Markov noise	We have no plans to model this.	
$\mathbf{D_{RQ}}$	Random drift attributable to Output Quantization, where Q is the coefficient	Quantization is modeled directly in the TSim model (when modeled at all), and not as a noise component	Allan Chart – Q datasheet ADC sensitivity/range
$\mathbf{D_T}$	$D_T\Delta T$ = Drift rate attributable to a change in temperature		Shown in FSL datasheets as TCOff. ST shows this as OffDr
$\mathbf{D_a}$	$D_a a$ = drift rate due to acceleration		

N, B, K, R and Q coefficients" listed above are defined in terms of Allan variance. See notes below for further details. [4] provides an overview of the Allan variance method of noise analysis.

After modifying the model to account for our explicit quantization modeling and use of a 1st order Markov process for D_{RK}, the model looks like:

Parameter Extraction via Allan Variance

Reference [5] describes use of Allan variance charts to extract parameters used in the drift model. The overall Allan variance is described by:

$$\sigma_{\Omega}^{2} = \frac{R^2 n^2 T_0^2}{2} + \frac{K^2 n T_0}{3} + B^2 \left[\frac{2}{\pi}\right] \ln(2) + \frac{N^2}{n T_0} + \frac{3Q^2}{n^2 T_0^2}$$

Each term in the Allan variance normally occurs in a different region of the curve, and they exhibit different slopes. Thus R, K, B, N and Q can be extracted visually from the Allan chart (which is in terms of σ not σ^2). Alternately, the Allan variance can be curve fitted to the equation above to determine the parameters.

The figure and table below are provided as a "cheat sheet" to aid the user in extracting parameters from the Allan chart.

Error Type (in terms of Gyro parameters)		σ	Curve Slope	Where to read Coefficient Value
Quantization Noise	Q	σ(T) = √3 Q/T	-1	Q = σ(√3)
Angle Random Walk (this results from white noise on the sensor output)	N	σ(T) = N / √T	-1/2	N = σ(1)

Bias Instability (AKA "Flicker Noise"	B	$\sigma(T) \approx 0.664B$	Flat asymptote at large T	$B = \sigma(f_0) / 0.664$ f_0 = cutoff freq
Rate Random Walk (Integrated random noise)	K	$\sigma(T) = K \sqrt{(T/3)}$	+ 1/2	$K = \sigma(3)$
Ramp	R	$\sigma(T) = RT / \sqrt{2}$	+1	$R = \sigma(\sqrt{2})$

Derivation of Modified Rate Random Walk Noise Model

Adapted from:

> "Modeling Inertial Measurement Units and Analyzing the Effect of Their Errors in Navigation Applications" by Warren S. Flenniken, IV

Terms:

$b_r = b$ = walking bias
ω = input noise in $\dot{b}$
σ_b = standard deviation of walking bias b
σ_ω = standard deviation of walking bias noise ω

E[x] = expected value of x
$E[\omega^2] = \sigma^2_\omega$
T_s = sampling interval = $1/f_s$
f_s = sampling frequency
τ = time constant

1st order Markov process is a stochastic process where the future value is only dependent upon a scaled portion of the previous value plus noise (ω).

$$\dot{b} = -\frac{b}{\tau} + \omega$$

$$b_{k+1} = b_k + T_s \dot{b}$$

$$b_{k+1} = b_k + T_s \left(\omega_{(k)} - \frac{b_k}{\tau} \right)$$

$$b_{k+1} = \left(1 - \frac{T_s}{\tau} \right) b_k + T_s \omega_{(k)}$$

Essentially, this can be viewed as a "leaky integrator":

$$E[bb^T] = E\left[\left(\left(1-\frac{T_s}{\tau}\right)b_k\right)\left(\left(1-\frac{T_s}{\tau}\right)b_k\right)^T\right] + E\left[T_s\omega_{(k)}\left(T_s\omega_{(k)}\right)^T\right]$$

$$\sigma_b^2 = \left(1-\frac{T_s}{\tau}\right)^2 \sigma_b^2 + {T_s}^2\sigma_\omega^2$$

$$\sigma_b^2\left(1-\left(1-\frac{T_s}{\tau}\right)^2\right) = {T_s}^2\sigma_\omega^2$$

$$\sigma_b^2\left(1-\left(1-\frac{2T_s}{\tau}+\frac{{T_s}^2}{\tau^2}\right)\right) = {T_s}^2\sigma_\omega^2$$

$$\sigma_b^2\left(\frac{2T_s}{\tau}-\frac{{T_s}^2}{\tau^2}\right) = {T_s}^2\sigma_\omega^2$$

$$\sigma_b^2\left(\frac{2}{T_s\tau}-\frac{1}{\tau^2}\right) = \sigma_\omega^2$$

Because τ is generally a large value, the inverse square can be assumed to be3 very small, so the 2nd term in the LHS can be ignored:

$$\sigma_\omega^2 = \frac{2\sigma_b^2}{T_s\tau}$$

model

$$\boldsymbol{\omega}_{(k)} = \boldsymbol{\sigma}_\omega\, \boldsymbol{v} = \sqrt{\frac{\boldsymbol{2\sigma_b^2}}{\boldsymbol{T_s\tau}}}\boldsymbol{v}$$

where v ~ [0,1] => (zero mean, σ=1)

$$\boldsymbol{b_{k+1} = \left(1 - \frac{T_s}{\tau}\right) b_k + T_s \omega_{(k)}}$$

Matlab code to demonstrate these equations is shown below:

```
function [ output_args ] = MarkovTest( input_args )
% Generate plot for example of random walk using 1st
% order Markov process.
close all;
clc;
sigmaB=1;
Ts = 0.01;
tau = 10000;
sigma_omega = sqrt(2*sigmaB^2/(Ts*tau));
lastB=0;
for i=1:1000000
    wk(i) = sigma_omega*randn(1);
    B(i) = wk(i)*Ts + lastB - (Ts/tau)*lastB;
    lastB = B(i);
    time(i)=Ts*i;
end
plot(time, B);
xlabel('seconds');
ylabel('b(k)');
title('1st Order Markov Process');
sb=std(B);
fprintf('Sigma(omega)=%f\n', sigma_omega);
fprintf('The standard deviation of the generated waveform is
%f.\n', sb);
end
```

The following figures illustrate 3 different runs with different values of tau:

tau = 100

Sigma(omega)=1.414214

The standard deviation of the generated waveform is 0.973796.

tau = 500

Sigma(omega)=0.632456

The standard deviation of the generated waveform is 0.855677.

tau = 500

Sigma(omega)=0.447214

The standard deviation of the generated waveform is 1.007037.

Environmental Scale Factor Model

This model currently includes only temperature in the list of parameters affecting scale factor. Specifically:

$$scale\ factor = \begin{bmatrix}1\\1\\1\end{bmatrix} + (T-25)(1e-6)\begin{bmatrix}\varepsilon_{Tx}\\\varepsilon_{Ty}\\\varepsilon_{Tz}\end{bmatrix}$$

Term	Description	Find It
ΔT	change in temperature	
e_{Tx}, e_{Ty}, e_{Tz},	scale factor error due to change in temperature for X, Y and Z axes	Shown in FSL datasheets as TCS_o – Sensitivity change vs temperature

Quantization Model

Rather than treat quantization as a noise adder, we will model it directly using the following:

```
continous_output = clamp(continuous_output, range);

Vd = range_per_lsb * round(continuous_output/range_per_lsb);
```

Notice that this model does not change the units of the process being studied. We are not dealing with number bits. We are still dealing with radians/sec, g's, microTeslas, etc. But the output waveforms will show the effects of quantization.

1-Axis Sensor Model

The single axis model is identical in form to the 3-axis model, with the following exceptions:

- Only 1 axis is modeled, therefore all model input parameters are single floating point numbers instead of 3x1 arrays.
- Acceleration-dependent drift term is omitted.

Utility Functions

Several utility functions, not associated with any specific trajectory, environment or sensor, are included in TSIM.

LPF (low pass filter)

Syntax

```
[ Num, Denom ] = LPF( Fc, Fs, N )
```

Description

Given a cutoff frequency, sample frequency and number of filter taps, the LPF function will compute numerator and denominator for a low pass filter description which can be used with the trajectory compute functions.

Definitions

Fc	Cutoff (-3dB) frequency for the desired filter
Fs	sample rate of the data stream to be filtered
N	number of taps allowed for the resulting filter implementation
Num	Matlab row vector containing low pass filter coefficients
Denom	always “1”

Example

See the 2nd example in the “Computing Trajectories” section.

standard_air_pressure

Syntax

```
[ P ] = standard_air_pressure(
      time,
      position,
      temperature,
      altitude
)
```

Description

Computes air pressure using the NASA "U.S. Standard Atmosphere, 1976" model, which can be simplified to:

$$P = P_0(T/(T+LH))^C$$

where:

P0 = 101325 Pascals

T is temperature in Kelvin

L = -6.5E-3 K/M

H = position + altitude (Note: invert position in the calling function for NED)

C =-5.25588

Definitions

P	computed air pressure
Position	is of form [X; Y; Z] in meters.
Temperature	is in Celsius
Altitude	is the differential between vertical=0 and sea level in meters
Time	is not used by this version of the function, but is a required parameter for the template used within .

Example

Normally the name of this function is passed as a parameter to "Env" constructors. It is not usually called directly. The user may choose to use their own function for describing air pressure as a function of these same input variables. The function prototype must match that of this function in order for TSim to operate correctly.

Rotations

Syntax

```
rotation_matrix = rotate_x_degrees( angle )
rotation_matrix = rotate_y_degrees( angle )
rotation_matrix = rotate_z_degrees( angle )
rotation_matrix = rotate_x_radians( angle )
rotation_matrix = rotate_y_radians( angle )
rotation_matrix = rotate_z_radians( angle )
```

Description

These functions provide a quick method for computation of a rotation matrix given rotation about X, Y or Z axis by a given angle in either degrees or radians, depending upon the function.

Definitions

angle	desired angle in radians or degrees (depending upon the function)
rotation_matrix	3X3 rotation matrix which can be used to rotate any 3X1 vector.

Example

```
>> RM = rotate_x_degrees(45)

RM =

   1.0000        0        0

        0   0.7071   0.7071

        0  -0.7071   0.7071

>> RM = rotate_x_degrees(45)*rotate_y_degrees(30)

RM =

   0.8660        0  -0.5000

   0.3536   0.7071   0.6124

   0.3536  -0.7071   0.6124
```

Quaternion to/from rotation matrix conversions

Syntax

```
RM = RM_from_quaternion( q )
q = RM_to_quaternion( RM )
```

Description

Convert between 3X3 rotation matrix and quaternions.

Definitions

RM	3X3 rotation matrix
q	quaternions in $[q_0, q_1, q_2, q_3]$[3] form.

Example

```
clc;
RM = rotate_x_degrees(45)
q = RM_to_quaternion(RM)
newRM = RM_from_quaternion(q)
```

yields

```
RM =
  1.0000       0       0
       0  0.7071  0.7071
       0 -0.7071  0.7071

q =
  0.9239
  0.3827
       0
       0

newRM =
  1.0000       0       0
       0  0.7071  0.7071
       0 -0.7071  0.7071
```

[3] or alternately [w, x, y, z]

sinusoid

Syntax

```
s = sinusoid(
    offset,
    magnitude,
    frequency,
    start_time,
    stop_time
)

data = s.value(ts)
```

Description

The sinusoid class is used for constructing sinusoid waveforms, normally for use in trajectory initialization.

Definitions

offset	sin wave offset from zero
magnitude	magnitude of the sinusoid to be generated
frequency	frequency of operation in Hertz
start_time	time at which to start oscillation (minimum start_time = 0)
stop_time	time at which to cease oscillations
s	sinusoid class object
ts	sample period
data	NX1 generated data sequence

Examples

```
clc;
ts = 0.01;
time=(0:ts:30);
%s = sinusoid(offset, magnitude, frequency, start_time,     stop_time);
s1 = sinusoid( 0,      1,          .5,         10,             20);
d1 = s1.value(time);
plot(time,d1);
title('sinusoid1 plot');
```

There are a number of utility functions built on top of the sinusoid class which are housed in the examples directory of the TSim installation. These are not considered a formal part of TSim, and can be copied and modified at will.

MoCap Interfaces

Still Pending. The intention is to add interfaces for loading motion capture files from the Carnegie Mellon Motion Capture repository at http://mocap.cs.cmu.edu/.

Installation

TSim is maintained on the MEMS Industry Group GitHub site at:

https://github.com/memsindustrygroup/TSim.

Git is the world's most popular configuration management system, and can be freely downloaded. You can find directions for downloading and using Git at https://help.github.com/articles/set-up-git/.

Those familiar with Git will find that installation is trivial

1. create a directory in which you wish to install TSIM
2. open a Git shell in that directory
3. git init
4. git remote add origin https://github.com/memsindustrygroup/TSim.git
5. git pull origin master

Alternately, you can download the entire repository as a zip file using the "Download ZIP" button on the right hand side of the GitHub page. Then just unzip the TSIM file to a convenient directory in your Matlab library.

Add the resulting TSIM/tool directory to your Matlab path variable as follows:

```
path(<path to TSIM/tool>, path);
```

TSIM contains the following subdirectories:

docs	contain this manual and related tool documentation
tool	contains class definitions for the TSIM
examples	contains example code used to generate figures in this document, along with other examples which may be found to be useful.
tests	unit tests used by the developer of TSIM to check various functions. Not needed for most users (can be deleted)

Use Guidelines

TSIM is distributed under the BSD-style license shown on page 2 of this document.

TSIM is intended to act as an enabler in creation of standard methods for algorithm validation and characterization.

A number of TSIM classes allow the user to specify their own functions for customization. This can be done without modification to the underlying TSIM class definitions. Modification of those TSIM classes is allowed, but discouraged.

Use guidelines include:

1. User's are free to modify the source code, but should not publish simulation results based upon proprietary modifications of the base TSIM class definitions.
2. Published simulation results based upon TSIM shall include or reference the specific trajectory, environment and device models used.
3. Please cite this document in any work which makes use of TSIM.
4. TSIM users are encouraged to make and distribute their own sensor models which inherit from the PhysicalSensorPod class type.

Useful Quaternions

w	x	y	z	Description
1	0	0	0	Identity quaternion, no rotation
0	1	0	0	180' turn around X axis
0	0	1	0	180' turn around Y axis
0	0	0	1	180' turn around Z axis
sqrt(0.5)	sqrt(0.5)	0	0	90' rotation around X axis
sqrt(0.5)	0	sqrt(0.5)	0	90' rotation around Y axis
sqrt(0.5)	0	0	sqrt(0.5)	90' rotation around Z axis
sqrt(0.5)	-sqrt(0.5)	0	0	-90' rotation around X axis
sqrt(0.5)	0	-sqrt(0.5)	0	-90' rotation around Y axis
sqrt(0.5)	0	0	-sqrt(0.5)	-90' rotation around Z axis

SLERP

When interpolating between positions on a sphere, straight linear interpolation may not produce desired results. The SLERP algorithm interpolates between points on a sphere such that the change in angle for any Δt between the two points is constant.

Referencing the figure above, we want to traverse from q0 to q1 in time t=1

$$q_0 \bullet q_1 = |q_0|\,|q_1| \cos(\Omega)$$

where Ω is the angle between q_0 and q_1 (this is the definition of the dot product)

Both q_0 and $q_1$have length 1 for this problem, therefore

$\Omega = \cos^{-1}(q_0 \bullet q_1)$

Define $\theta = t * \Omega$

We traverse the **angle** between q_0 and q_1 in a linear fashion, interpolating accordingly:

$$\mathbf{Slerp(q_0, q_1, t) = q_0 * \sin((1-t)\,\Omega)/\sin(\Omega) + q_1 * \sin((t\,\Omega)/\sin(\Omega)}$$

Revision History

Rev #	Date	Revised by	Description
0.1	23 Apr 2012	Mike.Stanley@freescale.com	1st draft
0.2	26 Apr 2012	Mike.Stanley@freescale.com	Merged modeling notes into the mainstream document
1.0	1 May 2012	Mike.Stanley@freescale.com	Added PhysicalSensorPodInterface class and expanded modeling examples. Added cover page.
1.1	6 May 2012	Mike.Stanley@freescale.com	Added animation for CompositeTrajectories. These can be saved as AVI files.
1.2	31 May 2012	Mike.Stanley@freescale.com	Added TRUE_gravity output
1.3	7 July 2014	Mike.Stanley@freescale.com	Modified constructor for Env class so that only the frame of reference must be supplied. Added computations in sensor outputs to support same. Env.one_g is now Env.magG (read as: magnitude of gravity). All .m files in the examples and tests directories have been updated and run.
1.4	22 Sept 2015	Mike.Stanley@freescale.com	Updated for public release via MIG GitHub site. There are no changes in the library itself. Only the software license and documentation.