LUND UNIVERSITY

Jitterbug - Reference Manual

Cervin, Anton; Lincoln, Bo

2003

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Cervin, A., & Lincoln, B. (2003). *Jitterbug - Reference Manual*. (Technical Reports TFRT-7604). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
2

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

ISSN 0280–5316
ISRN LUTFD2/TFRT--7604--SE

Jitterbug Reference Manual

Anton Cervin
Bo Lincoln

Department of Automatic Control
Lund Institute of Technology
January 2003

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
January 2003

Document Number
ISRN LUTFD2/TFRT--7604--SE

Author(s)
Anton Cervin, Bo Lincoln

Supervisor

Sponsoring organisation

Title and subtitle
Jitterbug Reference Manual

Abstract

The manual describes the use of Jitterbug, a Matlab toolbox for real-time control performance analysis. The tool facilitates the computation of a quadratic performance index for a linear control system under various timing conditions.

Key words

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
37

Security classification

Recipient's notes

The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden
Fax +46 46 222 44 22 E-mail ub2@ub2.lu.se

Contents

1. Introduction 2

2. System Description 2

2.1 Signal Model 2

2.2 Timing Model 4

3. Internal Workings 6

3.1 Sampling the System 6

3.2 Timing Representation 7

3.3 Calculating Variance and Cost 8

3.4 Calculating Spectral Densities 9

4. Examples 9

4.1 Distributed Control System 9

4.2 Notch Filter 13

4.3 Multirate Controller 16

4.4 Spectral Density Calculation 19

4.5 Overrun Handling Methods 21

5. Command Reference 24

initjitterbug 25

addtimingnode 26

addcontsys 27

adddiscsys 29

adddiscexec 31

adddisctimedep 32

calcdynamics 33

calccost 34

lqgdesign 35

6. References 37

1. Introduction

JITTERBUG [Lincoln and Cervin, 2002] is a MATLAB-based toolbox that allows the computation of a quadratic performance criterion for a linear control system under various timing conditions. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. As an additional feature, it is also possible to compute the spectral density of the signals in the control system. The main contribution of the toolbox, which is built on well-known theory (LQG theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

2. System Description

In JITTERBUG, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a JITTERBUG model is shown in Figure 1, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

2.1 Signal Model

The signal model consists of a number of inter-connected continuous-time and discrete-time linear systems driven by white noise. The cost is specified as a stationary, continuous-time quadratic cost function.

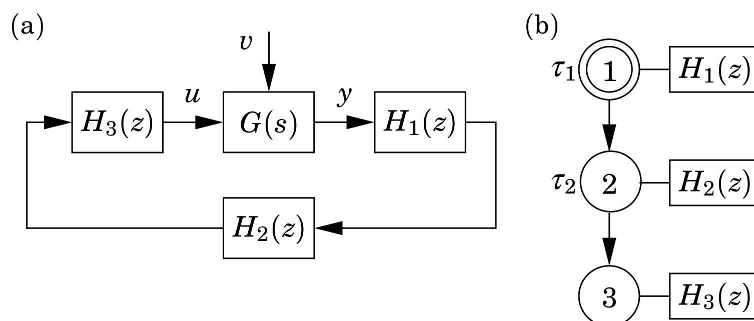

Figure 1 A simple JITTERBUG model of a computer-controlled system: (a) signal model and (b) timing model.

Continuous-Time Systems A continuous-time system is described by

$$
\begin{aligned}
\dot{x}_c(t) &= Ax_c(t) + Bu(t) + v_c(t) \\
y^0(t) &= Cx_c(t) && \text{(continuous output)} \\
y(t_k) &= y^0(t_k) + e_d(t_k) && \text{(measured discrete output)}
\end{aligned}
$$

where A, B, and C are constant matrices, and v_c is a continuous-time white-noise process with zero mean and covariance matrix R_1 (strictly speaking, v_c has the spectral density $\phi(\omega) = \frac{1}{2\pi}R_1$), and e_d is a discrete-time white-noise process with zero mean and covariance matrix R_2. Note that direct terms are not allowed (i.e., the system must be strictly proper). Also note that there is no *continuous-time* output noise. The ability to specify discrete-time measurement noise in connection with the plant is only offered as a convenience. The discrete-time output noise will be translated to input noise at any connected discrete-time system(s).

The cost of the system is specified as

$$
J_c = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix}^T Q_c \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix} dt
$$

where Q_c is a positive semi-definite matrix.

The system may also be specified in transfer-function form (see the description of `addcontsys` on page 27).

Discrete-Time Systems A discrete-time system is described by

$$
\begin{aligned}
x_d(t_{k+1}) &= \Phi x_d(t_k) + \Gamma u(t_k) + v_d(t_k) \\
y^0(t_k) &= Cx_d(t_k) + Du(t_k) && \text{(discrete output)} \\
y(t_k) &= y^0(t_k) + e_d(t_k) && \text{(measured discrete output)}
\end{aligned}
$$

where Φ, Γ, C, and D are possibly time-varying matrices (see below). The covariance of the discrete-time white noise processes v_d and e_d is given by

$$
R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T .
$$

The update instants t_k are determined by the timing model and are not necessarily equidistant in time. The input signal u is sampled when the system is updated, and the state x_d and the output signal y^0 are held between updates.

The cost of the system is specified as

$$
J_d = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_d(t) \\ y^0(t) \\ u(t) \end{pmatrix}^T Q_d \begin{pmatrix} x_d(t) \\ y^0(t) \\ u(t) \end{pmatrix} dt
$$

where Q_d is a positive semi-definite matrix. Note that $x_d(t)$ and $y^0(t)$ are piecewise constant signals, while $u(t)$ may be a continuous signal.

The system may also be specified in transfer-function form (see the description of `adddiscsys` on page 29).

Connecting Systems The total system is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. Throughout, MIMO formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous-time and discrete-time systems:

$$J = \sum J_c + \sum J_d \tag{1}$$

It's important to understand how cost and noise are handled when systems are interconnected. Three principal cases can be distinguished (see Figure 2):

(a) The interconnection of two continuous-time systems. Note that any discrete-time output noise e_d will be ignored.

(b) The interconnection of two discrete-time systems. No surprises here.

(c) The interconnection of a continuous-time and a discrete-time system. Note that the discrete-time output noise e_d will not be included in the input cost of the discrete-time system.

2.2 Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can also be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers and also greatly simplifies the cost calculations.

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = [\,P_\tau(0) \quad P_\tau(1) \quad P_\tau(2) \quad \ldots\,],$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain δ is a constant that is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period.

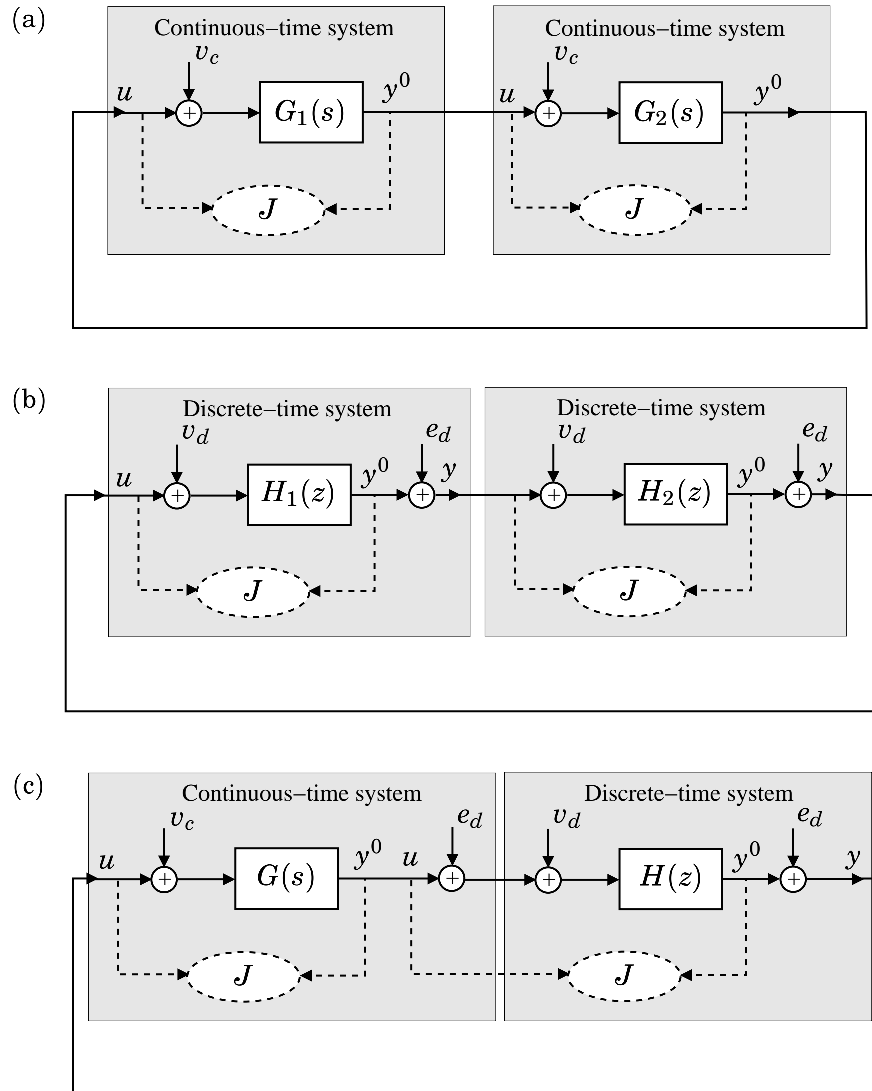

Figure 2 Possible interconnections of continuous-time and discrete-time systems.

Node- and Time-Dependent Execution The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers for example.

Alternative Execution Paths For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In JITTERBUG, two such cases can be modeled (see Figure 3):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with proba-

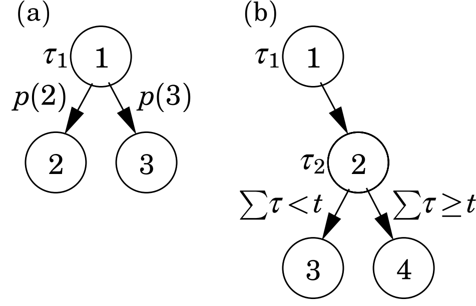

Figure 3 Alternative execution paths in a JITTERBUG execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

bility $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a timevector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

3. Internal Workings

Inside JITTERBUG, the states and the cost are considered in continuous time. The inherently discrete-time states, e.g. in discrete-time controllers or filters, are treated as continuous-time states with zero dynamics. This means that the total system can be written as

$$\dot{x}(t) = Ax(t) + w(t) \tag{2}$$

where x collects all the states in the system, and w is continuous-time white noise process with covariance $\widetilde{R}$. To model the discrete-time changes of some states as a timing node n is activated, the state is instantaneously transformed by

$$x(t^+) = E_n x(t) + e_n(t)$$

where e_n is a discrete-time white noise process with covariance W_n.

The total cost (1) for the system can be written as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T x^T(t)\widetilde{Q}x(t)\,dt \tag{3}$$

where $\widetilde{Q}$ is a positive semidefinite matrix.

3.1 Sampling the System

JITTERBUG relies on discretized time to calculate the variance of the states and the cost. No approximations are involved, however. Sampling the system (2) with a period of δ (the time-grain in the delay distributions) gives

$$x(k\delta + \delta) = \Phi x(k\delta) + v(k\delta) \tag{4}$$

where the covariance of v is R, and the cost (3) becomes

$$J = \lim_{N\to\infty} \frac{1}{N\delta} \sum_{k=0}^{N-1} \Big(x^T(k\delta) Q x(k\delta) + q \Big)$$

The matrices Φ, R, Q, and q are calculated as

$$\begin{aligned}
\Phi &= e^{A\delta} \\
R &= \int_0^{\delta} e^{A(\delta-\tau)} \widetilde{R} e^{A^T(\delta-\tau)}\, d\tau \\
Q &= \int_0^{\delta} e^{A^T t} \widetilde{Q} e^{At}\, dt \\
q &= \mathrm{tr}\Big(\widetilde{Q} \int_0^{\delta} \int_0^{\delta} e^{A(t-\tau)} \widetilde{R} e^{A^T(t-\tau)}\, d\tau\, dt \Big)
\end{aligned}$$

or, equivalently, from

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} = \exp\left(\begin{pmatrix} -A^T & \widetilde{Q} \\ 0 & A \end{pmatrix} \delta \right)$$

and

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} = \exp\left(\begin{pmatrix} -A & I & 0 \\ 0 & -A & \widetilde{R}^T \\ 0 & 0 & A^T \end{pmatrix} \delta \right)$$

so that

$$\begin{aligned}
\Phi &= P_{22} \\
Q &= P_{22}^T P_{12} \\
R &= M_{33}^T M_{23} \\
q &= \mathrm{tr}\big(Q M_{33}^T M_{13} \big)
\end{aligned}$$

3.2 Timing Representation

As time is discretized, we can transform the system description into a jump linear system, where the Markov state represents the current timing state of the system. Each timing node is represented by one Markov node. In between timing nodes additional Markov nodes representing the delay are inserted as illustrated in Figure 4.

Consider following one path in the Markov chain. For each node which is not a timing node, only the continuous states of the system change. In each time-step, they evolve as in (4), and thus the state covariance $P(k\delta) = \mathbf{E}\left\{ x(k\delta) x^T(k\delta) \right\}$ evolves as

$$P(k\delta + \delta) = \Phi P(k\delta) \Phi^T + R$$

At each timing node n, the system is additionally transformed as in (3),

$$P(k\delta^+) = E_n P(k\delta) E_n^T + W_n$$

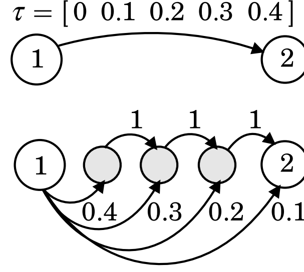

Figure 4 A random delay (above) modeled as a jump linear system (below), where the delay is represented by additional Markov nodes in between the timing nodes.

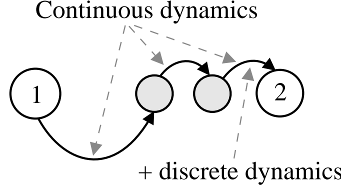

Figure 5 The continuous-time dynamics is active between all Markov nodes, whereas the discrete-time dynamics is activated only before a timing node.

where W_n is the covariance of the discrete-time noise $e_n(k\delta)$ in node n. See Figure 5 for an illustration. Combining the above, we define Φ_n as

$$\Phi_n = \begin{cases} \Phi & \text{if } n \text{ is not a timing node} \\ E_n\Phi & \text{if } n \text{ is a timing node} \end{cases}$$

and similarly R_n as

$$R_n = \begin{cases} R & \text{if } n \text{ is not a timing node} \\ E_n R E_n^T + W_n & \text{if } n \text{ is a timing node} \end{cases}$$

3.3 Calculating Variance and Cost

Now consider all possible Markov states simultaneously. Let $\pi_n(k\delta)$ be the probability of being in Markov state n at time $k\delta$, and let $P_n(k\delta)$ be the covariance of the state if the system is in Markov state n at time $k\delta$. Furthermore, let the transition matrix of the Markov chain be σ, such that

$$\pi(k\delta + \delta) = \sigma\pi(k\delta)$$

The state covariance then evolves as

$$P_n(k\delta + \delta) = \sum_i \sigma_{ni}\pi_i(k\delta)\Big(\Phi_n P_i(k\delta)\Phi_n^T + R_n\Big) \tag{5}$$

and the immediate cost at time $k\delta$ is calculated as

$$\frac{1}{\delta}\sum_n \pi_n(k\delta)\Big(\mathrm{tr}\big(P_n(k\delta)Q\big) + q\Big)$$

For systems without a periodic node, equation (5) must be iterated until the cost and variance converge. For periodic systems, the Markov state always

returns to the periodic timing node every h/δ time steps. As equation (5) is affine in P, we can find the stationary covariance $P_1(\infty)$ in the periodic node by solving a linear system of equations. The total cost is then calculated over the timesteps in one period. The toolbox returns the cost $J = \infty$ if the system is not mean-square stable.

3.4 Calculating Spectral Densities

For periodic systems, the toolbox also computes the discrete-time spectral densities of all outputs as observed in the periodic timing node. The spectral density of an output y is defined as

$$\phi_y(\omega) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} r_y(k) e^{-ik\omega}$$

The covariance function $r_y(k)$ is given by

$$\begin{aligned} r_y(k) &= \mathbf{E}\left\{ y(t)y^T(t+kh) \right\} = \mathbf{E}\left\{ Cx(t)x^T(t+kh)C^T \right\} \\ &= \mathbf{E}\left\{ C\bar{\Phi}^{|k|}x(t)x^T(t)C^T \right\} = C\bar{\Phi}^{|k|}P_1(\infty)C^T \end{aligned}$$

where $\bar{\Phi}$ is the average transition matrix over a period, and $P_1(\infty)$ is the stationary covariance in the periodic node. The spectral density is returned as a discrete-time linear system $F(z)$ such that $\phi_y(\omega) = F(e^{i\omega})$.

4. Examples

In this section, various examples that illustrate the use of JITTERBUG are given.

4.1 Distributed Control System

In the example, we will study the distributed control system shown in Figure 6. The setup is taken from [Nilsson, 1998]. In the control loop, the sen-

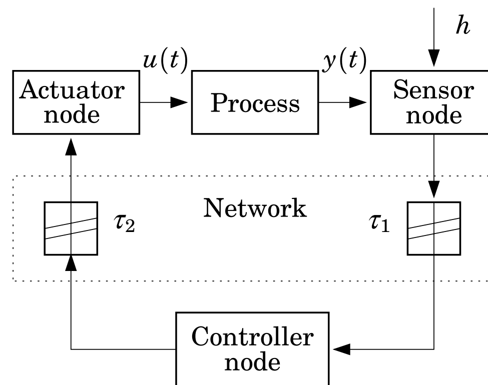

Figure 6 Distributed control system with communication delays τ_1 and τ_2.

sor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time-driven, whereas the controller and actuator nodes are assumed to be event-driven. At a fixed

period h, the sensor samples the process and sends the measurement sample over the network to the controller node. There the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated.

The JITTERBUG model of the system was shown in Figure 1 on page 2. The DC servo process is given by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}.$$

The process is driven by white continuous-time input noise. There is assumed to be no measurement noise.

The process is sampled periodically with the interval h. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

and the discrete-time PD controller is implemented as

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right),$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$. (A real implementation would include a low-pass filter in the derivative part, but that is ignored here.)

The delays in the computer system are modeled by the two random variables τ_1 and τ_2. The total delay from sampling to actuation is given by $\tau_{tot} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise JITTERBUG would skip the remaining updates).

As a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T\to\infty} \frac{1}{T}\int_0^T \left(y^2(t) + u^2(t)\right) dt. \tag{6}$$

Sampling Period and Constant Delay A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control, rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that

$$0.2 < \omega_b h < 0.6,$$

where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of computational delay is typically not considered in such rules of thumb, however. Using JITTERBUG, the combined effect of sampling period and computational delay can be easily investigated. In Figure 7, the cost function (6) for the networked control system has been

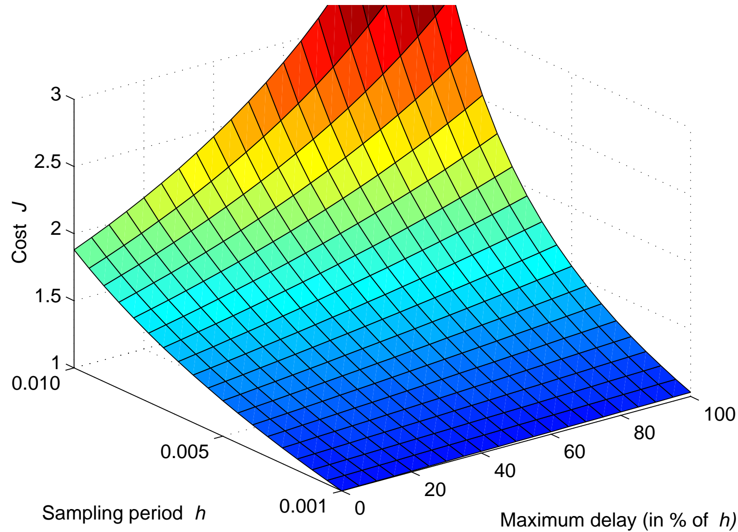

Figure 7 The cost as a function of sampling period and constant delay in the distributed control system example.

evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant total delay ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at run-time will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since JITTERBUG allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the JITTERBUG model of the networked control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $\tau_{max}/2$, where τ_{max} denotes the maximum round-trip delay in the loop. A range of PD controller parameters (ranging from $K = 1.5$ and $T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $\tau_{max}/4$.

In Figure 8, the cost function (6) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for maximum total delay ranging from 0 to 100% of the sampling interval. Compared to Fig 7, the cost is considerably lower.

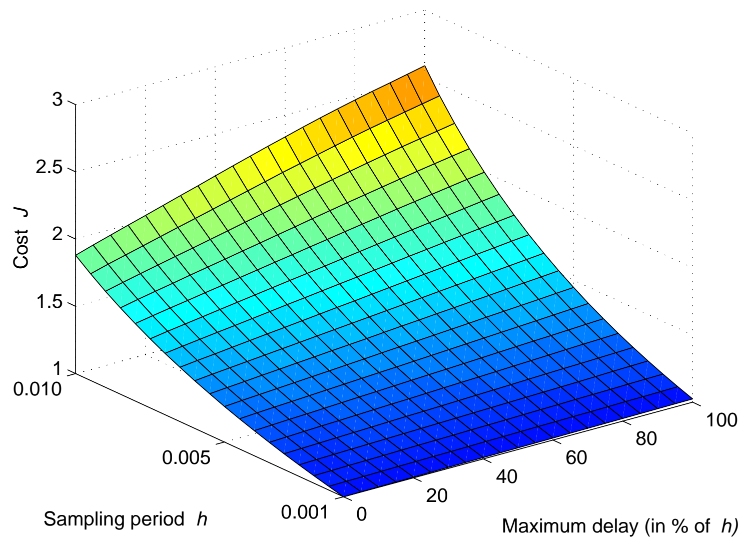

Figure 8 The cost as a function of sampling period and maximum delay with jitter compensation in the distributed control system example.

The Matlab script for the computations is given below:

```
% Jitterbug example: distributed.m
% ================================
% Calculate the performance of a distributed control system with
% delays/jitter

scenario = 1;  % 1 = constant delay, 2 = random delay,
               % 3 = random delay + jitter compensation
s = tf('s');
G = 1000/(s^2+s);  % The process
R1 = 1;            % Input noise
R2 = 0;            % Output noise
Q = diag([1 1]);   % J = E(y^2 + u^2)

% Default PD parameters
K = 1.5;
Td = 0.035;

% Gain(delay)-scheduled PD parameters
tauv = [0 0.0035 0.0045 0.0055 0.0065 0.0075];
Kv = [1.5 1.2 1.1 0.98 0.86 0.78];
Tdv = [0.035 0.04 0.042 0.046 0.049 0.052];

hvec = 0.001:0.0005:0.010;
Jmat = [];
for h = hvec
  dt = h/40;
  taumaxvec = 0:2*dt:h;
  for taumax=taumaxvec
    Ptau = zeros(1,round(h/dt)+1);
    if scenario == 1
      Ptau(round(taumax/2/dt)+1) = 1;   % constant delay
    else
      Ptau(1:round(taumax/2/dt)+1) = 1; % random delay
```

```
    end
    Ptau = Ptau/sum(Ptau);

    H1 = 1;                              % Sampler
    H2 = ss(0,1,K*Td/h,-K*(Td/h+1),-1); % Controller
    H3 = 1;                              % Actuator

    N = initjitterbug(dt,h);          % Initialize Jitterbug

    N = addtimingnode(N,1,Ptau,2);    % Add node 1
    N = addtimingnode(N,2,Ptau,3);    % Add node 2
    N = addtimingnode(N,3);           % Add node 3

    N = addcontsys(N,1,G,4,Q,R1,R2);  % Add sys 1 (G)
    N = adddiscsys(N,2,H1,1,1);       % Add sys 2 (H1) to node 1
    N = adddiscsys(N,3,H2,2,2);       % Add sys 3 (H2) to node 2
    N = adddiscsys(N,4,H3,3,3);       % Add sys 4 (H3) to node 3

    if scenario == 3  % jitter compensation
      for k=1:round(taumax/2/dt)
        tau1 = dt*k;       % known delay
        tau2 = taumax/4;   % predicted remaining delay
        t = tau1 + tau2;
        Kt = interp1(tauv,Kv,t,'linear','extrap');
        Tdt = interp1(tauv,Tdv,t,'linear','extrap');
        H2 = ss(0,1,Kt*Tdt/h,-Kt*(Tdt/h+1),-1);
        N = adddisctimedep(N,3,H2,k);  % Make sys 3 (H2) time-dependent
      end
    end

    N = calcdynamics(N);       % Calculate the internal dynamics
    J = calccost(N)            % Calculate the cost
    Jmat(find(h==hvec),find(taumax==taumaxvec)) = J;
  end
end

Jmat=Jmat/Jmat(1,1);  % scale plot to 1 in (0,0)
figure
surf(0:5:100,hvec,Jmat)
axis([0 100 hvec(1) hvec(end) 1 3])
caxis([0.7 3])
xlabel('Maximum Delay (in % of h)')
ylabel('Sampling Period h')
zlabel('Cost J')
```

4.2 Notch Filter

Cleaning signals from disturbances using e.g. notch filters is important in many applications. In some cases these filters are very sensitive to lost samples due to the very narrow-band characteristics, and in real-time systems lost samples are sometimes inevitable. In this example JITTERBUG is used to evaluate the effects of this problem on different filters.

The setup is as follows. A good signal x (modeled as low-pass filtered noise) is to be cleaned from an additive disturbance e (modeled as band-pass filtered noise). An estimate $\hat{x}$ of the good signal should be found by applying

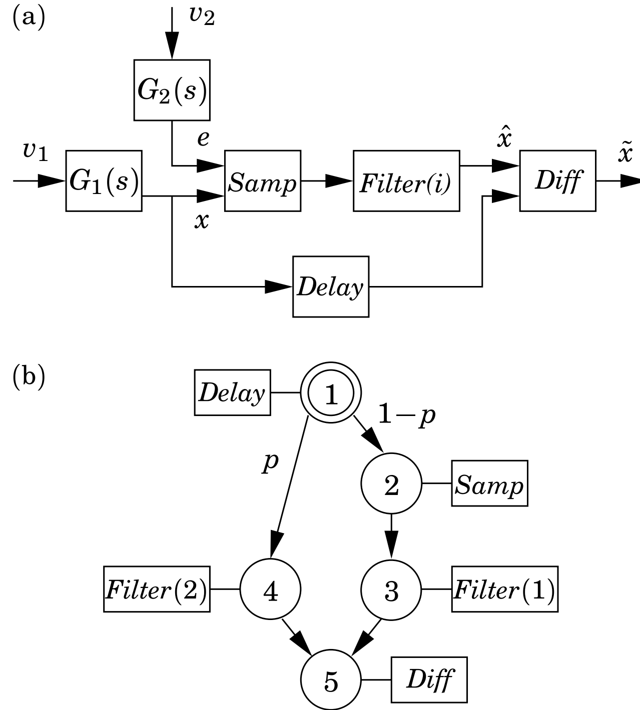

Figure 9 JITTERBUG model of the notch filter: (a) signal model and (b) timing model.

a digital notch filter with the sampling interval $h = 0.1$ to the measured signal $x + e$. Unfortunately, a fraction p of the measurements are lost.

A JITTERBUG model of the system is shown in Figure 9. The signals x and e are generated by white noise being filtered through the continuous-time systems G_1 and G_2. The digital filter is represented as two discrete-time systems: *Samp* and *Filter*. The good signal is buffered in the system *Delay* and is compared to the filtered estimate in the system *Diff*. In the timing model, there is a probability p that the *Samp* system will not be updated. In that case, it is possible to execute an alternate version, *Filter*(2), of the filter dynamics.

Two different filters are compared. The first filter is an ordinary second-order notch filter with two zeros on the unit circle. The same update equations are used regardless if a sample is available or not. The second filter is a second-order Kalman filter based on a simple model of the signal dynamics. In the case of lost samples, only prediction is performed in the filter.

The spectral density of the estimation error $\tilde{x} = x - \hat{x}$ in the two filter cases is shown in Figure 10. It has been assumed that $p = 10\%$ of the samples are lost. It is seen that the ordinary notch filter performs well around the disturbance frequency while the lost samples introduce a large error at lower frequencies. The time-varying Kalman filter is less sensitive towards lost samples and has a more even error spectrum. Overall, the variance of the estimation error is about 40% lower in the Kalman filter case.

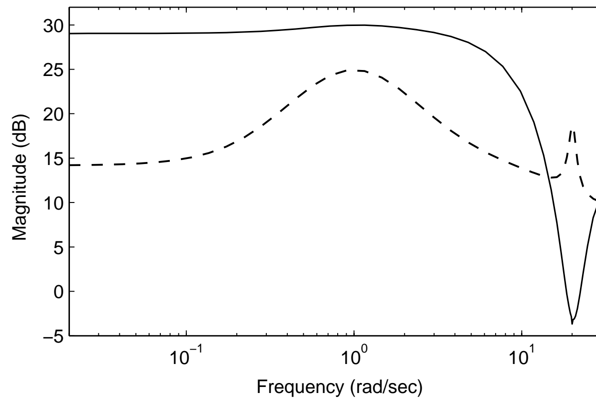

Figure 10 The spectral density of the error output $\tilde{x}$ when 10% of the samples are lost, using a notch filter (full) or a time-varying Kalman filter (dashed).

The Matlab script for the computations is given below:

```
% Jitterbug example: notch.m
% ========================
% Calculate the performance of a notch filter with lost samples.

scenario = 1; % 1=no filter, 2=notch filter, 3=Kalman filter
p = 0.1;      % Probability of lost sample

s = tf('s');
z = tf('z');

h = 0.1;       % Sampling period

% System generating the good signal
G1 = 100/(s+1)^2;
R1 = 2*pi;     % Input noise variance

% System generating the disturbance
omega = 20;    % Resonance frequency
zeta = 0.001;  % Damping
G2 = 50/(s^2+2*zeta*omega*s+omega^2);
R2 = 2*pi;     % Input noise variance

Samp = [1 1];  % Discrete-time system that samples x + e
Diff = [1 -1]; % Discrete-time system that computes x - xhat

switch scenario,
 case 1,
  % No filter
  Filter1 = 1;
  Filter2 = []; % same dynamics (i.e., none)
  Delay = 1;

 case 2,
  % Zero-phase notch filter
  a = -0.5/cos(omega*h);
  Filter1 = ss(tf([a 1 a],[1 0 0],h));
  Filter1 = Filter1/dcgain(Filter1);
```

```
  Filter2 = []; % same dynamics
  Delay = 1/z;  % The notch filter has a delay of one sample

 case 3,
  % Kalman filter based on simple model of G1 (integrator)
  [a1,g1,c1] = ssdata(ss(-0.00001,15,1,0));
  [a2,g2,c2] = ssdata(G2);
  a = blkdiag(a1,a2);
  g = eye(size(a,1));
  c = [c1 c2];
  r1 = blkdiag(g1*g1',g2*g2');
  r2 = 0;
  phi = ssdata(c2d(ss(a,g,c,0),h));
  kf = lqed(a,g,c,r1/h,r2,h);
  k = phi*kf;
  phio = (phi-k*c);
  gammao = k;
  co = [c1 0*c2]*(eye(size(a,1))-kf*c);
  do = [c1 0*c2]*kf;
  Filter1 = ss(phio,gammao,co,do,h);         % Prediction and correction
  Filter2 = ss(phi,zeros(size(a,1),1),[c1 0*c2],0,-1); % Prediction only
  Delay = 1;
end

delta = h;          % Time-grain = sampling interval
Ptau = [1];         % Zero delay between timing nodes
Q = diag([1 0 0]);  % J = xtilde^2

N = initjitterbug(delta,h);       % Initialize Jitterbug

N = addtimingnode(N,1,Ptau,[2 4],[1-p p]);  % Add node 1
N = addtimingnode(N,2,Ptau,3);              % Add node 2
N = addtimingnode(N,3,Ptau,5);              % Add node 3
N = addtimingnode(N,4,Ptau,5);              % Add node 4
N = addtimingnode(N,5);                     % Add node 5

N = addcontsys(N,1,G1,0,[],R1);      % Add sys 1 (G1)
N = addcontsys(N,2,G2,0,[],R2);      % Add sys 2 (G2)
N = adddiscsys(N,3,Samp,[1 2],2);    % Add sys 3 (Samp) to node 2
N = adddiscsys(N,4,Filter1,3,3);     % Add sys 4 (Filter) to node 3
N = adddiscexec(N,4,Filter2,3,4);    % Add execution of sys 4 to node 4
N = adddiscsys(N,5,Delay,1,1);       % Add sys 5 (Delay) to node 1
N = adddiscsys(N,6,Diff,[5 4],5,Q);  % Add sys 6 (Diff) to node 5

N = calcdynamics(N);    % Calculate internal dynamics
[J,P,F] = calccost(N);  % Calculate cost and spectral densities
J

figure
bodemag(F{1},F{2},F{4},F{6})  % Plot spectra of outputs 1,2,4,6
axis([0.1 pi/h -20 100]);
legend('Good Signal','Disturbance','Filter Output','Error');
title('Spectral Densities')
```

4.3 Multirate Controller

In this example we will show how to compute the performance of a multi-

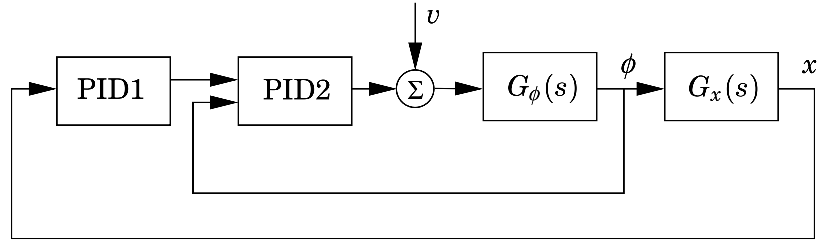

Figure 11 The ball & beam cascaded controller.

rate controller. This is illustrated on a cascade controller for the ball and beam process, see Figure 11. In this control structure, the inner controller, PID2, is responsible for controlling the beam dynamics,

$$G_\phi(s) = \frac{4.4}{s},$$

while the outer controller, PID1, controls the ball on beam dynamics,

$$G_x(s) = -\frac{9.0}{s^2}.$$

Since the inner loop is typically designed to be much faster than the outer loop, it can be a good idea to execute the inner loop at a higher frequency, especially if CPU resources are scarce. We will compare the performance of an ordinary cascade controller with a multirate cascade controller where the inner controller executes at twice the frequency of the outer controller.

The JITTERBUG timing model in the multirate case is shown in Figure 12. The sampling interval of the outer controller is denoted h. The sampling interval of the inner controller is thus $h/2$. The execution time of the control algorithm is ignored in this simple model. At the beginning of each period, PID1 is executed, immediately followed by PID2, which uses the control signal produced by PID1 as a reference value. Then, half a period later, the PID2 is executed again, using the same reference value as in the first invocation but a new measurement value.

Assume that the process is disturbed by white input noise v with unit variance, and that the performance is measured by the cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(\phi^2(t) + x^2(t)\right) dt$$

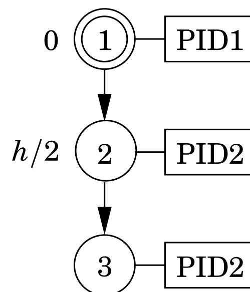

Figure 12 Timing model for the multirate ball & beam controller.

Assuming certain PID parameters, the performance in the different cases becomes

$$J_{ordinary} = 3.40, \qquad J_{multirate} = 1.99.$$

(Running both controllers at the fast rate gives $J = 1.93$, i.e. only a small further improvement.)

The Matlab script comparing the two cases is shown below:

```
% Jitterbug example: multirate.m
% ==============================
% Calculate the performance of ordinary/multirate ball & beam controller

s = tf('s');

Gphi = 4.4/s;
Gx = -9.0/s^2;

Q = diag([1 0]);
R1 = 1;
R2 = 0;

h = 0.1;
delta = h/2;

K1 = -0.2;
Ti = 10;
Td = 1;
N = 10;
PID1c = -K1*(1+1/Ti/s+s*Td/(1+s*Td/N));  % PID controller
PID1 = c2d(PID1c,h,'matched');

K2 = 4;
PID2 = K2*[1 -1];                        % P controller

%% Case 1: ordinary cascade controller
N = initjitterbug(delta,h);
N = addtimingnode(N,1,[1],2);       % Add node 1
N = addtimingnode(N,2);             % Add node 2
N = addcontsys(N,1,Gphi,4,Q,R1);    % Add sys 1 (Gphi)
N = addcontsys(N,2,Gx,1,Q);         % Add sys 2 (Gx)
N = adddiscsys(N,3,PID1,2,1);       % Add sys 3 (PID1) to node 1
N = adddiscsys(N,4,PID2,[3 1],2);   % Add sys 4 (PID2) to node 2
N = calcdynamics(N);                % Calculate internal dynamics
J = calccost(N)                     % Calculate cost

%% Case 2: multirate cascade controller
N = initjitterbug(delta,h);
N = addtimingnode(N,1,[1],2);       % Add node 1
N = addtimingnode(N,2,[0 1],3);     % Add node 2
N = addtimingnode(N,3);             % Add node 3
N = addcontsys(N,1,Gphi,4,Q,R1);    % Add sys 1 (Gphi)
N = addcontsys(N,2,Gx,1,Q);         % Add sys 2 (Gx)
N = adddiscsys(N,3,PID1,2,1);       % Add sys 3 (PID1) to node 1
N = adddiscsys(N,4,PID2,[3 1],2);   % Add sys 4 (PID2) to node 2
N = adddiscexec(N,4,[],[3 1],3);    % Add exec of sys 4 (PID2) to node 3
```

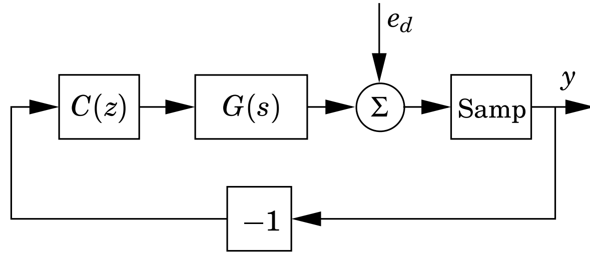

Figure 13 The signal model to calculate the sensitivity spectral density (i.e., the spectral density of y when e_d is white noise).

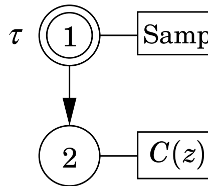

Figure 14 The timing model of the spectral density example.

```
N = calcdynamics(N);                    % Calculate internal dynamics
J = calccost(N)                         % Calculate cost
```

4.4 Spectral Density Calculation

The following example computes the influence of jitter on the sensitivity function for a control system. The sensitivity function for a control system with a plant G and a controller C is defined as $S = \frac{1}{1+CG}$. For a randomly time-varying system, though, this definition cannot be used.

The idea in this example is to form a system which is driven by white noise e_d at the output of the process G (see Figure 13). The spectral density of the output y may then be interpreted as a kind of sensitivity function for the stochastic system.

The example system is a continuous $G(s) = \frac{1}{s^2}$ which is controlled by a LQG-designed controller $C(z)$. The process is sampled periodically, but there is a random delay τ between the process and the controller (see Figure 14). The delay is uniformly distributed between zero and $\tau_{\max}$. The sensitivity spectral density for $\tau_{\max}$ between zero and h (for different amounts of jitter) is plotted in Figure 15.

The Matlab script for the computations is given below:

```
% Jitterbug example: spectdens.m
% ==============================
% Compute the sensitivity power spectral density with jitter

s = tf('s');
G = 1/s^2;              % The process is a double integrator

h = 0.25;
delta = h/10;
```

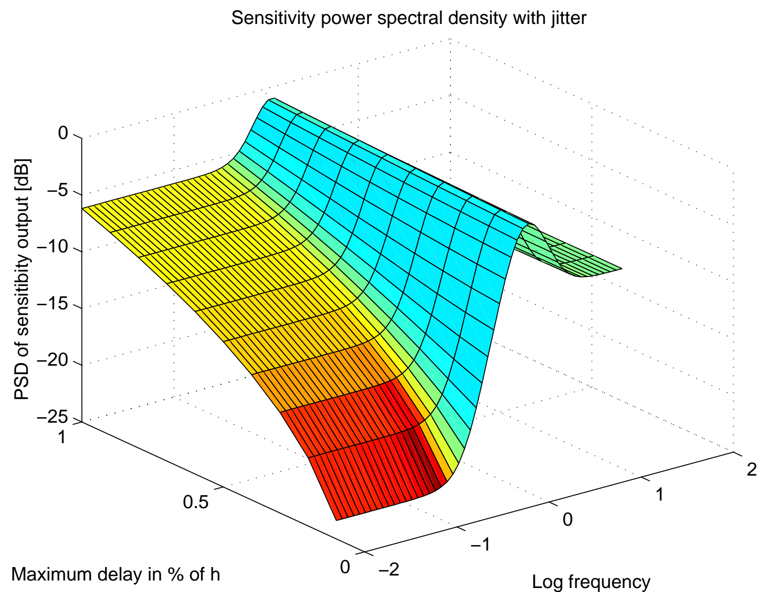

Figure 15 The sensitivity spectral density from the example.

```
Mvec = [];
delays = (1:round(h/delta))/round(h/delta);
for delay = delays
  Ptau1 = ones(1,delay*round(h/delta)+1); % Uniform delay
  Ptau1 = Ptau1/sum(Ptau1);

  Q = diag([1 1]);
  R1 = 1;
  R2 = 1;

  Samp = 1;                              % Sampler system
  C = lqgdesign(G,Q,R1,R2,h,h*delay/2); % Design an LQG controller

  N = initjitterbug(delta,h);           % Initialize Jitterbug
  N = addtimingnode(N,1,Ptau1,2);       % Add node 1
  N = addtimingnode(N,2);               % Add node 2
  N = addcontsys(N,1,G,3,Q,[],1);       % Add sys 1 (G) with output noise
  N = adddiscsys(N,2,Samp,1,1);         % Add sys 2 (Samp) to node 1
  N = adddiscsys(N,3,C,2,2);            % Add sys 3 (C) to node 2

  N = calcdynamics(N);                  % Calculate internal dynamics
  [J,P,F] = calccost(N);                % Calculate spectral densities
  H = F{2};                             % y is the second output (sys 2)
  w = logspace(-2,log10(pi/h),50);
  M = bode(H,w);
  M = squeeze(M);
  Mvec = [Mvec M];
end
figure
surfl(log10(w),delays,10*log10(Mvec)')
title('Sensitivity power spectral density with jitter');
xlabel('Log frequency');
ylabel('Maximum delay in % of \ith');
zlabel('PSD of sensitibity output [dB]')
```

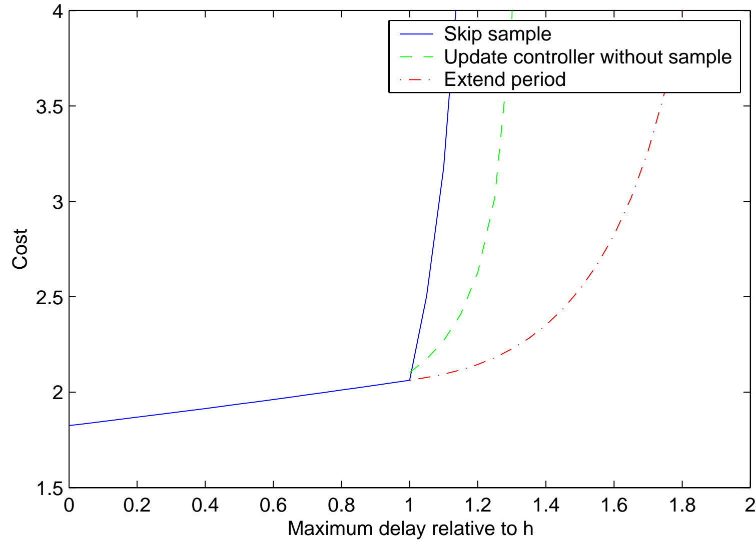

Figure 16 The costs in the overrun example.

4.5 Overrun Handling Methods

This example presentes (a rather long) script that compares three ways of handling long delays in a control system. The problem is what to do if the controller does not get a process sample in time. Three approaches are tested:

a) Do not update the controller, and use the old control signal.

b) Update the controller state and control signal based on no input. This is not done by feeding zeros to the observer, but rather by disconnecting the input part of the observer.

c) Extend the sample period until the sample does arrive. This creates an aperiodic system, and an iterative solver has to be used.

This kind of problem can also be interpreted as a computation time problem, where the computation of some control signal may take long enough time to miss a deadline.

The set-up is as follows. A plant $G(s) = \frac{1}{s(s^2+2\zeta\omega s+\omega^2)}$ with $\zeta = 0.2$ and $\omega = 1$ is to be controlled by an LQG regulator. The controller is calculated for the mean time delay using the function `lqgdesign()`. As for the delay, it is uniformly distributed between 0 and $\tau_{\max}$, where $\tau_{\max}$ is swept from 0 to 2h (i.e. two sample periods).

The three cases are compared in Figure 16. As expected, using the old control signal gives the worst performance, while extending the control period until the control signal is produced gives the best results.

The Matlab script for the computations is given below. Note that the iterative solver is very much slower than the algebraic solver, and is only used when the system is aperiodic.

```
% Jitterbug example: overrun.m
% ============================
% Compare three overrun handling methods for a control system with
% delayed samples. The plant to be controlled is an integrator with a
% resonance (a third order system). The controller is an LQG
% controller, designed for the mean time delay. The delay for the
% sample from the plant is uniformly distributed between 0 and
% tau_max, which varies between 0 and 2h.
%
% When a sample is delayed more than one period,
% the controller will:
% Case 1) Not be updated at all
% Case 2) Let its observer run without input
% Case 3) Extend the period until the sample arrives (aperiodic system).
%
% The last case is very computationally intensive as it requires an
% iterative solver.

s = tf('s');

zeta = 0.2;
omega = 1;
G = 1/s/(s^2+2*zeta*omega*s+omega^2); % The process
Samp = 1;

h = 0.25;
delta = h/20;

Q = diag([1 1]);
R1 = 1;
R2 = 0.001;

clf;
hold on;
for mode = 1:3
  slots = round(h/delta);
  if mode == 1
    delays = (0:2*slots)/slots;
  else
    delays = (slots:2*slots)/slots;
  end
  Jvec = [];

  for delay = delays
    % All three modes do the same thing for delay < 1.
    if (mode < 2 | delay >= 1)
      Ptau = ones(1,round(delay*slots)+1); % Uniform delay
      Ptau = Ptau/sum(Ptau);
      if (mode == 2)
        if (size(Ptau,2) > slots+1)
          Ptau = [Ptau(1:slots) sum(Ptau(1,slots+1:end))];
        end
      end
      Pwait = zeros(round(slots*delay)+1,slots+1);
      for d = 1:(slots*delay+1)
        if (d > slots+1)
```

```
        Pwait(d,1) = 1;
      else
        Pwait(d,slots-d+2) = 1;
      end
    end

    % Create optimal controller based on mean delay
    [C,L,Obs,K,Kbar,Gd] = lqgdesign(G,Q,R1,R2,h,delay*h/2);
    % Create optimal controller based on observer with no input
    Cnodata = ss(Gd.A-Gd.B*L,Gd.B*0,-L,Gd.D*0,h);

    % Add different timing nodes depending on mode.
    if (mode == 3)
      N = initjitterbug(delta,0);       % Aperiodic system
    else
      N = initjitterbug(delta,h);       % Periodic system
    end
    if (mode == 2)
      N = addtimingnode(N,1,Ptau,[2*ones(1,round(h/delta)) 3]);
    else
      N = addtimingnode(N,1,Ptau,2);
    end
    if (mode == 3)
      N = addtimingnode(N,2,Pwait,1);
    else
      N = addtimingnode(N,2);
    end
    if (mode == 2)
      N = addtimingnode(N,3);
    end

    N = addcontsys(N,1,G,3,Q,R1,R2);    % Add sys 1 (G)
    N = adddiscsys(N,2,Samp,1,1);       % Add sys 2 (Samp) to node 1
    N = adddiscsys(N,3,C,2,2);          % Add sys 3 (C) to node 2
    if (mode == 2)
      N = adddiscexec(N,3,Cnodata,2,3); % Add exec of sys 3 (C) to node 3
    end
    N = calcdynamics(N);                % Calculate internal dynamics
    J = calccost(N)                     % Calculate cost
    Jvec = [Jvec J];
    if J == Inf
      delays = delays(1:find(delays==delay));
      break; % Skip remaining delays
    end
  end
end
if (mode == 1)
  plot(delays,Jvec,'b');
  Jvec1 = Jvec;
elseif (mode == 2)
  plot(delays(find(delays >= 1)),Jvec,'g');
  Jvec2 = Jvec;
else
  plot(delays(find(delays >= 1)),Jvec,'r');
  Jvec3 = Jvec;
end
```

```
  Jvec = [];
  pause(0);
end
hold off;
legend('Skip sample', 'Update controller without sample', 'Extend period');
xlabel('Maximum delay relative to h');
ylabel('Cost');
axis([0 2 1.5 4])
```

5. Command Reference

A summary of the available JITTERBUG commands are given in Table 1.

Command	Description
`initjitterbug`	Initialize a new JITTERBUG system.
`addtimingnode`	Add a timing node.
`addcontsys`	Add a continuous-time system.
`adddiscsys`	Add a discrete-time system to a timing node.
`adddiscexec`	Add an execution of a previously defined discrete-time system.
`adddisctimedep`	Add time-dependence to a previously defined discrete-time system.
`calcdynamics`	Calculate the internal dynamics of a JITTERBUG system.
`calccost`	Calculate the total cost of a JITTERBUG system and, for periodic systems, calculate the spectral densities of the outputs.
`lqgdesign`	Design a discrete-time LQG controller for a continuous-time plant with a constant time delay and a continuous-time cost function.

Table 1 Summary of the JITTERBUG commands.

initjitterbug

Purpose

Initialize a new JITTERBUG system.

Syntax

```
N = initjitterbug(delta,h)
```

Description

Initialize a new JITTERBUG system with a given time-grain and period.

Arguments

`delta`	The time grain (in seconds). The computations in JITTERBUG are completely based on this discretization. Computation times and memory scale inversely proportionally to `delta`.
`h`	The period of the system (in seconds). Specify 0 if the system should be aperiodic.

Return Values

`N` The JITTERBUG system which must be passed to all other functions.

addtimingnode

Purpose

Add a timing node to a JITTERBUG system.

Syntax

```
N = addtimingnode(N,nodeid)
N = addtimingnode(N,nodeid,Ptau,nextnode)
N = addtimingnode(N,nodeid,Ptau,nextnodes,nextprobs)
N = addtimingnode(N,nodeid,Ptau,timedepnextnodes)
```

Description

Add a timing node to the JITTERBUG system `N`. The delay in the node is given by the discrete probability distribution `Ptau`. The next node to be visited can be either deterministic, random, or dependent on the total delay since the first node.

Note 1: The JITTERBUG system must have a node with ID 1. If the system is periodic, this will be the periodic node.

Note 2: If the total delay exceeds the period, the execution will restart in the periodic node (if the system is periodic).

Arguments

`N`	The JITTERBUG system to add this timing node to.
`nodeid`	The ID of this timing node (a positive integer).
`Ptau`	The delay probability vector. `Ptau(1)` is the probability of a delay of `0*delta` seconds, `Ptau(2)` is the probability of a delay of `1*delta` seconds, etc. If omitted, the system will stay in this node until the next period.
`nextnode`	The next node to be visisited, after the delay in this node has elapsed.
`nextnodes`	A vector of possible next nodes to be visited.
`nextprobs`	A vector specifying the probabilities for each of the nodes in `nextnodes` to be visited next.
`timedepnextnodes`	A vector of next nodes to be visited depending on the total delay since the first node (including the delay in this node).

addcontsys

Purpose

Add a continuous-time system to a JITTERBUG system.

Syntax

```
N = addcontsys(N,sysid,sys,inputid)
N = addcontsys(N,sysid,sys,inputid,Q,R1,R2)
```

Description

The continuous-time system can be given in state-space form or in transfer-function (or zero-pole-gain) form.

In *state-space form*, the system is described by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + v(t) \\ y^0(t) &= Cx(t) && \text{(continuous output)} \\ y(t_k) &= y^0(t_k) + e(t_k) && \text{(measured discrete output)} \end{aligned}$$

where v is a continuous-time white-noise process with zero mean and covariance[1] R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

In *transfer-function form*, the system is described by

$$\begin{aligned} y^0(t) &= G(p)\big(u(t) + v(t)\big) && \text{(continuous output)} \\ y(t_k) &= y^0(t_k) + e(t_k) && \text{(measured discrete output)} \end{aligned}$$

where $G(p)$ is a strictly proper transfer function, v is a continuous-time white-noise process with zero mean and covariance R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

Note that the measured discrete output is only used when the system is connected to a discrete-time system.

[1] By this we mean that v has the spectral density $\phi(\omega) = \frac{1}{2\pi}R_1$.

Arguments

- `N` The JITTERBUG system to add this continuous-time system to.
- `sysid` A unique ID number for this system (pick any). Used when referred to from other systems.
- `sys` A strictly proper continuous-time LTI system in state-space, transfer function, or pole-zero-gain form. Internally, the system will be converted to state-space form.
- `inputid` A vector of 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. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).

Optional Arguments

- `Q` The cost matrix (default is zero).
- `R1` The state or input noise covariance matrix (default is zero).
- `R2` The measurement noise covariance matrix (default is zero). Note that measurement noise will only be added when the system is sampled by a discrete-time system. The measurement noise *will not* be included in the input cost of the connected discrete-time system. Also, the measurement nosie *will not* affect any connected *continuous-time* systems (see Figure 2).

Any optional arguments can be left as `[]` for default values.

Limitations

To avoid problems with algebraic loops and infinite variances, continuous-time systems with direct terms are not supported. Also, continuous-time output noise is not supported.

adddiscsys

Purpose

Add a discrete-time system to a JITTERBUG system.

Syntax

```
N = adddiscsys(N,sysid,sys,inputid,nodeid)
N = adddiscsys(N,sysid,sys,inputid,nodeid,Q,R)
```

Description

The discrete-time system can be given in state-space form or in transfer-function form.

In *state-space form*, the system is described by

$$\begin{aligned} x(t_{k+1}) &= Ax(t_k) + Bu(t_k) + v(t_k) \\ y^0(t_k) &= Cx(t_k) + Du(t_k) \quad \text{(discrete output)} \\ y(t_k) &= y^0(t_k) + e(t_k) \quad \text{(measured discrete output)} \end{aligned}$$

where v and e are discrete-time white-noise processs with zero mean and covariance

$$R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T .$$

The cost of the system is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix. Note that $x(t)$ and $y^0(t)$ are piecewise constant signals, while $u(t)$ may be a continuous signal.

In *transfer-function form*, the system is described by

$$\begin{aligned} y^0(t_k) &= H(q)\big(u(t_k) + v(t_k)\big) \quad \text{(discrete output)} \\ y(t_k) &= y^0(t_k) + e(t_k) \quad \text{(measured discrete output)} \end{aligned}$$

where $H(q)$ is a proper transfer function, and v and e are discrete-time white-noise processs with zero mean and covariance

$$R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T .$$

The cost of the system is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix. Again, note that $y^0(t)$ is a piecewise constant signal, while $u(t)$ may be a continuous signal.

Arguments

`N`	The JITTERBUG system to add this discrete-time system to.
`sysid`	A unique ID number of this system (pick any). Used when referred to from other systems.
`sys`	A discrete-time LTI system in state-space or transfer function form, or a double/matrix (interpreted as a static gain transfer function). Internally, the system will be converted to state-space form.
`inputid`	A vector of 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. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).
`nodeid`	The timing node where this discrete-time system should be executed. If you want the same system to be executed in further nodes, use `adddiscexec`.

Optional Arguments

`Q`	The cost matrix (default is zero).
`R`	The noise covariance matrix (default is zero). Added each time the system is updated. Note that noise may also enter the system from the output nose of another system.

Any optional arguments can be left as `[]` for default values.

Remark

The input cost is really defined on whatever signal is used as input. If the input signal is continuous, the continuous cost (*not* sampled) will be calculated. If you really want the sampled cost, insert a sampling discrete-time system in between.

See Also

`adddiscexec`, `adddisctimedep`

adddiscexec

Purpose

Add an execution of a previously defined discrete-time system.

Syntax

```
N = adddiscexec(N,sysid,sys,inputid,nodeid)
```

Arguments

`N`	The JITTERBUG system.
`sysid`	The ID of the discrete-time system.
`sys`	A discrete-time LTI system or `[]` for the same dynamics as before. To ensure that the same state vector is used internally, both this and the original system should be given in state-space form.
`inputid`	A vector of 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. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).
`nodeid`	The ID of the timing node where this discrete-time system should be executed again.

Remark

It is not possible to change the cost or the noise of the system.

See Also

`adddiscsys`, `adddisctimedep`

adddisctimedep

Purpose

Add time-dependence to a previously defined discrete-time system.

Syntax

```
N = adddisctimedep(N,sysid,sys,timestep)
```

Description

Makes the dynamics of the discrete-time system with ID `sysid` time-dependent. The system model `sys` will be used for all delays greater than or equal to `timestep*delta` seconds since the first timing node (unless another definition overrides for longer delays).

Arguments

`N`	The JITTERBUG system.
`sysid`	The ID of the discrete-time system.
`sys`	A discrete-time LTI system describing the new dynamics. To ensure that the same state vector is used internally, both this and the original system should be given in state-space form.
`timestep`	The system model `sys` will be used for all delays greater than or equal to `timestep*delta` seconds since the first timing node.

Remark

It is not possible to change the cost or the noise of the system.

See Also

`adddiscexec`, `adddiscsys`

calcdynamics

Purpose

Calculate the internal dynamics of a JITTERBUG system.

Syntax

```
N = calcdynamics(N)
```

Description

Calculate the total system dynamics for the JITTERBUG system `N`. The continuous-time noise, the continuous-time cost functions, and the continuous-time systems are sampled with the time grain `delta`. The resulting system description is stored in `N.nodes`.

This function must be called before `calccost`.

Arguments

`N` The JITTERBUG system.

See Also

`calccost`

calccost

Purpose

Calculate stationary variance, cost, and output spectral densities of a JITTERBUG system.

Syntax

```
[J,P,F] = calccost(N)
[J,P,F] = calccost(N,options)
```

Description

Calculate the stationary variance and cost of the JITTERBUG system `N`. For periodic systems, also compute the (discrete-time) spectral densities of all outputs in the periodic node.

If the system is periodic, the solution is calculated algebraically, by solving a linear system of equations. If the system is aperiodic, an iterative solver is used.

This function must be called after `calcdynamics`.

Arguments

`N`	The JITTERBUG system.
`options`	For aperiodic systems, `options` is a struct with any of the following fields:
	`accuracy` — The iterative solver will quit whenever the relative cost change for one time step is less than this. Default is `1e-7`.
	`horizon` — The horizon over which the cost is averaged. May be Inf. Default is the maximum system period.
	`print` — Enable printouts. Default is `1` (on).

Return Values

`J`	The cost (`Inf` if unstable).
`P`	The stationary variance in the periodic node (`Inf` if unstable).
`F`	The spectral densities of the outputs (in the order they were defined). The spectral density of each output is returned as a discrete-time system $F(z)$ such that $\phi(\omega) = F(e^{i\omega h})$. Use e.g. `bodemag(F{1})` to plot the spectral density of the output of the first defined system.

See Also

`calcdynamics`

lqgdesign

Purpose

Design a discrete-time LQG controller for a continuous-time plant with a constant time delay and a continuous-time cost function.

Syntax

```
[ctrl,L,obs,K,Kf,sysd] = lqgdesign(sys,Q,R1,R2,h,tau)
```

Description

Design a discrete-time LQG controller with direct term for the continuous-time system `sys` assuming a constant sampling interval `h` and a constant time delay `tau`. The system can be given in state-space form or in transfer-function (or zero-pole-gain) form.

In *state-space form*, the system is described by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t-\tau) + v(t) \\ y(t_k) &= Cx(t_k) + e(t_k) \end{aligned}$$

where τ is a constant time delay, v is a continuous-time Gaussian white-noise process with zero mean and covariance R_1, and e is a discrete-time Gaussian white-noise process with zero mean and covariance R_2. The noise processes v and e are assumed to be independent. The sampling instants are given by $t_k = kh$. The cost function to be minimized by the controller is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

In *transfer-function form*, the system is described by

$$\begin{aligned} y^0(t) &= G(p)\big(u(t-\tau) + v(t)\big) \\ y(t_k) &= y^0(t_k) + e(t_k) \end{aligned}$$

where τ is a constant time delay, $G(p)$ is a strictly proper transfer function, v is a continuous-time white-noise process with zero mean and covariance R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

The resulting controller has the form

$$\begin{aligned} u(k) &= -L\hat{x}_e(k \mid k) \\ \hat{x}_e(k \mid k) &= \hat{x}_e(k \mid k-1) + K_f\big(y(k) - C_e\hat{x}_e(k \mid k-1)\big) \\ \hat{x}_e(k+1 \mid k) &= \Phi_e\hat{x}_e(k \mid k-1) + \Gamma_e u(k) + K\big(y(k) - C_e\hat{x}_e(k \mid k-1)\big) \end{aligned}$$

where $\hat{x}_e(k) = \begin{pmatrix} \hat{x}(k) \\ u(k-1) \end{pmatrix}$.

Arguments

- `sys` A strictly proper continuous-time LTI system in state-space, transfer function, or pole-zero-gain form. Any delay specified in this system will be ignored. Use the `tau` argument instead.
- `Q` The cost matrix.
- `R1` The state or input noise covariance matrix.
- `R2` The measurement noise covariance matrix.
- `h` The sampling period of the controller (in seconds).
- `tau` The time delay (in seconds), $0 \leq$ `tau` $\leq$ `h`.

Return Values

- `ctrl` The complete LQG controller as a discrete-time LTI system.
- `L` The state feedback gain vector.
- `obs` The observer as a discrete-time LTI system.
- `K, Kf` The observer gains.
- `sysd` The sampled delayed plant, `sysd = ss(Phie,Gammae,Ce,0,h)`.

6. References

Lincoln, B. and A. Cervin (2002): “Jitterbug: A tool for analysis of real-time control performance.” In *Proceedings of the 41st IEEE Conference on Decision and Control*.

Nilsson, J. (1998): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.