Lecture 5: Simulating Dynamical Systems in MATLAB

Numerically simulating dynamical systems allows visualizing their behavior and which is especially useful when analytical solutions are intractable. In this lecture we review some numerical techniques (and their implementation in MATLAB) simulating the dynamical systems we've studied so far. Many mathematical programming languages (i.e., Mathematica, Maple, numpy/scipy, etc.) have similar functions that facilitate simulations. Some dynamical system simulators use a graphical interface (e.g., Simulink, Modelica) which can be more intuitive when working with block diagrams. We'll begin by discussing simulation of LTI systems (`lsim` in MATLAB), followed by fixed time-step (Euler's method) and variable time-step approachs for nonlinear systems (`ode45`). Lastly, we'll discuss discrete-time simulations.

Simulation LTI Dynamics and Measurements

Linear time-invariant (LTI) systems can be simulated using the variation of constants formula discussed previously. Recall that for LTI systems the state-transition matrix simplifies to the matrix exponential. In MATLAB, an LTI system can be simulated by defining a system object using the `ss` command then using `lsim` to simulate the system. The syntax of each command is as follows:

- `sys = ss(A,B,C,D)`: creates a continuous-time state-space model object where `A,B,C,D` are real-valued matrices corresponding to the LTI system

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$$
$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u}$$

- `[yhist,thist,xhist] = lsim(sys,u,t,x0)` plots the simulated time response of the system model `sys` to the input history `(t,u)` where `x0` is a vector of initial state values. The vector `t` specifies the time samples for the simulation. For single-input systems, the input signal `u` is a vector of the same length as `t`. This syntax also returns the state trajectories `xhist`, the time samples used for the simulation in `thist`, and the output history `yhist`.

Example. The code below illustrates simulating the linear system

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{1}$$

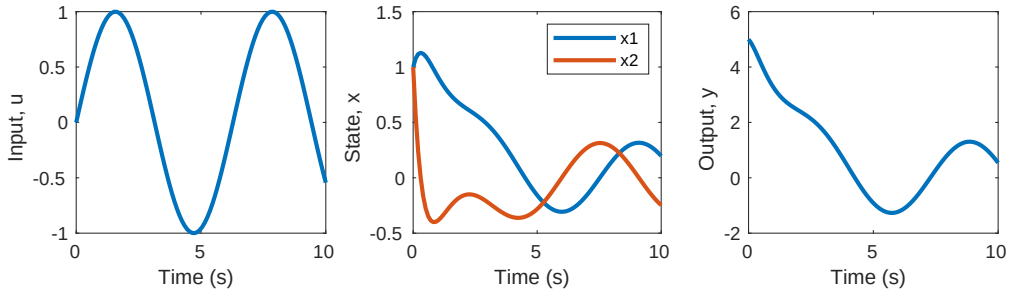
$$y = \begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2}$$

subject to a sinusoidal input $u(t) = \sin t$ from $\boldsymbol{x}(t_0) = [1,1]^{\mathrm{T}}$ at time $t_0 = 0$ to time $t_f = 10$ sec.

```
A = [0 1;−2 −3]; % define LTI system matrices, (A,B,C,D)
B = [0; 1];
C = [4 1];
D = 0;
```

```
x0 = [1; 1]; % initial condition
sys = ss(A,B,C,D); % create matlab system object
t = linspace(0,10); % setup control input u(t)
u = sin(t);
[yhist,thist,xhist] = lsim(sys,u,t,x0); % simulate system
```

Which can be plotted to produce:



Simulating Nonlinear Continuous Dynamics and Measurements (Fixed time-step)

Euler's method provides an approximate solution (i.e., a state and time history) for any ODE (linear or nonlinear) and input $\boldsymbol{u}(t)$. Rather than obtaining a mathematical solution to the free or forced response, Euler's method gives a sequence of values of the state $\boldsymbol{x}_0, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N$ at corresponding time instants $t_0, t_1, \cdots, t_N$ that are spaced apart by a regular interval called the *time-step*, h, as shown in Fig. 1. That is, $t_1 - t_0 = h$ and $t_2 - t_1 = h$, and so on. The final desired simulation time T corresponds to the final time instant, $t_N = T$. (Warning: the notation is similar to that of a discrete-time system but recall that discrete-time systems can be exact whereas Euler's method is an approximation of the continuous dynamics.)

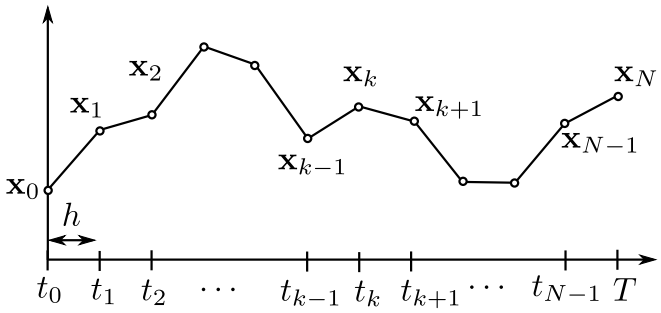


Figure 1: An approximated solution to an initial value problem obtained using Euler's method

It is instructive to consider a scalar first-order system. The basic idea in Euler's method is to construct a series of "straight-line" (i.e., constant state-rate) approximations of the solution over each time-step, starting from the initial state x_0 and the initial time t_0. First, the ODE is re-written in the form

$$\frac{dx}{dt} = f(t, x, u) \ .$$

Then, by approximating the change in time $dt \approx t_{k+1} - t_k = h$ and the change in state $dx \approx x_{k+1} - x_k$ the ODE is converted from continuous to discrete form as

$$\frac{x_{k+1} - x_k}{t_{k+1} - t_k} \approx f(t_k, x_k, u_k) \tag{3}$$

$$\implies x_{k+1} = x_k + f(t_k, x_k, u_k) \cdot h \tag{4}$$

When simulating the next step (t_{k+1}, x_{k+1}) we compute the slope of the line according to $f(t_k, x_k, u_k)$ evaluated with the current step (t_k, x_k). Then, the process is repeated. Graphically, the slope $f(t_k, x_k, u_k)$ at each time-step is a tangent line to the *slope field* or the *integral curves* that pass through (t_k, x_k, u_k) as shown in Fig. 2. In reality, the ODE derivative varies smoothly and the

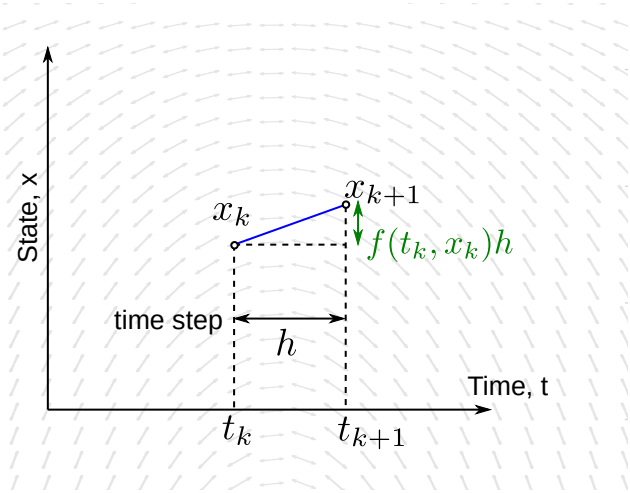


Figure 2: The grey arrows in the background represent the slope field and have a slope given by $f(t, x)$ (that is the right-hand side of the ODE $\dot{x} = f(t, x)$ ignoring the control input). When using Euler's method the slope over each time-step is determined from this slope field evaluated at the current time-step and state, (t_k, x_k).

straight-line approximation introduces some error into the next state x_{k+1} at each iteration. In turn, the (slightly incorrect) state, x_k, causes the next iteration to use a (slightly incorrect) slope $f(t_k, x_k)$ which further compounds the error of the simulation. For small enough time-steps this may not be a major concern. However, this approximation becomes worse if the system is highly nonlinear, the time-step h is large (see Fig. 3), or if the system is simulated for a long period of time (causing even small errors to accumulate).

Euler's Method (Algorithm). Begin by writing the system to be simulated as a set of first order differential equations:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \tag{5}$$

with the initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$. Then proceed with the following steps:

1. Define an integer index k that we will increment by one as the algorithm proceeds. At the initial "zeroth" step the initial conditions provide the time and state. That is, at $k = 0$ we have t_0 and $\boldsymbol{x}_0$ given. The remaining steps simulate the system for the $(k+1)$th step, assuming the kth previous step is already simulated. (See note below about choice of initial index being either $k = 0$ or $k = 1$.)

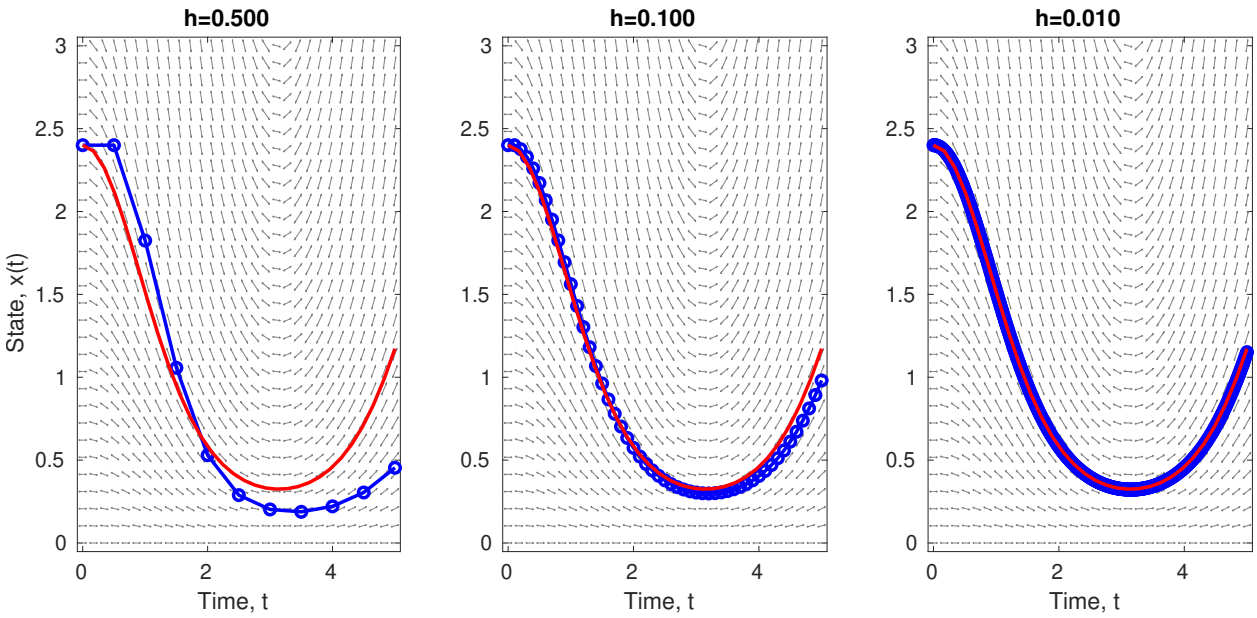


Figure 3: The red curve is the exact solution of an initial value problem, whereas the blue curves represent numerical approximations obtained with Euler's method using an increasingly smaller step-size, h.

2. Compute the state-rate (i.e., the derivative $\dot{\boldsymbol{x}}$) at the previous step k. That is, evaluate the right-hand side of (5) with t_k, $\boldsymbol{x}_k$, and a control input $\boldsymbol{u}_k$ to obtain the vector $\boldsymbol{f}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k)$. The control input $\boldsymbol{u}_k$, if present, may be based on a feedback control law (e.g., $\boldsymbol{u}_k = -\boldsymbol{K}\boldsymbol{x}_k$) or it may be obtained/interpolated from a pre-determined control history). Then evaluate the next time-step as

$$t_{k+1} = t_k + h \tag{6}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{f}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k) \cdot h \tag{7}$$

3. Increment k (first time around this will be incremented from $k = 0$ to $k = 1$).

4. Repeat Steps 2-3 for $k = 1, \cdots, N-1$ until you've simulated the final $(k+1)$th step t_{k+1} and x_{k+1} where $t_{k+1} = T$ and T is the terminal time of the simulation.

Note: Some computer programming languages index vectors either starting from 1 (e.g., MATLAB) or from 0 (e.g., in C++). In either case there will be a total of $T/h + 1$ time/state value pairs in the simulation (including the initial condition). When starting from index 1, as in MATLAB, we end up with a time history vector `[ t(1), t(2), ... , t(N) ]` and `[ x(1), x(2), ... , x(N) ]` where `t(N)= T` where `T` is the final simulation time, `N=T/h+1`, `t(1) = 0` is the initial time, and `x(1) = x0` is the initial state $x(0)$.

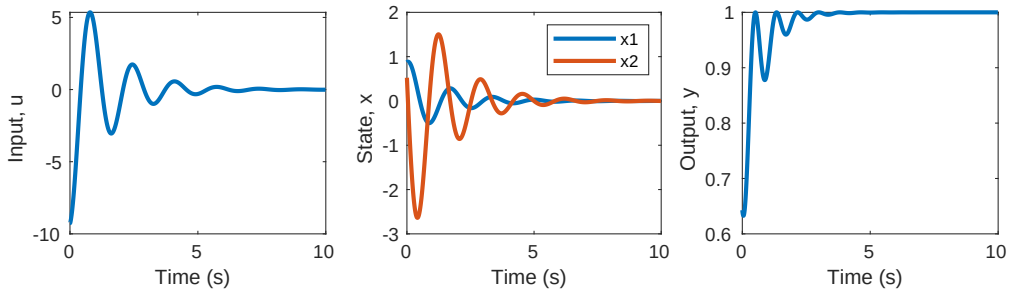
Example. Consider simulating the following nonlinear system using Euler's method:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -(d/m)x_2 - (g/L)\sin x_1 + u \end{bmatrix} \tag{8}$$

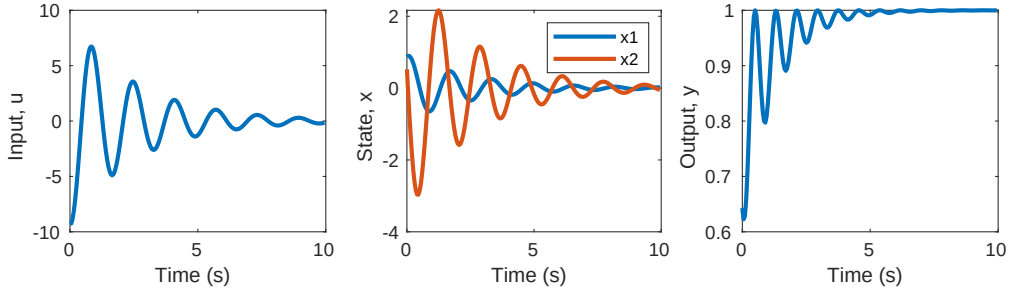
$$y = \cos x_1 \tag{9}$$

with the proportional-derivative feedback control law $u = -k_p x_1 - k_d x_2$ from $\boldsymbol{x}(t_0) = (\pi/180)[50, 30]^{\mathrm{T}}$ at time $t_0 = 0$ to time $t = 10$ sec.

```
d = 0.5; % model/control parametrs
g = 9.81;
L = 2;
m = 1;
kp = 10;
kd = 1;
x0 = [50; 30]*pi/180; % initial condition
T = 10; % final time
h = 0.01; % time-step

% simulate
N = T/h;
thist(1) = 0;
xhist(:,1) = x0;
yhist(1) = cos(x0(1));
uhist(1) = -kp*x0(1)-kd*x0(2);
for k = 2:1:N
    % time/state at previous step
    xkm1 = xhist(:,k-1); % x_{k-1}
    tkm1 = thist(k-1); % t_{k-1}
    uhist(k) = -kp*xkm1(1)-kd*xkm1(2); % feedback control
    xdot(1,1) = xkm1(2); % dynamics
    xdot(2,1) = -d/m*xkm1(2) - g/L*sin(xkm1(1)) + uhist(k);

    xhist(:,k) = xkm1 + xdot*h; % Euler's method
    thist(k) = thist(k-1) + h;
    yhist(k) = cos(xhist(1,k));  % output equation
end
```



If we change the time-step of integration from $h = 0.01$ to $h = 0.05$ we see a different response (below). For accurate results the time-step should be selected sufficiently small that small changes in h do not lead to large changes in the trajectory (i.e., the solutions are converging).



Runge-Kutta Methods. Selecting the time-step using Euler's method requires a balance between accuracy and computation time. Euler's method is a member of a wider class of numerical methods called *Runge-Kutta methods* that are used to simulate ODEs. All of the Runge-Kutta methods rely on a fixed-time step but differ in how they update the state. For example, the well known *Runge-Kutta-4 (RK4)* method uses:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{1}{6}(\boldsymbol{f}_1 + 2\boldsymbol{f}_2 + 2\boldsymbol{f}_3 + \boldsymbol{f}_4)h \tag{10}$$

$$t_k = t_n + h \tag{11}$$

where the four values $\boldsymbol{f}_1, \boldsymbol{f}_2, \boldsymbol{f}_3, \boldsymbol{f}_4$ are the state-rates evaluated at four unique points determined recursively as:

$$\boldsymbol{f}_1 = \boldsymbol{f}(t_k, \boldsymbol{x}_k) \tag{12}$$

$$\boldsymbol{f}_2 = \boldsymbol{f}(t_k + \frac{h}{2}, \boldsymbol{x}_k + \frac{h}{2}\boldsymbol{f}_1) \tag{13}$$

$$\boldsymbol{f}_3 = \boldsymbol{f}(t_k + \frac{h}{2}, \boldsymbol{x}_k + \frac{h}{2}\boldsymbol{f}_2) \tag{14}$$

$$\boldsymbol{f}_4 = \boldsymbol{f}(t_k + h, \boldsymbol{x}_k + h\boldsymbol{f}_3) \tag{15}$$

Notice that (10) includes a weighted average of the state-rates $\boldsymbol{f}_1$ through $\boldsymbol{f}_4$ that gives twice as much weight to $\boldsymbol{f}_2$ and $\boldsymbol{f}_3$.

Simulating Nonlinear Continuous Dynamics and Measurements (Variable time-step)

Variable time-step solvers allow the time-step h to change throughout the simulation while maintaining a desired level of accuracy/tolerance. For many ODEs, the step size can be relaxed (made bigger) in regions of the state-space where the system is varying slowly and smoothly while, conversely, tightening the step size (making it smaller) in regions with greater variability. One popular tool in MATLAB used for variable time-step solvers is `ode45`. This function implements the fourth order RK4 method but uses a RK5 method at each step as well. Comparing RK4 and RK5 the local error is estimated and the step size is reduced if needed.

We briefly review the syntax of using `ode45` below.

- `[thist,xhist] = ode45(odefun,tspan,x0,options).`
  - `thist` a vector providing output times from the simulation
  - `xhist` a matrix providing states at the corresponding output times from the simulation
  - `odefun` is the name of the ODE function to solve and it must be in the form `dxdt = odefun(t,x)`. The function can be provided in a separate file, at the bottom of the MATLAB script, or as an anonymous function directly. If the function `odefun` requires some additional parameters they can be specified using the syntax `@(t,x) odefun(t,x,A,B)` where `A,B` are the extra parameters.
  - `tspan` a vector (usually `tspan = [t0 T]` specifying the start and end times of the simulation where, by default, `thist` and `xhist` will output the solution at arbitrarily spaced time steps used during the evaluation process. Instead, tspan has more than two values then `ode45` will output the solution exactly at those time values provided.
  - `x0` the initial state of the system
  - `options` variable can be used to specify tolerances and other parameters

To recover the output history $\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ or a feedback control history $\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \ldots$ we may pass the resulting state/time pairs through our functions that define the state feedback control and the measured output. Alternatively, these values can be saved to a global variable during simulation.

Example. For an example, refer to the code below that simulates the nonlinear system

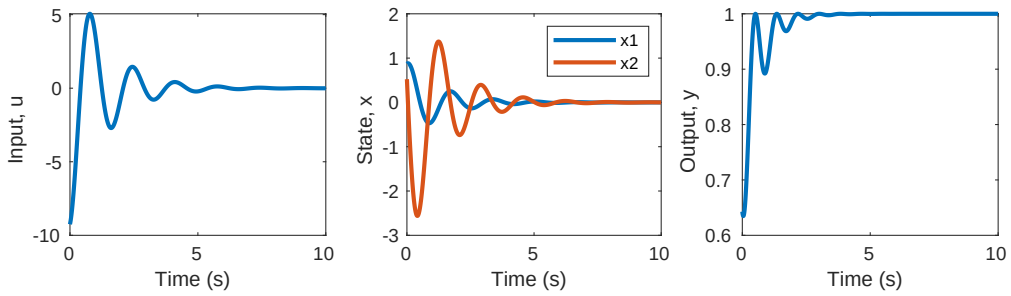
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ \frac{d}{m} x_2 - \frac{g}{L} \sin x_1 + u \end{bmatrix} \tag{16}$$

$$y = \cos x_1 \tag{17}$$

subject to a sinusoidal input as in the previous example. Note that the time-step parameter is only used to specify the output times (it is not used as part of the solution process).

```
params.d = 0.5; % model/control parametrs
params.g = 9.81;
params.L = 2;
params.m = 1;
params.kp = 10;
```

```
params.kd = 1;
x0 = [50; 30]*pi/180; % initial condition
T = 10; % final time
h = 0.01; % time-step

% simulate
tspan = [0:h:T]; % output values
options = odeset('RelTol',1E-8,'AbsTol',1E-8);
[thist,xhist] = ode45(@(t,x) pendulum(t,x,params),tspan,x0,options);
xhist = xhist';

% generate observations and recover control inputs
for i = 1:1:length(thist)
    x = xhist(:,i);
    yhist(i) = cos(x(1));
    uhist(i) = -params.kp*x(1)-params.kd*x(2);
end

% pendulum dynamics used by ode45
function xdot = pendulum(t,x,params)
    % pass in parameters
    d = params.d;
    g = params.g;
    L = params.L;
    m = params.m;
    kp = params.kp;
    kd = params.kd;
    % feedback control
    u = -kp*x(1)-kd*x(2);
    % dynamics
    xdot(1,1) = x(2);
    xdot(2,1) = -d/m*x(2) - g/L*sin(x(1)) + u;
end
```



Surprisingly, some ODEs exhibit the property that even a seemingly smooth system leads to numerical instabilities (oscillations in the resulting $\boldsymbol{x}(t)$) unless the step-size is chosen to be extremely small. Such systems are called *stiff systems* and solvers like `ode15s` are more suitable for them.

Simulating Discrete Dynamics and Measurements

When simulating discrete systems we have exact expressions that propagate the system forward in time. The system may be inherently discrete (e.g., a communication network) or we've already done some of the hard work ahead of time in bringing a continuous system into discrete form. In either case, once we have a discrete system, we can simulate it using the finite difference equations describing it directly:

$$\boldsymbol{x}_k = \boldsymbol{f}_{k-1}^d(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1}) \tag{18}$$
$$\boldsymbol{y}_k = \boldsymbol{h}(\boldsymbol{x}_k) \tag{19}$$

Example. Consider the following discrete-time system which was found by discretizing the continuous LTI example above

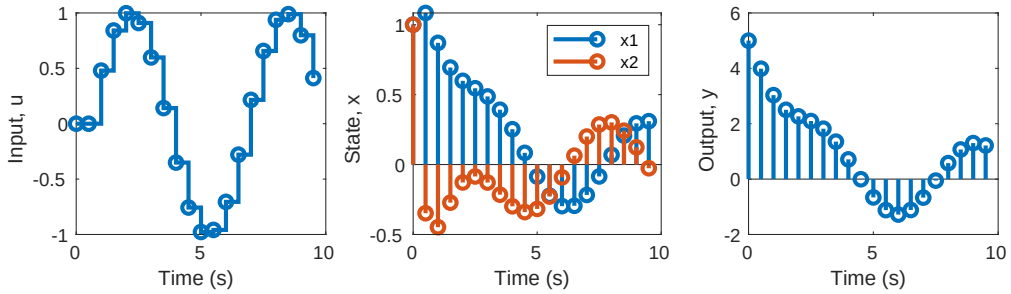
$$\boldsymbol{x}_{k+1} = \boldsymbol{F}\boldsymbol{x}_k + \boldsymbol{G}u_k \tag{20}$$
$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} 0.8452 & 0.2387 \\ -0.4773 & 0.1292 \end{bmatrix} \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \begin{bmatrix} 0.0774 \\ 0.2387 \end{bmatrix} u_k \tag{21}$$
$$y_h = \boldsymbol{H}\boldsymbol{x}_k \tag{22}$$
$$y_k = \begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} \tag{23}$$

subject to a sinusoidal input $u_k = \sin t_k$ from $\boldsymbol{x}_0 = [1,1]^{\mathrm{T}}$ at time $t_0 = 0$ to time $t = 10$ sec. For the following simulation we assume a zero order hold control input.

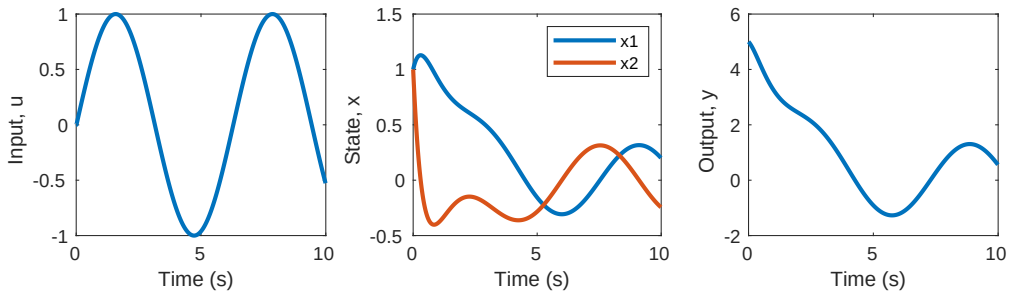
```
clear; close all; clc;
A = [0 1;−2 −3]; % define LTI system matrices, (A,B,C,D)
B = [0; 1];
C = [4 1];
D = 0;
x0 = [1; 1]; % initial condition
sys = ss(A,B,C,D); % create matlab system object
h = 0.5; % sampling time for discrete system
sys2 = c2d(sys,h); % convert continuous to discrete time
F = sys2.A;
G = sys2.B;
H = sys2.C;
T = 10;

% simulate
N = T/h;
```

```
thist(1) = 0;
xhist(:,1) = x0;
yhist(1) = H*x0;
uhist(1) = sin(0);
for k = 2:1:N
    % time/state at previous step
    xkm1 = xhist(:,k-1); % x_{k-1}
    tkm1 = thist(k-1); % t_{k-1}
    % feedback control
    ukm1 = sin(tkm1);
    % Discrete-time dynamics
    xhist(:,k) = F*xkm1 + G*ukm1;
    thist(k) = thist(k-1) + h;
    yhist(k) = C*xhist(:,k);
    uhist(k) = ukm1;
end
```

Occasionally discrete systems are plotted with the `stairs` and `stem` style plots to emphasize they are discrete.



If we increase the time-step (and obtain new corresponding $\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}$ matrices then we obtain a response that is very similar to the continuous response obtained earlier with `lsim`.



Simulating Hybrid Continuous Dynamics and Discrete Measurements/Controls

Hybrid systems (also called *sampled-data systems*) represent systems with continuous dynamics but discrete measurements or control inputs. This is an accurate description of a mechanical system combined with discrete-time measurements and controllers that "act continuously but thinks discretely".