JitterTime 1.2—Reference Manual

Anton Cervin

LUND UNIVERSITY

Department of Automatic Control

Technical Report TFRT-7658, version 3
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

This technical report describes JITTERTIME, a Matlab toolbox for calculating the time-varying state covariance of a mixed continuous/discrete linear system driven by white noise. It also integrates a quadratic cost function for the system. The passing of time and the updating of the discrete-time systems are explicitly managed by the user in a simulation run. Since the timing is completely handled by the user, any complex timing scenario can be analyzed, including scheduling algorithms, timing jitter and drift, and asynchronous execution in distributed systems. Some examples of how the toolbox can be used to evaluate the control performance of time-varying systems are given.

Contents

1. Introduction

JITTERTIME is a spin-off from the Matlab toolbox JITTERBUG [**LinCer02**] and can be used for calculating the performance of a controller under non-ideal timing conditions. Examples of such conditions include delay and jitter due to CPU and network scheduling, lost samples or lost controls due to packet loss or execution overruns, and aperiodic behavior due to clock drift, asynchronous nodes, and random sampling. Both JITTERBUG and JITTERTIME evaluate a quadratic cost function for a mixed continuous-time/discrete-time linear system driven by white noise and/or deterministic impulse disturbances. The main difference is the timing model. In JITTERBUG, the timing of the discrete systems are governed by random delays with specified probability density functions. This allows the total system to be treated as a jump-linear system, and covariance can be calculated by solving a set of linear equations. In JITTERTIME, however, the timing is arbitrary and completely driven by the user. This allows for more complex timing scenarios to be analyzed, including scheduling algorithms with long-term timing dependencies and asynchronous execution in distributed control systems. For deterministic timing scenarios over a finite horizon (or a repeating hyperperiod), the performance is evaluated exactly. For stochastic timing scenarios, however, lengthy Monte Carlo simulations can be needed to obtain results with high confidence. This is a major drawback of the tool compared to JITTERBUG.

2. Using the Toolbox

Getting Started

JITTERTIME consists of a small number of functions and requires Matlab with the Control System Toolbox (any reasonably recent version). Simply add the directory containing the JITTERTIME functions to the Matlab path.

Creating JitterTime Models

A JITTERTIME model is created by adding and connecting any number of continuous-time and discrete-time linear systems. Throughout, MIMO systems are allowed, and a system may receive its inputs from several other systems. All noise sources in the model are assumed independent.

Continuous-time systems must be strictly proper and can be specified as state-space or transfer-function (or zpk) objects. Optionally, continuous-time white noise with a given intensity can be added to the system. A continuous-time quadratic cost function can also be specified. For state-space systems, a non-zero initial state may also be given.

Discrete-time systems must be proper and can be specified as state-space or transfer-function (or zpk) objects. Optionally, discrete-time white noise with a given variance can be added to the system. For state-space systems, a non-zero initial state/output vector may also be given. When the system is executed, its inputs are read (sampled), noise is added, and its states and outputs are updated. Between executions, all states and output signals are held. A continuous-time quadratic cost function (acting on the piece-wise constant signals) can also be specified. Multiple versions of the dynamics for same system can be specified to allow for gain scheduling or other parametric behavior of the system during simulation.

An example of a simple JITTERTIME model is given in Figure 1. It models a sampled-data control loop with a continuous-time plant, $P(s)$, an ideal sampler, $S(z) = 1$, and a combined discrete-time controller and zero-order hold actuator, $CA(z)$. When the sampler executes, it reads the measurement signal y with some added discrete noise v_d. When the controller executes, it updates the control signal u, which is fed back to the plant. Assuming that the systems P, CA and S and the noise and cost parameters have already been specified, the lines of code needed to construct the model are:

```
N = jtInit;                         % Initialize JitterTime
N = jtAddContSys(N,1,P,3,R1,Q);     % Add sys 1 (P), input from sys 3
N = jtAddDiscSys(N,2,S,1,R2);       % Add sys 2 (S), input from sys 1
N = jtAddDiscSys(N,3,CA,2);         % Add sys 3 (CA), input from sys 2
N = jtCalcDynamics(N);              % Calculate the internal dynamics
```

The variable `N` is a data structure that contains all the added systems. Every system is identified by a unique number. `jtCalcDynamics` checks that all system connections are correct and creates a large state-space model of the total system. Each continuous-time system requires n states, and each discrete-time system requires $n+r$ states, where n is the system order and p is the number of system outputs. Internally, the order of the model states corresponds to the order that the systems have been added.

Simulating a JitterTime Model

The model is simulated by a number of calls to `jtPassTime`, `jtExecSys`, `jtResetSys` and `jtStateDynamics`, in any order. `jtPassTime` (or `jtPassTimeUntil`) is used to simulate the passing of time and integrates the covariance of all continuous-time systems. It also integrates the cost of all systems. `jtExecSys` executes a given discrete-time system, which is assumed to take zero time. An optional argument can be used to control what version of the system dynamics should be applied. In the following example, the simple control loop model described above is simulated for 1000 periods of length h. The sampler executes at the start of each period, while the controller/actuator executes after a random delay, uniformly distributed in $[0,h]$.

```
for t = 1:1000
   N = jtExecSys(N,2);           % Execute sys 2 (S)
   tau = rand*h;                 % Generate random delay
   N = jtPassTime(N,tau);        % Pass time until actuation
   N = jtExecSys(N,3);           % Execute sys 3 (CA)
   N = jtPassTime(N,h-tau);      % Pass time until end of period
end
```

During a simulation, the model variables `N.P`, `N.m`, `N.J`, and `N.Tsim` are updated after each call to `jtPassTime` and `jtExecSys`. `N.P` contains the covariance matrix of all the states in the model, while `N.m` contains the state mean values. `N.J` holds the accumulated cost, and `N.Tsim` keeps track of the simulation time. All of these variables are initialized to zero in `jtCalcDynamics`. `N.J` and `N.Tsim` may be reset by the user at any time during a simulation. This can be useful for, for example, skipping the transient behavior at the start of a simulation.

Obtaining the Results

Depending on the purpose, the model variables `N.P`, `N.m`, `N.J`, and `N.Tsim` can be logged by the user during a simulation and analyzed afterwards. If the purpose is to calculate the average cost per time unit, this can simply be done as follows:

```
Javg = N.J / N.Tsim;
```

3. Theory

JitterTime is based on well-known theory for linear stochastic systems, see, e.g., [**AstWit97**]. The toolbox aids the user in setting up a mixed continuous/discrete linear system model, driven by random noise and/or deterministic disturbances, and calculating the evolution of its total state

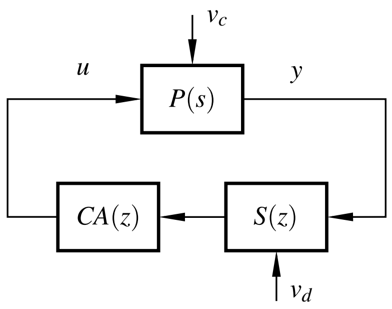

Figure 1 A simple JitterTime model.

covariance. The calculations themselves are trivial. At time zero, the model state covariance P and mean value m are assumed to be zero. Between events (executions of discrete-time systems), the covariance and mean value evolve according to

$$\begin{aligned}\dot{P}(t) &= AP(t) + P(t)A^T + R_c,\\ \dot{m}(t) &= Am(t),\end{aligned}$$

where A describes the total continuous dynamics and R_c is the intensity of the total continuous noise. Any discrete system states are kept constant by corresponding zeros in the A matrix. When a discrete-time system k is executed at time t_k, the covariance and mean value are immediately updated according to

$$\begin{aligned}P(t_k^+) &= A_{dk}P(t_k)A_{dk}^T + R_d,\\ m(t_k^+) &= A_{dk}m(t_k),\end{aligned}$$

where A_{dk} describes the discrete state transition for system k and its connection to other systems, while R_d is the variance of the discrete noise. The increase in cost between two events is given by

$$\Delta J = \int_{t_k}^{t_{k+1}} \big(\mathrm{tr}\ Q_c P(t) + m^T(t) Q_c m(t)\big)\, dt,$$

where Q_c is the cost matrix for the total model.

At each call to `jtPassTime`, the continuous dynamics, noise, and cost are internally sampled using the helper function `calcc2d` from JITTERBUG (see [**LinCer02**] for details). Any linear system dynamics may be simulated, and the tool does not check for stability. If the model is indeed unstable, the state covariance P and the mean value m will grow unbounded.

4. Examples

Example 1—Simple Control Loop with Controller Activation

Consider the simple control loop in Figure 1, where the process is assumed to be an integrator driven by unit-intensity white noise,

$$\dot{y}(t) = u(t) + v_c(t), \quad y(0) = 0.$$

The control objective is to minimize the following cost function:

$$J(t) = \int_0^t y^2(\tau)\, d\tau.$$

The stationary minimum-variance controller (see [**AstWit97**]) is given by the proportional feedback

$$u(t_k) = -\frac{1}{h}\frac{3+\sqrt{3}}{2+\sqrt{3}}\, y(t_k),$$

where h is the sampling interval.

Assuming $h = 1$ and that the controller is activated at $t = 3$, the process variance, $P(t) = \mathrm{E}\, y^2(t)$, and the accumulated cost, $J(t)$, are calculated using JITTERTIME. The variance and cost are logged every 0.1 time units to show the inter-sample behavior, and results are plotted in Figure 2. It is seen that the variance grows linearly when the process runs in open loop, as expected. When the controller is activated at $t = 3$, the variance decreases and soon reaches a stable periodic behavior. It can be shown that the average variance approaches

$$\bar{P} = \frac{3+\sqrt{3}}{6} \approx 0.79,$$

and the simulation agrees with this. The complete code for the example is given below:

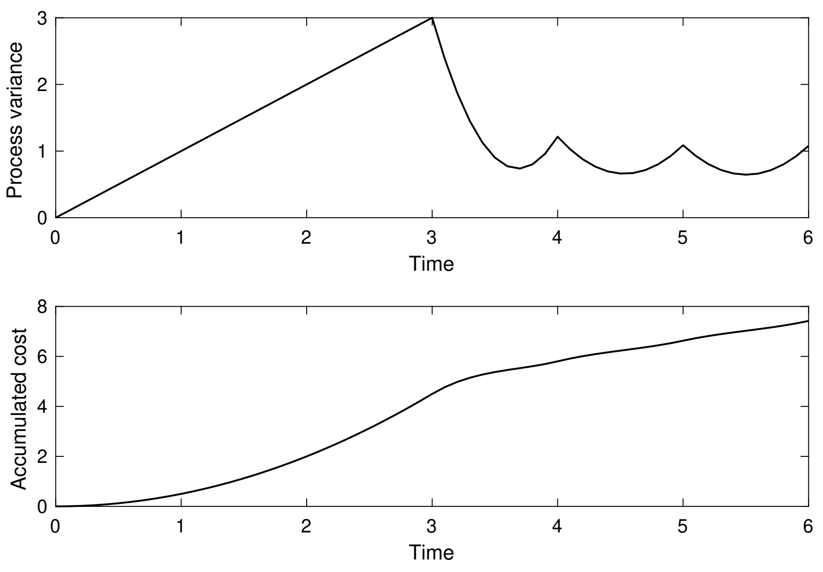

Figure 2 Example 1: The controller is activated at $t = 3$ and then executes once per time unit.

```
%%% Example 1: Simple Control Loop with Controller Activation %%%

% Define process, controller and sampler
P = ss(0,1,1,0);  % Integrator process
Qc = diag([1 0]); % State and control cost
R1c = 1;          % State noise
h = 1;            % Sampling period
CA = -1/h*(3+sqrt(3))/(2+sqrt(3)); % Minimum-variance controller
S = 1;            % Sampler

% Define the JitterTime model
N = jtInit;
N = jtAddContSys(N,1,P,3,R1c,Qc); % Add sys 1 (P)
N = jtAddDiscSys(N,2,S,1);        % Add sys 2 (S)
N = jtAddDiscSys(N,3,CA,2);       % Add sys 3 (CA)
N = jtCalcDynamics(N);

% Simulate the system and log the results
Nsteps = 6;       % Large time steps (control periods)
dt = h/10;        % Small time steps (for plotting)
l = 0;
tvec = [];pvec = []; Jvec = [];
for k = 1:Nsteps
  for j = 1:10
    l = l+1;
    tvec(l) = N.Tsim; pvec(l) = N.P(1,1); Jvec(l) = N.J;
    N = jtPassTime(N,dt);
  end
  l = l+1;
  tvec(l) = N.Tsim; pvec(l) = N.P(1,1); Jvec(l) = N.J;
  if k >= 3       % Activate controller after 3 samples
    N = jtExecSys(N,2);
    N = jtExecSys(N,3);
  end
end

% Plot the results
subplot(211)
plot(tvec,pvec)
```

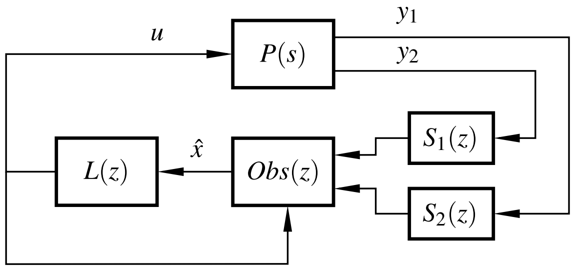

Figure 3 Example 2: JITTERTIME model with two samplers that execute asynchronously.

```
xlabel('Time')
ylabel('Process variance')
subplot(212)
plot(tvec,Jvec)
xlabel('Time')
ylabel('Accumulated cost')
```

Example 2—Asynchronous Kalman Filtering with Two Sensors

In this example, it is assumed that a process to be controlled is equipped with two different sensors. The first sensor takes measurements at regular intervals. The second sensor is much slower, but delivers its measurements asynchronously, as soon as they are ready. The model is illustrated in Figure 3. (For sake of clarity, the noise inputs are not shown.) The process $P(s)$ is assumed to be an inverted pendulum, described in state-space form as

$$\begin{aligned}\dot{x}(t) &= \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}x(t) + \begin{bmatrix}0\\1\end{bmatrix}u(t) + \begin{bmatrix}0\\1\end{bmatrix}v_c(t),\\ y(t) &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}x(t)\end{aligned}$$

Here, $y_1 = x_1$ corresponds to the pendulum angle, $y_2 = x_2$ corresponds to the angular velocity, and v_c is input noise with intensity $R_{1c} = 1$. The periodic sensor, $S_1(z) = 1$, has additive noise with variance $R_{21} = 0.1$, while the aperiodic sensor, $S_2(z) = 1$, has measurement noise with variance $R_{22} = 0.01$. The design objective is to minimize the quadratic cost function

$$J(t) = \int_0^t \Big(x_1^2(\tau) + 0.1x_2^2(\tau) + 0.1u^2(\tau)\Big)\,d\tau.$$

The controller consists of a Kalman filter $Obs(z)$ and a static feedback gain $L(z)$. The observer operates in two modes (versions). In the periodic mode, a standard Kalman filter with direct term based on the measurement y_1 is implemented as

$$\hat{x}(t_k) = (I - K_1C_1)\big(\Phi_1\hat{x}(t_{k-1}) + \Gamma_1 u(t_{k-1})\big) + K_1y_1(t_k),$$

where Φ_1, Γ_1, and K_1 are calculated assuming the constant sampling interval $h = 0.15$. When an aperiodic sample of y_2 arrives at time t_j, an extra measurement update step is executed as

$$\hat{x}(t_j^+) = (I - K_2C_2)\hat{x}(t_j) + K_2y_2(t_j),$$

where K_2 is designed assuming the average sampling interval $\bar{h}_2 = 0.5$. Both observers and the feedback gain L can be computed using `lqgdesign` from JITTERBUG.

In Figure 4, the performance of the control system is simulated, either using only y_1 or using both y_1 and y_2. It is seen that the inclusion of the aperiodic measurements y_2 decreases the state variance when each new sample arrives. A longer simulation run reveals that the average cost decreases by 18% by also including sensor 2. The complete code for the example is given below:

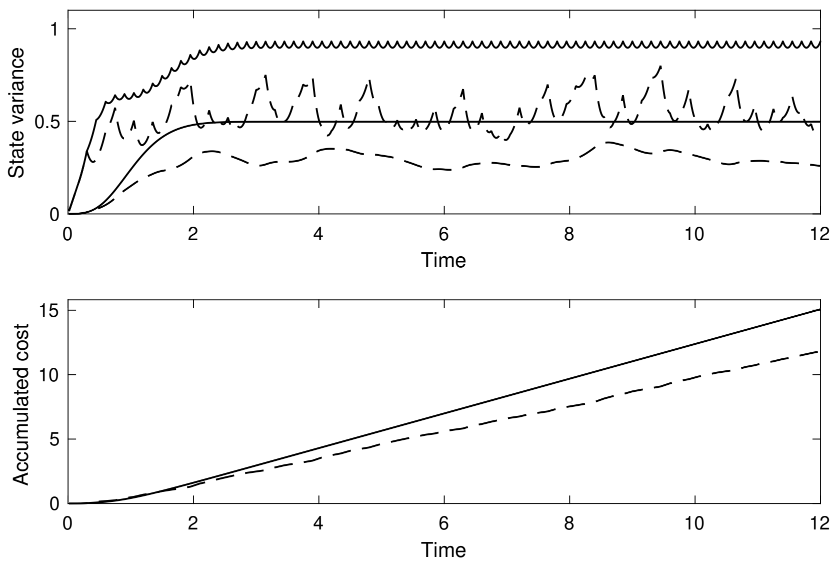

Figure 4 Example 2: Output feedback based on one periodic sensor (full) or combined periodic and aperiodic sensors (dashed).

```
%%% Example 2: Asynchronous Kalman Filtering with Two Sensors %%%

% Define the process (inverted pendulum) and design objective
A = [0 1; 1 0];
B = [0; 1];
P = ss(A,B,eye(2),0);      % Process dynamics
S1 = [1 0];                % Sensor 1 (position)
S2 = [0 1];                % Sensor 2 (velocity)
Qc = diag([1 0.1 0.1]);    % Cost function
R1c = B*B';                % Process input noise
R21 = 0.1;                 % Measurement noise on sensor 1
R22 = 0.01;                % Measurement noise on sensor 2

% Design state feedback and the two Kalman filters
h1 = 0.15;
[~,L,~,~,K1,sysd1] = lqgdesign(P(1),Qc,R1c,R21,h1);
h2bar = 0.5;
[~,~,~,~,K2,sysd2] = lqgdesign(P(2),Qc,R1c,R22,h2bar);

% Formulate Kalman filter based on y1
[Phi1e,Gam1e,C1e] = ssdata(sysd1);
IK1C1 = eye(size(Phi1e)) - K1*C1e;
Obs1 = ss(IK1C1*Phi1e,[IK1C1*Gam1e K1 0*K2],IK1C1*Phi1e,[IK1C1*Gam1e K1 0*K2],-1);

% Formulate Kalman filter based on y2
[Phi2e,Gam2e,C2e] = ssdata(sysd2);
IK2C2 = eye(size(Phi2e)) - K2*C2e;
Obs2 = ss(IK2C2,[0*Gam2e 0*K1 K2],IK2C2,[0*Gam2e 0*K1 K2],-1);

% Define the JitterTime model
N = jtInit;
N = jtAddContSys(N,1,P,5,R1c,Qc);          % Add sys 1 (Process)
N = jtAddDiscSys(N,2,S1,1,diag([R21 0]));  % Add sys 2 (Sensor 1)
N = jtAddDiscSys(N,3,S2,1,diag([0 R22]));  % Add sys 3 (Sensor 2)
N = jtAddDiscSys(N,4,{Obs1,Obs2},[5 2 3]); % Add sys 4 (Observers)
N = jtAddDiscSys(N,5,-L,4);                % Add sys 5 (Feedback)
N = jtCalcDynamics(N);
```

```
% Simulate the system
l = 0;
tvec = []; p1vec = []; p2vec = []; Jvec = [];
for k = 1:12/h1  % Simulate for 12 time units
  for j = 1:10   % Small time steps (for plotting)
    l = l+1;
    N = jtPassTime(N,h1/10);
    tvec(l) = N.Tsim; p1vec(l) = N.P(1,1); p2vec(l) = N.P(2,2); Jvec(l) = N.J;
    if rand < 0.05
      N = jtExecSys(N,3);    % Run the aperiodic sampler
      N = jtExecSys(N,4,2);  % Execute the aperiodic observer update
    end
  end
  N = jtExecSys(N,2);     % Run the periodic sampler
  N = jtExecSys(N,4,1);   % Execute the regular observer
  N = jtExecSys(N,5);     % Execute the state feedback
end

% Plot the result
subplot(211); plot(tvec,p1vec,tvec,p2vec)
xlabel('Time'); ylabel('State variance')
subplot(212); plot(tvec,Jvec)
xlabel('Time'); ylabel('Accumulated cost')
```

Example 3—Control Task Period Selection under Fixed-Priority Scheduling

Again consider the simple control loop in Figure 1. The process to be controlled is again an inverted pendulum, see Example 2. Assuming process input noise with intensity $R_{1c} = 1$ and measurement noise with variance $R_2 = 0.01$, the goal is to minimize the quadratic cost function

$$J(t) = \int_0^t \Big(y^2(\tau) + 0.01u^2(\tau) \Big)\, d\tau.$$

A discrete-time LQG controller for the process can be designed using, e.g., the command `lqgdesign` from JITTERBUG.

For the implementation, the control task should execute in parallel with two other hard real-time tasks in a shared CPU. Fixed-priority preemptive scheduling is used, and it is assumed that the controller, Task 3, has the lowest priority. The task set is summarized in the table below:

Task	Priority	Period (T)	Deadline (D)	Exec. time (E)
1	High	0.080	0.080	0.027
2	Middle	0.140	0.140	0.044
3	Low	T_3	—	0.025

The control task period, T_3 is left as a design parameter. According to fixed-priority scheduling theory, assuming $D_i = T_i$, the three tasks are guaranteed to meet their deadlines if $U = \sum \frac{E_i}{T_i} \leq 0.78$. The shortest controller period that satisfies this condition is $T_3 = 0.195$. However, since a control task is not a hard real-time task, shorter periods than this could be considered. Depending on how the control task is implemented, a shorter period could mean that the idle CPU time is better utilized.

To investigate how the controller cost depends on the control task period, the fixed-priority scheduling algorithm is simulated using TRUETIME [**Hen+02**] and the timing results of Task 3 (the controller) are fed into JITTERTIME. Since TRUETIME is also Matlab-based, it is in fact possible to run the JITTERTIME analysis from within TRUETIME as a co-simulation. An example run with $T_3 = 0.195$ is shown in Figure 5. The schedule is completely deterministic but still creates a large jitter for Task 3 due to the preemption from the two higher-priority tasks.

To find the best controller period, T_3 is varied between 0.050 and 0.200 in steps of 0.001. In each case, the schedule and control loop are simulated for $T_{sim} = 1000$ time units, and the average controller cost is calculated. The results are reported in Figure 6. The smallest cost is achieved for $T = 0.075$, with a total utilization of $U = 0.985$. This is about 28% lower than the cost for $T = 0.195$

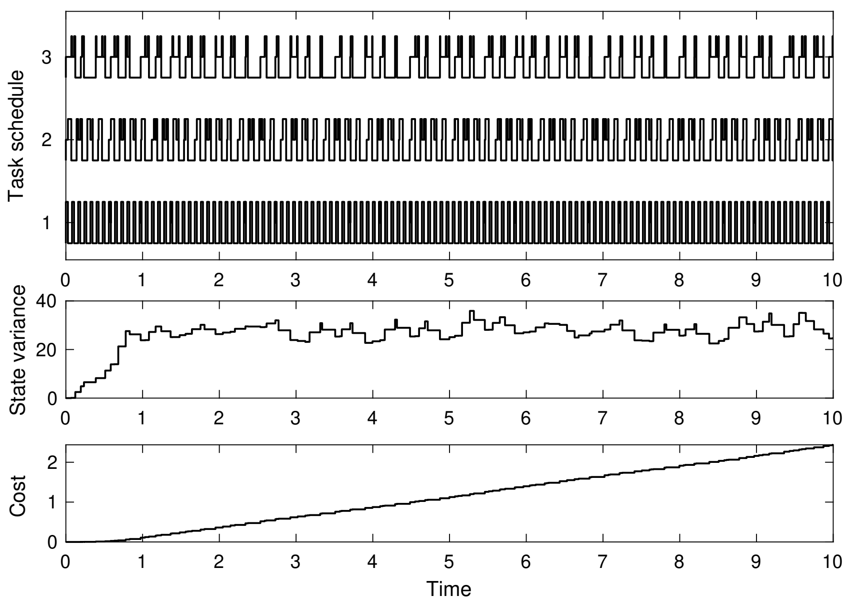

Figure 5 Example 3: TRUETIME schedule simulation (top) and JITTERTIME analysis (middle and bottom). The simulation shows that Task 3 (the controller) suffers from large jitter due to preemption.

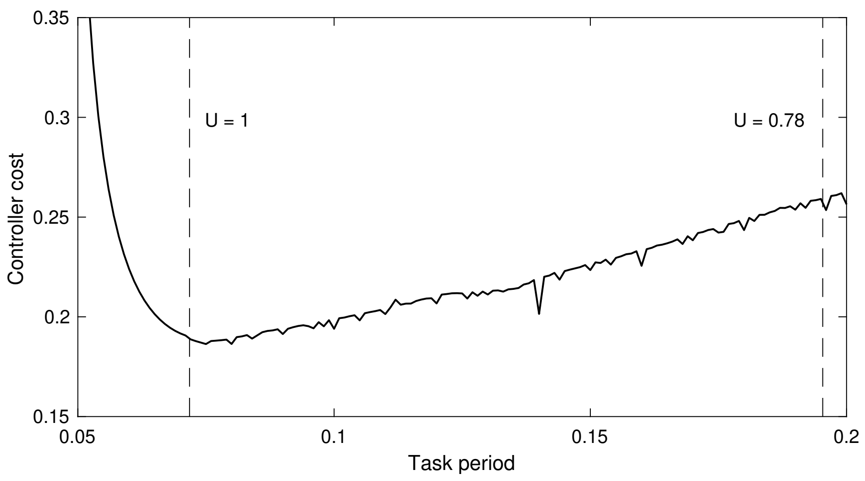

Figure 6 Example 3: The smallest controller cost is achieved for $T_3 = 0.075$, with a total utilization close to 1. This is about 28% smaller than the cost for $T_3 = 0.195$ (corresponding to $U = 0.78$).

(corresponding to $U = 1$). The curve looks a bit "noisy", but this is due to the nonlinearity of the scheduling algorithm: A small change in the period T_3 can create a completely different schedule, which in turn yields a different delay pattern (and hence cost) for the controller.

The code for the example consists of several different files for setting up the TRUETIME simulation and is not listed here.

Command Reference

JITTERTIME consists of the seven commands listed below, plus the `calcc2d` helper function from JITTERBUG.

jtAddContSys

Purpose

Add a continuous-time linear system to a JITTERTIME model.

Syntax

```
N = jtAddContSys(N,sysid,sys,inputid)
N = jtAddContSys(N,sysid,sys,inputid,Rc,Qc)
```

Description

The continuous-time system can be given in state-space, transfer-function, or zero-pole-gain form. In *state-space form*, the system is described by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + v_c(t) \\ y(t) &= Cx(t) \end{aligned}$$

where v_c is a continuous-time white-noise process with zero mean and covariance function

$$\mathrm{E}\, v_c(t) v_c^T(s) = R_c\, \delta(t-s)$$

The cost of the system is specified as

$$J = \int_0^{T_{sim}} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}^T Q_c \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt$$

where Q_c is a positive semi-definite matrix.

Arguments

`N`	The JITTERTIME model to add this continuous-time system to.
`sysid`	A unique positive ID number for this system (pick any). Used when referred to from other systems.
`sys`	A strictly proper, delay-free continuous-time LTI system in state-space or transfer function (or zpk) form. Internally, the system will be converted to state-space form.
`inputid`	A vector of input system IDs. The outputs of the corresponding systems will be used as inputs to this system. The number of inputs in this system must equal the total number of outputs in the input systems. An empty vector (or zero) indicates that the system inputs are unconnected.

Optional Arguments

`Rc`	State (`ss`) or input (`tf`, `zpk`) noise intensity matrix (default = 0).
`Qc`	Cost function weighting matrix (default = 0).

Return Values

`N`	The JITTERTIME model which must be passed to all other functions.

See Also

`jtAddResetDynamics`

jtAddDiscSys

Purpose

Add a discrete-time linear system to a JITTERTIME model.

Syntax

```
N = jtAddDiscSys(N,sysid,sys,inputid)
N = jtAddDiscSys(N,sysid,sys,inputid,Rd,Qc)
```

Description

The discrete-time system can be given in state-space, transfer-function, or zero-pole-gain form. In *state-space form*, the system is described by

$$x(t_k^+) = Ax(t_k) + Bu(t_k) + v_1(t_k)$$
$$y(t_k^+) = Cx(t_k) + Du(t_k) + v_2(t_k)$$

where $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ is a discrete-time white-noise process with zero mean and covariance

$$R_d = \mathrm{E}\, v(t_k) v^T(t_k)$$

Noise is added each time the system is executed. The cost of the system is specified *in continuous time* as

$$J = \int_0^{T_{sim}} \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}^T Q_c \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} dt$$

where Q_c is a positive semi-definite matrix. Note that both $x(t)$ and $y(t)$ are constant between system executions.

Arguments

`N`	The JITTERTIME model to add this system to.
`sysid`	A unique positive ID number for this system (pick any). Used when referred to from other systems.
`sys`	A discrete-time LTI system in state-space or transfer function (or zpk) form, or a double/matrix (interpreted as a static gain transfer function). Internally, the system is converted to state-space form, where the held outputs are treated as additional states.
`inputid`	A vector of input system IDs. The outputs of the corresponding systems will be used as inputs to this system. The number of inputs in this system must equal the total number of outputs in the input systems. An empty vector (or zero) indicates that the system inputs are unconnected.

Optional Arguments

`R`	State (`ss`) or input (`tf`, `zpk`) noise intensity matrix (default = 0).
`Qc`	Cost function weighting matrix (default = 0).

Return Values

`N`	The JITTERTIME model which must be passed to all other functions.

Remark

For multiple-version systems, `sys`, `inputid` and `R` can be given as cell arrays (which must all have the same length, or length 1). Only state-space systems are allowed for multiple-version systems.

`jtAddResetDynamics`

Purpose

Add reset dynamics to a previously defined continuous-time state-space system.

Syntax

```
N = jtAddResetDynamics(N,sysid,A)
N = jtAddResetDynamics(N,sysid,A,B,inputid)
```

Description

Reset dynamics is only supported for state-space systems. When `jtResetSys` is called at time t_k during simulation, the system state is momentarily updated as

$$x(t_k^+) = Ax(t_k) + Bu(t_k)$$

Arguments

`N`	The JITTERTIME model to add this system to.
`sysid`	The ID number of a previously defined continuous-time state-space system.
`A`	The state update matrix of the reset dynamics.

Optional Arguments

`B`	The input matrix for the reset dynamics (default = 0).
`inputid`	A vector of system IDs. The outputs of the corresponding systems will be used as inputs for the reset update. The number of columns in B must equal the total number of outputs in the input systems.

Return Values

`N`	The JITTERTIME model which must be passed to all other functions.

Remark

For multiple-version dynamics, `A`, `B` and `inputid` can be given as cell arrays (which must all have the same length, or length 1).

See Also

`jtAddContSys`, `jtResetSys`

jtBeginPeriodicAnalysis, jtEndPeriodicAnalysis

Purpose

Perform a periodic steady-state analysis of the mean and covariance of a JITTERTIME system.

Syntax

```
N = jtBeginPeriodicAnalysis(N)
  ...
N = jtEndPeriodicAnalysis(N)
```

Description

This pair of functions can be used to calculate the steady-state periodic behavior of an infinitely repeated timing scenario. Between `jtBeginPeriodicAnalysis` and `jtEndPeriodicAnalysis`, any number of calls to the simulation commands `jtPassTime`, `jtExecSys`, `jtResetSys` and `jtStateDisturbance` can be performed. The total effect of these commands on the mean and covariance of the system is stored in the internal variables `N.Atot`, `N.dtot` and `N.Rtot`. Finding a steady-state periodic solution means finding m_p and P_p such that

$$\begin{aligned} m_p &= A_{tot} m_p + d_{tot} \\ P_p &= A_{tot} P_p A_{tot}^T + R_{tot} \end{aligned}$$

The calculation is only possible to perform if the total periodic dynamics A_{tot} is asymptotically stable, i.e., if it has all eigenvalues within the unit circle. If successful, the result is stored in the variables `N.mperiodic` and `N.Pperiodic`.

`jtCalcDynamics`

Purpose

Calculate the total dynamics of a JITTERTIME system before simulation can start.

Syntax

```
N = jtCalcDynamics(N)
```

Description

This function should be called once all systems have been added to the model, before the simulation is started. It checks the interconnection of all subsystems and merges all the systems into a large state space. Afterwards, the total continuous dynamics, noise, and cost are described by the large matrices `N.Ac`, `N.Rc`, and `N.Qc`. For each discrete system `N.sys{k}`, a large discrete transition matrix `N.Ad` and a large discrete noise matrix `N.Rd` are calculated. Finally, the initial state covariance `N.P` and state mean value `N.m` are set to zero.

Return Values

`N`	The JITTERTIME model which must be passed to all other functions.

jtExecSys

Purpose

Simulate the execution of a discrete-time system.

Syntax

```
N = jtExecSys(N,sysid)
N = jtExecSys(N,sysid,ver)
```

Description

Execute the discrete-time system with ID `sysid`, simulating the sampling of its inputs, calculation of a new state, and updating of its outputs. For multiple-version discrete systems, the version number `ver` can be specified (default is 1). Internally, the following is executed, updating the mean value and the covariance of the total system:

```
N.m = N.sys{sysid}.Ad{ver} * N.m;
N.P = N.sys{sysid}.Ad{ver} * N.P * N.sys{sysid}.Ad{ver}' + N.sys{sysid}.Rd{ver};
```

The state update matrices `Ad` and discrete noise matrices `Rd` have already been calculated in `jtCalcDynamics`.

Return Values

`N`	The JITTERTIME model which must be passed to all other functions.

jtInit

Purpose

Initialize a new JITTERTIME model.

Syntax

```
N = jtInit()
```

Description

Initialize a new JITTERTIME model, creating a struct N for holding all system and simulation data. Internally, the following variables are set to zero:

```
N.P = 0;
N.m = 0;
N.J = 0;
N.Tsim = 0;
```

Return Values

N	The JITTERTIME model, which must be passed to all other functions.

jtPassTime, jtPassTimeUntil

Purpose

Simulate the passing of time, integrating the dynamics of all continuous-time systems.

Syntax

```
N = jtPassTime(N,T)
N = jtPassTimeUntil(N,time)
```

Description

All continuous-time systems are simulated, evolving their state covariance and mean value, and accumulating cost. Internally, the continuous system matrices, noise and cost are sampled using `T` as the timestep. The following is calculated in `jtPassTime`:

```
[A,R,Q,Qconst] = calcc2d(N.Ac,N.Rc,N.Qc,T);  % Sample the model
N.J = N.J + trace(Q * N.P) + Qconst;         % Acc. cost due to covariance
N.J = N.J + N.m' * Q * N.m;                  % Acc. cost due to mean value
N.P = A * N.P * A' + R;                      % Evolve state covariance
N.m = A * N.m;                               % Evolve state mean value
N.Tsim = N.Tsim + T;                         % Increase clock
```

In `jtPassTimeUntil`, `T` is first calculated as

```
T = time - N.Tsim;
```

Return Values

`N`	The JITTERTIME model, which must be passed to all other functions.

jtResetSys

Purpose

Execute the reset dynamics of a continuous-time system.

Syntax

```
N = jtResetSys(N,sysid)
N = jtResetSys(N,sysid,ver)
```

Description

When called, the continuous system state of system `sysid` is momentarily updated as

$$x(t_k^+) = Ax(t_k) + Bu(t_k)$$

For multiple-version discrete systems, the reset dynamics number `ver` can be specified (default is 1). Internally, the reset works like a discrete-time system update, except that no noise is added. The following code is executed to update the total system mean value and covariance:

```
  N.m = N.sys{sysid}.Ad{ver} * N.m;
  N.P = N.sys{sysid}.Ad{ver} * N.P * N.sys{sysid}.Ad{ver}';
```

The state update matrices `Ad` have already been calculated in `jtCalcDynamics`.

Arguments

`N`	The JitterTime model.
`sysid`	The ID number of continuous-time system with reset dynamics.

Optional Arguments

`B`	The input matrix for the reset dynamics (default = 0).
`inputid`	A vector of system IDs. The outputs of the corresponding systems will be used as inputs for the reset update. The number of columns in B must equal the total number of outputs in the input systems.

Return Values

`N`	The JitterTime model, which must be passed to all other functions.

See Also

`jtAddContSys`, `jtAddResetDynamics`

jtStateDisturbance

Purpose

Add a disturbance with given mean and variance to the state of a system during the simulation.

Syntax

```
N = jtStateDisturbance(N,sysid,m,P)
```

Description

The command can be called at any time after `jtCalcDynamics` to simulate an impulse disturbance acting on the state (for continuous-time state-space systems) or on the state and output (for discrete-time state-space systems).

Arguments

`N`	The JitterTime model.
`sysid`	A system ID.
`m`	The mean value of the disturbance vector. Must be of size $n \times 1$ for continuous systems and of size $(n+p) \times 1$ for discrete systems, where n is the number of states and p is the number of outputs.

Optional Arguments

`P`	The discrete variance of the disturbance vector (default = 0). Must be of size $n \times n$ for continuous systems and of size $(n+p) \times (n+p)$ for discrete systems, where n is the number of states and p is the number of outputs.

Return Values

`N`	The JitterTime model which must be passed to all other functions.

Remark

Since the system is linear, the stochastic and deterministic disturbances can be analyzed independently from each other.